Modular Detection of Feature Interactions Through Theorem Proving: A Case Study

by

Brian Roberts

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

Aug 2003

APPROVED:
Professor Kathi Fisler, Major Thesis Advisor
Professor George T. Heineman, Thesis Reader
Professor Michael Gennert, Department Head

Abstract

Feature-oriented programming is a way of designing a program around the features it performs, rather than the objects or files it manipulates. This should lead to an extensible and flexible "product-line" architecture that allows custom systems to be assembled with particular features included or excluded as needed. Composing these features together modularly, while leading to flexibility in the feature-set of the finished product, can also lead to unexpected interactions that occur between features. Robert Hall presented a manual methodology for locating these interactions and has used it to search for feature interactions in email[Hal00]. Li et al. performed automatic verification of Hall's system using model-checking verifications tools[LKF02a, LKF02b]. Model-checking verification is state-based, and is not well-suited for verifying recursive data structures, an area where theorem-proving verification tools excel.

In this thesis, we propose a methodology for using formal theorem-proving tools for modularly verifying feature-oriented systems. The methodology presented captures the essential steps for using modular techniques for modeling and verifying a system. This enables verification of individual modules, without examining the source code of the other modules in the system. We have used Hall's email system as a test case for validating the methodology.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

When users request additional features in their software, programmers naturally respond by implementing these features, increasing the marketability of the program. However, an impressive array of features comes at a price: complexity. The more features a program has, the more complex the program is, and thus, by extension, the more defects the code could potentially hide.

To reduce the complexity of a program, one could split the code into distinct modules [Par72]. Splitting code into modules facilitates code reuse and when updating on section of code, reduces the amount of code-redesign needed in other modules[1]. Object-oriented design takes this idea a step further, suggesting that modules (or classes) should encapsulate each actor or entity in a system [Mey91].

[1] This, of course, assumes that the modules are sufficiently independent of each other.

One problem inherent in object-oriented designs for certain types of systems is that the implementation of a system feature could be spread (or cross-cut) across multiple classes. For large systems, implementing or changing a single feature can take weeks and requires the cooperation of multiple programmers who are responsible for each class. Since the code is spread over multiple objects, the complexity of the resultant changes can introduce errors that are costly both to find and fix.

1.2 Feature-Oriented Programming

Feature-oriented programming introduces a different principle for organizing code. In this paradigm, modules encapsulate features, rather than actors or entities. Intuitively, a *feature* is a product characteristic, such as voice mail, that allows customers to distinguish between products. A system might involve three actors: the host delivery system, the database of user-information and the email client. The auto-response feature in the system would use the delivery system to send and receive email, retrieve data from the database, and use the email client to respond to messages. Instead of splitting the code across these three actors, feature-oriented programming keeps the code together in a single module. A feature-oriented module contains fragments of code that implement a single feature across actors of a system. Using Figure 1.1 as a guide, traditional object-oriented design would group together the code according to columns—all the code for the host (or other actor) would be grouped together in a module. Using a feature-oriented approach, code would be grouped according to rows—all code for a particular feature, such as auto-response, would be placed in the same module or component [HC01]. In either approach, linking together (composing) modules produces the final system.

	Host	**User**	**Network**
Auto-response	*auto-resp host-code*	*auto-resp user-code*	*auto-resp client-code*
Decrypt	*decrypt host-code*	*decrypt user-code*	*decrypt client-code*
Encrypt	*encrypt host-code*	*encrypt user-code*	*encrypt client-code*
Sign	*sign host-code*	*sign user-code*	*sign client-code*

Figure 1.1: Selected features and actors in an email system.

Separating well-defined features into components facilitates maintenance of the system, by allowing easy identification and removal of obsolete functionalities. In an email system, for example, the ability to encrypt and decrypt messages could be removed by simply removing the encrypt and decrypt modules from the system. Updating the system to provide support for a new encryption scheme would be equally straight forward.

Feature-orientation also facilitates programmers' comprehension of code. By design, each feature can be coded independently of other features; this allows one programmer to implement an entire feature, and be able to comprehend its functionality across the whole system. This simplifies updating or redesigning a feature at a later date—understanding one module of code is generally considered to be easier than understanding an entire system. Separate feature development also allows the implementation to take place in geographically- or chronologically-separate locations.

In general, feature-oriented programming derives these benefits from grouping together code that cross-cuts the actors or entities in a system; such architectures enable *separation of concerns.* Many researchers are exploring programming with cross-cutting, albeit under a variety of terms: aspect-oriented programming [Asp01], mixin layers [BO92], units [FF98], subject-oriented programming [OT99], and others [BA01, LLM99, Pre97] all propose similar architectures for similar motivations. Feature-oriented programming is not an ideal model for all software systems, but it works particularly well for product-line architectures [Gri00].

Product-line software [CNea98] draws its analogy from manufacturing, where large amounts of similar items are produced every day in cost-efficient manner. Although many items are produced, each item might be distinctly different from all the others, depending on which features are chosen for the finished product. For example, in automobile manufacturing, one car of a particular make might have red paint, the next blue; the first car has a cassette player installed, the second one a CD player; and so on, until the cars are both finished, that is, when all the features have been chosen. Product-line software is similar, the features of the system can be chosen with great flexibility, allowing many different combinations of features to arise. A software company might provide a product-line of word processors, some with grammar checking, document format conversion, and other features. Developing the family of products as a product-line amortizes development costs across many systems.

1.2.1 Validation: The Product-line Development Challenge

Product-line architectures are designed to provide flexible and cost-effective ways to create families of software systems. This flexibility comes at a price, however:

the number of ways to combine the features of a system grows almost factorially[2]. It becomes harder to effectively test the full set of products, because the testing methodology must verify the absence of errors in every possible feature combination. Separate testing of each possible product dramatically reduces, and can even eliminate, the cost-effectiveness of product-line development. Product-line techniques such as feature-oriented programming need validation techniques that operate at the granularity of individual features.

Testing combinations of feature modules exposes a related and well-known problem: features can unintentionally interact with each other (the *feature-interaction problem* [KK98]). For example, by including both message signing/verifying and encryption/decryption in an email system, we introduce the possibility that a message is properly signed, but cannot be verified because the message cannot be decrypted. As the possible number of combinations of features can reach factorial proportions, even if these feature interactions were always readily apparent to the human eye, automatic tools to check for such interactions are essential because of the large number of potential interactions.

Product-line development therefore comes with a research challenge: *we need validation techniques that allow each feature (module) to be checked independently, and that can efficiently locate potential interactions between features once we decide which ones to compose into a complete system.* The second part implies that we can't look for interactions by validating the whole composed system.

[2] For N features, there are only 2^N unordered combination possibilities. However, features can be executed in any order, resulting a factorial number of combinations.

1.3 Software Verification

Formal software verification is a form of validation that uses mathematics and logic to formally prove properties of programs. Formal verification is appealing for its ability to locate unusual defects in software designs. Defects, if left unfixed, “emerge to trouble the world through the entire lifetime of their software” [Ray03]. The earlier a defect is caught, the quicker and cheaper it is to correct. *Software Engineering Economics* [Boe81] estimates that it costs 20 times as much to fix a defect after the product has been released, as to fix the same defect during the design phase.

Verifying a design consists of several steps:

1. The design is mapped into a simple model—that is, one small enough to be easily verified. Often the model of the system is a high-level model that assumes that correct functioning of the lower levels, and attempts to prove the functionality of the system structure. For example, when verifying a compiler, the reading and parsing, as well as the writing of file could be abstracted, leaving the more interesting compiling step for verification. For communication protocols, state machines usually provide suitable models, leaving the network abstracted.

2. The model is compared (informally) with the actual design, checking whether—at least for particular inputs—the model and the design exhibit the same behavior. This initial testing is not exhaustive, but can serve as an important “sanity check” in writing the model. Formal comparisons between models and implementations are generally intractable for interesting systems.

3. Theorems are proven about the model that establish some interesting properties about the modeled system. For example, in an email system model, one might prove that all emails sent are eventually delivered; or that given the correct key, any encrypted email can be decrypted. Since the model and design are not proven be exactly the same manner, proving theorems on the model may seem like a fruitless search. However, the failure of theorems on the model can pinpoint areas where defects may exist. Likewise, proving theorems on the model can increase the confidence of all developers involved that the design is sound.

Verification approaches differ in the styles of models and proof techniques that they use. Two of the most popular approaches are theorem proving [KMM00] and model checking [CGP00a]. Theorem proving models designs and desired properties as programs or predicate logic and uses techniques such as natural deduction, resolution, or sequent calculus-style reasoning; theorem proving captures powerful properties, but requires considerable manual intervention from the user. Model checking maps a design into a state machine and needs properties as formulas in temporal logic; model checking is almost fully automated, but at the expense of being less powerful and expressive.

There are other trade-offs between using model-checking and formal theorem proving. Model checking can generate counter examples when properties do not hold, but is known to be limited in its ability to reason about data, due to its inability to handle unbounded, or even large, data. Model checking also suffers from a serious problem known as *state-explosion*: design models often have ordersof-magnitude more states than the checker can handle (modern checkers handle designs up to roughly 2^{150} states).

In theorem proving, given a system, a property that must hold, and necessary assumptions allowed, a programmer must build increasingly complex formulas to prove that the property holds in the system. While this method is able to cope well with many types of data, there are two major drawbacks: creating the needed formulas can be a time-consuming task; and the failure of the prover to accept a formula could mean many things. When a proof is not accepted by the prover, it could mean that the formula is incorrectly written, or that the system can not find the proof, or that the property will not hold in the system—it is hard to pinpoint the source of the error.

1.3.1 ACL2

The language used in this case study, ACL2 [KM97], which stands for "A Computational Logic for Applicative Common Lisp", is a theorem proving system written by Matt Kaufmann and J. Strother Moore. ACL2 is functional programming language similar to Common Lisp but has carefully chosen restrictions to facilitate proving theorems on the resulting programs. Like Common Lisp, it can be used to define functions (in order to model computing systems). Unlike Common Lisp, ACL2 can also be programmed to make assertions and prove theorems about the defined functions. ACL2 contains a first-order logic which contains induction that allows the user to prove complex theorems about the defined functions. For example, ACL2 is well-suited for proving theorems that, given constraints on its inputs, assure that a functions outputs are of a particular type.

ACL2 has been used to model and prove the correctness of many different kinds of systems, including microprocessors [BKM96, MLK, Rus99], programming languages [Moo99, BM96], and floating-point arithmetic [RF00]. Typically, ACL2 programs are simplified models of a system which emulate the behavior to be tested. In the email system, we abstracted the underlying network protocols, accessing the database, and other parts involved in message sending. Our focus is to prove the correctness of the features involved in processing and delivering mail, for example, the encryption and decryption features of the system.

1.3.2 Verification and Feature-Orientation

Given our need for validation techniques that can analyze features modularly and the need to detect feature interactions, a formal verification approach to feature validation is worth exploring. Feature-oriented modules offer an interesting benefit over standard object-oriented designs from the standpoint of verification. One challenge in verification is isolating the part of the design that actually matters for checking a property; failure to do so may make the verification task too hard or intractable to complete. With feature-orientation, this problem goes away because properties align with features (modules), thus allowing the verifier to easily identify the part of the model that's needed. For example, a property about auto-responding would need to be verified only against the auto-response feature. In a standard OO design (the column-oriented version of Figure 1.1), all three modules would contain some of the needed code, making it much harder to eliminate irrelevant details (experience shows that eliminating irrelevant code from within a module is hard to do in practice [SE98]).

What would a verification approach for feature-oriented programs look like? From the argument in Section 1.2.1, we know that we must be able to verify features independently and to locate possible interactions without verifying the whole system. Compositional verification [Pnu85] supports proving properties about modules in isolation and determining whether properties of modules imply properties of whole systems. It works by having each module come with interfaces written as the properties that it satisfies. Modular feature verification therefore requires some kind of interfaces for features that are (a) provable against the features and (b) sufficient to locate at least some kinds of feature interactions (it is well known that only some kinds of feature interactions can be detected by formal verification [Hal98a]).

1.4 Thesis Overview

The contribution of this thesis is a case study in, and proposed methodology for, verifying feature-oriented software using the formal theorem prover ACL2 [KM97]. The methodology supports verifying each feature individually, and using the resulting theorems to prove results for the system as a whole. While there has been some work on each of modular feature verification with model checking and using theorem proving for feature interaction, as described in the related work chapter (Chapter 2), no earlier work presents a modular theorem proving methodology for feature-oriented verification. The main challenge in the thesis is identifying the interface functions and theorems that enable modular verification. Using theorem proving lets us use richer models and check more powerful theorems than previous modular approaches to feature verification.

Chapter 2 presents other works that are related to this thesis, including Robert Hall's original paper on the system modeled in the case study [Hal00]; we have modeled Hall's system in the theorem-proving environment of ACL2 [KM97]. For each feature in the system, we have verified its correct functionality through theorem proving. An overview of the system, the resultant proofs, as well as proofs on combinations of features are put forth in Chapter 3. In modeling this system, we developed a general methodology for modeling feature-based systems that involve more complex forms of data and enabling compositional verification in a systematic manner. This general methodology is shown in Chapter 4. The conclusions drawn from modeling the system, verifying the system and creating the general methodology are covered in Chapter 5.

Chapter 2

Related Work

The use of modular verification techniques to verify feature interactions is a very new area of study. Works such as Felty and Namjoshi's "Feature specification and automated conflict detection" [FN98] relate to automatically detecting feature interactions in systems, but do not address verifying these interactions in a modular fashion. The few works which are directly relevant to this thesis are presented below.

2.1 Robert Hall

Robert Hall presented a manual methodology for searching for interactions and has used it to search for feature interactions in email [Hal00]. His email study includes 10 features: address-book, message signing, encryption, decryption, verifying message signatures, auto-responding to message, forwarding messages, re-mailing messages, filtering messages, and delivering of messages by the host. Hall modeled these features in a reactive system, and attempted to find interactions by manually inspecting numerous combinations of these features. In total, Hall inspected 100 (10 x 10) ordered pairs of features, and found 26 distinct interactions. Listed below are two typical examples of interactions that were found in the study:

- *SignMessage vs. VerifySignature*: If Bob sends a signed message to rjh, but rjh attempts to verify it using the wrong key, possibly because rjh's key has expired, or Bob somehow was tricked into using the wrong key, the *SignMessage*'s purpose, namely convincing rjh that Bob authored it, is defeated by rjh's inability to verify it.

- *EncryptMessage vs. VerifySignature*: Bob's *EncryptMessage* interferes with rjh's *VerifySignature* if rjh's *DecryptMessage* is unable to successfully decrypt the message, since in the configuration of the Client signing is performed prior to encryption.

The self-described "heuristic" methods used by Hall, which were "an attempt to blend the strength of human intuition with some systematicity afforded by formal modeling" [Hal00], are adequate for small cases. For more complex systems, Hall advised readers that "it may be useful to consider different test coverage metrics and tools." This verification work was done on the system as a whole, with no means to plan for how future features might interact with those currently in the system. The verification of the system was not completed in a modular manner, and in this respect differs from the work in this thesis. The case study in this thesis closely follows Hall's model, though, as he used a Lisp-like language with many similarities to Lisp-like ACL2.

2.2 Harry Li

Using a model-checking verification tool, Harry Li et al., created a modular verification methodology to modularly model and verify the interactions of Hall's email system [LKF02a]. Li discovered that it was possible to reproduce many of the interactions automatically, without laborious hand-simulations.

Li modeled each feature as a state machine and formally verified the properties needed for "proper" operation of each feature specified in the temporal logic of CTL [CGP00b]. Informal descriptions of two such properties are presented in Figure 2.1, as well as their corresponding CTL formulae.

Informal Description	**CTL Formula**
If one tries to verify a signature, then the message must be verifiable.	$AG[try{-}verify \rightarrow verifiable]$
If an auto-response is generated, the response eventually is delivered or received.	$AG[auto{-}response \rightarrow AF(deliver \bigvee received)]$

Figure 2.1: Example of a CTL property.

Li individually verified each feature through model checking. The interfaces needed for modular verification were generated automatically by his algorithms in the form of temporal logic formulas. The following is an example using a system with two features: F_1 and F_2. Li verified the system using the following methodology.

1. Model check each individual feature to prove its CTL properties.

2. Automatically determine interfaces for each feature that are sufficient to preserve each property after composition.
3. Prove that the feature F_1 satisfies the preservation constraints of feature F_2 by analyzing at most F_1, and not the composition of both F_1 and F_2.

This modular verification technique enabled Li to easy verify product-line architectures—each piece can be verified individually with out foreknowledge of any of the other features in the final system.

One of the main challenges in Li's work is that the model checker must reason with incomplete information: if a property to be checked against F_1 refers to some variable or attribute set in F_2, the model checker won't have all of the information it needs to verify the property in F_1. This situation comes up when checking constraints between interfaces, despite the fact that the properties generally align with the features. For example, using a system composed of an encryption and a forwarding feature, we wish to prove that once a message is encrypted, it is never sent out on the network un-encrypted. After verifying the encryption feature, we must ensure that the forwarding feature preserves the encryption property. However, since the forwarding feature contains no references to whether the message is encrypted, its value is unknown.

Li partially worked around the unknown property limitation by using three-valued logic and attempting the same verifications twice: once assuming all unknown attributes to be true; and once assuming all unknown attributes to be false. By using a model-checker, Li was limited in his ability to model the incomplete information inherently present in feature interfaces: although the values of all outside attributes are not always known, the model checker must assign a value for each attribute in the system. The work of this thesis avoids this problem because theorem provers are able to handle unknown values without resorting to checking only special cases. Handling unknown values is one of the main benefits of theorem provers that this thesis aims to exploit for feature verification.

2.3 Christian Prehofer

Using UML statechart diagrams as a modeling tool, Prehofer presented a work on building components by modular composition of features and discovering interactions between features [Pre02]. Robert Hall's email system was used as the problem set for presenting his methodology.

Each feature in the system was modeled using a statechart diagram, where transitions are labeled with the function calls which trigger that transition. For feature combination, he combines two statecharts into a single statechart with the combined functionality. When using statecharts for combining state machines, each individual state machine retains its original structure, and thus its functionality. In the case of an interaction between the statecharts, one feature is adapted in the presence of the other—a special case is defined to address where the features interact. Since his method of combining features is a manual and time-intensive process and must be applied to each new combination of features, it need not be further discussed.

Chapter 3

The Case Study

The model in this thesis is a simplified version of a feature-oriented system because the different actors do not execute a feature simultaneously. Instead, first the sender's mail host, then the network, then the receiver's mail host, carry out their tasks to process each action. This is consistent with Li's model, and focuses on the initial questions of what kinds of interfaces and methodologies support theorem proving for feature-oriented designs.

3.1 Modeling the System

3.1.1 Base System

The base system is the system, devoid of any added features. It includes the base functionality (infrastructure) that supports the addition of these features. For the email system, this includes: the core network for sending message, the existence of servers on the network, as well as infrastructure to enable adding features. Figure 3.1 shows the main code for the base system: the full code appears in Appendix A.

The `simulate-network` function is the main driver of the system—taking a list of actions to perform, a hash table of user environments and a hash table of host environments. An action is a listing of the command to perform, and the data necessary to perform that command. For example, to mail a message, the action would be: `('mail sender message)`, where the recipient is captured in the message. Each user environment is a hash table that captures the data necessary for the features used, such as a unique username and other attributes (an encryption key, an address-book, etc) associated with it. In the base system, this is initially empty. The host environments are analogous to the user environments, but contain data specific to each host in the system, a unique hostname, as well as its users associated with it. Additionally, each host and user has fields for keeping track of the messages that are both sent and received, to facilitate the verification of message properties.

The functions `do-action` and `do-action-cond` work together to process each action in turn. All actions are processable in the base system code. When new features are added, new code is inserted in the `do-action-*` helper functions; an example of this is shown in Section 3.1.2. With the base system, users can send and receive basic emails, but have no access to the encryption, decryption, or auto-response features.

```
(defun do-action-cond (action users hosts)
  (let ((type   (action-type action)))
    (mv-let (new-users new-hosts)
    (cond ((equal 'init    type)
   (do-init    action users hosts))
  ((equal 'command type)
   (do-command action users hosts))
  (t (begin
      (cw "Can't execute command ~x0~%" action)
      (mv users hosts))))
    (mv new-users new-hosts))))

(defun do-actions (actions users hosts)
  (cond ((endp actions) (mv 'end users hosts))
(t (let* ((action (car         actions))
  (rest   (cdr         actions)))
     (mv-let (new-users new-hosts)
     (if (member-equal (action-type action)
       '(mail send deliver))
 (do-action-mail action users hosts 10)
       (do-action-cond action users hosts))
     (do-actions rest new-users new-hosts ))))))

;;(simulate-network list[actions])
;;sets up initial env and calls recursive simulation func
(defun simulate-network (actions users hosts)
  (do-actions actions users hosts))
```

Figure 3.1: Code for base system.

3.1.2 Features

Our model of the email system presents each feature as a collection of functions, each implementing a particular functional interface.

Interface	**Description**
`-init`	Initializes data structures needed for feature.
`-command`	Provides an interface for data structures to be changed.
`-outgoing`	Contains logic for transforming messages being sent from (by) a user or via a host.
`-incoming`	Similar to `-outgoing`, for messages sent to a user. In additional, may refuse or respond to messages.

Figure 3.2: Interfaces implemented by features.

The functions of each feature are identified with a unique descriptive prefix, for example, the functions relating to auto-response all begin with ’`email-auto-`’. Additionally, the functions that each feature exposes are also identified, with unique suffixes, for example: ’-init’ and ’-incoming’ for handling state initialization and incoming message. In this manner, each function is readily identifiable as to both its feature and calling interface.

Consider the auto-response feature, similar in functionality to the Unix “vacation” [Uni], which sends a pre-specified message back to each correspondent the first time they send you an email while you are away. If a correspondent sends you a second email, no auto-response email is sent a second time. The source code in Figures 3.3 and 3.4 is a listing of the auto-response feature in ACL2. The feature consists of four functions called `email-auto-init`, `email-auto-command`, `email-auto-outgoing`, and `email-auto-incoming`. All the functionality of the feature is encapsulated in those four functions. The `email-auto-init` function contains code for initializing the data structures needed for the feature; `email-auto-command` provides an interface for mail messages and data structures to be changed, `email-auto-outgoing` provides the logic for handling messages being sent from a user, and `email-auto-incoming` is responsible for handling messages sent to a user.

The four calling functions make up the component for a feature. The ACL2 implementation does not package these up into a formal (code) component because the ACL2 module facility (called a *book*) introduced unnecessary complexity into the verification step for the stage of the project. Using books would have required an additional step in creating modules—certifying each book before it could be used. When rapidly developing the code and theorems, this restriction kept us from using books to split the code into modules. Section 4.1 explains how to use features of ACL2 to simulate the effect of having code encapsulated in modules during verification.

All the other features in the system are similarly put together. The features are linked together, using a minimal of glue code, such that the pieces are able to interact with one another; Section 3.1.3 discusses how this is done.

3.1.3 Compositions of Features

We add features to the base system by modifying the action functions (shown in the base system) to call the appropriate feature functions. This section shows how to add one feature, other features can be added similarly.

```
;;(email-auto-init env) -> env
;; initialize envronment
(defun email-auto-init (env)
   (set-var 'already-answered '()
    (set-var 'user '()
     (set-var 'default-response '() env))))

;;(email-auto-command string list[<anything>] env -> env
(defun email-auto-command (cmd args env)
  (cond ((equal 'SET_USER cmd)
         (let ((?user (car args)))
            (set-var 'user ?user env)))
        ((equal 'SET_DEFAULT_RESPONSE cmd)
         (let ((resp (car args)))
           (set-var 'already-answered '()
           (set-var 'default-response resp env))))
        (t env)))

;;(email-auto-outgoing message env) -> action
;; what transpires when an message is sent
(defun email-auto-outgoing (msg env)
  (begin (cw "  [email-auto-outgoing: ~x0]~%" (get-var 'user env))
         (act comment "[Outgoing events not handled]")))
```

Figure 3.3: ACL2 source code of auto-response feature (1 of 2).

```
;;(email-auto-incoming message env) -> action
;; when a message is received
(defun email-auto-incoming (msg env)
   (if (equal '() (get-var 'user env))
        (act comment "    [User not set yet --> no action]")
      (begin
       (let ((?from (message-sender msg)))
          (if (member-equal ?from (get-var 'already-answered env))
               (act comment "[No autoresponse, already answered -->
                   no further action]")
            (if (equal '() (get-var 'default-response env))
                (act comment "[No default autoresponse -->
                                     no further action]~%")
              (let ((?recip    (recipient msg))
                    (?response (get-var 'default-response env)))
                (begin
                 (cw "  respond '~x0'to ~x1~%" ?response ?recip)
                 (act mail
                      (mk-message ?recip
                       (list (message-sender msg))
                       (set-var 'subject (list 're (subject msg)) '())
                       (cons ?response '()))
                      (set-var
                       'already-answered
                       (cons ?from (get-var 'already-answered env))
                       env))))))))))
```

Figure 3.4: ACL2 source code of auto-response feature (2 of 2).

```
(defun user-init (user)
  (let-seq user
   ;; could add a feature here
   (email-auto-init user)
   ;; ...or here.
           user))
```

Figure 3.5: User initialization source code.

The `user-init` function is used for the initialization action, sequentially calling each feature's initialization function in turn with the user environment as its argument. Each function will return a (possibly) modified environment, which `user-init` passes on as the argument to the next feature. Figure 3.5[1] shows the auto-response's initialization function, `email-auto-init`, being called. To add another feature, a similar line would be added before or after `email-auto-init` in the proper sequence.

The `user-command` function, used for the command action, is similar in nature to the `user-init` function. It sequentially calls each feature's `-command` function, passing on the (possibly) modified environment to input of the next feature. Adding another feature is obtained by adding a line in the desired sequence.

The `do-user-send` function is responsible for tying together the features that affect a user's outgoing message, then passing the message back to the network to send to the specified host. The `user-send-begin` is an alias that points to the

[1] The code listed in the appendix is set up slightly differently, with heavy macro usage, so as to facilitate rapid modification of the features enabled.

```
(defun user-command (act user)
  (let ((cmd  (action-arg1 act))
        (args (action-arg2 act)))
    (let-seq user
             (email-auto-command    cmd args user)
             user)))
```

Figure 3.6: User command action source code.

first feature function. When the encryption feature is enabled, `user-send-begin` is an alias for `user-send-encrypt`. The function `user-send-encrypt` describes the part of the sending state machine that corresponds to the encryption feature. The function calls the encryption feature's `-outgoing` calling function, and then passes the resulting (possibly changed) message and environment to the next feature. Adding a feature requires two steps:

1. Create a `user-send-` function to represent the state transitions of the feature.

2. Define the appropriate aliases to call the features in the desired sequence.

The `do-user-deliver` function is similar to `do-user-send`, but on the receiving host. It passes the message through the features that could affect the message, and then delivers the message. In the case of the auto-response feature, `do-user-deliver` is able to return a send message action, which in turn is handled by the `do-user-send` function. Adding a feature is handled similarly to adding a feature to the sending code.

```
; Define sequence of features to apply
; In this case, using the encrypt feature at the beginning
(fif encrypt
     (defalias user-send-begin user-send-encrypt)
     (defalias user-send-begin user-send-encrypt-next))

; The next ‘‘feature’’ is signifies that the message was sent
;  correctly.
(defalias user-send-encrypt-next        user-send-ok)

(fif encrypt
  (defun user-send-encrypt (msg user)
      (mv-let (status new-msg new-user)
              (email-encrypt-outgoing msg user)
              (cond ((equal 'mail status)
                     (user-send-encrypt-next new-msg new-user))
                    (t (user-send-abort new-msg new-user))))))

(defun user-send-ok (msg user)
   (mv 'deliver   msg (set-var 'sent-msg msg user)))

(defun do-user-send (action users hosts)
  (let ((name (action-name action))
       (msg  (action-arg1 action)))
    (mv-let (status new-msg new-user)
            (user-send-begin msg (get-var name users))
            (let ((new-users (set-var name new-user users)))
               (if (equal status 'send)
                   (mv (mk-host-mail-action 'send new-msg)
                       new-users
                       hosts)
                 (mv nil new-users hosts))))))
```

Figure 3.7: User send action source code.

```
(fif decrypt (defalias user-deliver-begin user-deliver-decrypt)
        (defalias user-deliver-begin user-deliver-decrypt-next))

(fif auto
     (defalias user-deliver-decrypt-next user-deliver-auto)
     (defalias user-deliver-decrypt-next user-deliver-auto-next))

(defalias user-deliver-auto-next deliver-ok)

(defun deliver-ok (msg user)
  (mv ’deliver-ok msg (set-var ’recv-msg msg user)))

(defun do-user-deliver (action users hosts)
  (let* ((name (action-name action))
         (msg  (action-arg1 action)))
    (cond ((get-var name users)
           (mv-let (status new-msg new-user)
                   (user-deliver-begin msg (get-var name users))
                   (let ((new-users (set-var name new-user users)))
                     (if (equal ’mail status)
                          (mv (mk-user-mail-action ’send new-msg)
                              new-users
                              hosts))
                     (mv ’() new-users hosts))))
          (t (mv ’() users hosts)))))
```

Figure 3.8: User deliver action source code.

3.1.4 Interfaces

Components come with interfaces that summarize what information they provide to other components [HC01]. The feature components in this thesis have two kinds of interface information. The calling functions introduced in Section 3.1.2 provide the *construction interface* (what we need to compose components into an existing system). For modular verification, components also need a *verification interface* containing properties that are true of them. Section 3.2.2 illustrates the contents of the verification interface for features in ACL2.

3.2 Verifying the System

In selecting the features to implement and verify in the system, we chose features that were representative of the features included in Hall's system. The *host feature* is the simplest feature, sending a "postmaster response" when an attempt is made to send a message to an unknown user. The *auto-response* feature was chosen to represent features that do not modify the message being sent, but messages with certain properties can trigger other actions to be performed. The *encryption* and *decryption* feature was selected to provide a case where an unknown attribute is present in the system. Hall's study provides several interactions between the host, auto-response and en-/decryption features, which makes them especially suited for including in this study.

For the auto-response feature, the major property to preserve is as follows: The auto-response should reply once and only once to all messages from active users of the system, but not including automated messages such as the postmaster. The property is shown as a combination of f-auto-adds-already-answered-and-thm and f-auto-auto-response-if-not-already-answered-thm in Figure 3.9. The first theorem proves that sending a message to a user with an auto-response mechanism enabled adds the sender to the 'already-answered list in the user's environment. Lines 2–9 make up the body of the theorem. The implies is the ACL2 implication function, identical to its mathematical meaning. Lines 3–6 comprise the *assumptions* that are taken to be true to the *action* (lines 7-9). The assumptions, in this case, are: 1) the auto-response feature is enabled ('default-response is non-nil); 2) the environment is a valid user environment ('user is non-nil); and 3) the sender of the message msg is equal to sender. The action (implication) part of the theorem states thusly: the resultant environment from delivering (via user-deliver) the message msg will contain sender in the 'default-response variable. The second theorem proves that sending a subsequent message to a user, when the sender is already in the 'already-answered list, results in no additional auto-response messages. The two theorems are sufficient to prove the above property.

For the host feature, Figure 3.10 shows f-host-postmaster-response-to-unknown-user-thm, the property that must be preserved. An informal definition of this property is as follows: If a message received by the host is not sent to a user of that host, then a postmaster 'user unknown' message is sent in reply.

When reporting verification results, we do not include standard performance measures such as runtime or memory usage. These measures are not as meaningful

```
; Line numbers added for illustrative purposes only
(defthm f-auto-adds-already-answered-thm
  (implies
    (and
      (get-var 'default-response env)
      (get-var 'user             env)
      (equal (message-sender msg) sender))
   (user-in-already-answered
     sender
     (mv-env (user-deliver msg env)))))

(defthm f-auto-auto-response-if-not-already-answered-thm
  (implies
    (and
      (get-var 'default-response env)
      (get-var 'user             env)
      (equal sender (message-sender msg))
      (not (member-equal sender (get-var 'already-answered env))))
   (mv-let (status new-msg new-env)
       (user-deliver msg env)
       (equal status 'mail))))
```

Figure 3.9: ACL2 preservation properties for the auto-response feature.

```
(defthm f-host-postmaster-response-to-unknown-user-thm
  (implies
   (and
    (equal recipient (recipient msg))
    (equal (email-host recipient) (get-var ’hostname env))
    (equal (email-user recipient) username)
    (not (member-equal username (get-var ’users env)))
    (message-p msg))
   (mv-let (status new-msg new-env)
   (host-send msg env)
   (and
    (equal status                                ’mail)
    (equal (email-user (message-sender new-msg)) ’postmaster)
    (equal (recipient new-msg)                   sender)))))
```

Figure 3.10: ACL2 preservation properties for the host feature.

in theorem proving, where the main focus is on user overhead to develop the proofs. Given the exploratory nature of the work presented here, figures on the amount of user time would be premature.

3.2.1 Verifying the Whole Composed System

When verifying a system modularly, it is important that the results be identical to those that would have resulted from verifying the entire composed system. The first step in this case study, therefore was to verify the given properties on a system comprised of the all three features previously listed. This provides the baseline for the intended results of the modular verification. All of the theorems listed above proved as expected in the entire system.

3.2.2 Verifying Individual Features

When verifying an individual feature, we start with the four calling functions, and the property that we want to verify. The four calling functions do not stand entirely on their own—they need the infrastructure of the base system to tie them together to the network system. Therefore, when verifying an individual system, we also include the base system. This follows the same technique in Section 3.1.2.

Verifying a property proceeds in several stages. To start, we verify the interface of the feature, that is, how it affects the message being sent/received or the environment. These basic theorems are then used to prove more complex theorems on the code that performs each individual action. Finally, these serve as lemmas to lift the property to the `simulaste-network` code. Figure 3.11 shows the hierarchy of theorems that correspond to those needed to prove the auto-responser property of adding the message sender to the `'already-answered` list. As the proves moves up the levels, the function on which the theorem is being proven changes. The assumptions stay the same, or tighten slightly, depending on what is relevant for the current level. Figure 3.12 depicts the levels of theorems necessary to produce a final theorem on the `simulate-network` level. Theorems proven on the level immediately below are used to prove theorems about the current level.

3.2.3 Verifying Non-interference Between Features

As an example an interaction between two features, consider the auto-response and host features. The first feature provides for an automated response to incoming

```
(defthm email-auto-incoming/auto-response-adds-sender-to-already-answered
  (implies
   (and (get-var 'default-response env)
        (get-var 'user env)
        (equal sender (message-sender msg))
        (not (member-equal sender (get-var 'already-answered env))))
   (user-in-already-answered sender
     (mv-env (email-auto-incoming msg env)))))

(defthm user-deliver/auto-response-adds-sender-to-already-answered
 (implies
  (and (get-var 'default-response recipient-env)
      (get-var 'user             recipient-env)
      (equal (message-sender msg) sender)
      (not (member-equal sender (get-var 'already-answered recipient-env))))
  (user-in-already-answered sender
   (mv-env (user-deliver msg recipient-env)))))

(defthm simulate-network/auto-response-adds-sender-to-already-answered
 (implies
  (and (get-var 'default-response recipient-env)
     (get-var 'user             recipient-env)
     (equal (message-sender msg) sender)
     (equal (email-user sender)  sender-name)
     (not (member-equal sender (get-var 'already-answered recipient-env)))
     (equal (email-user (recip msg)) recip-name)
     (equal (get-var user users) recipient-env))
  (user-in-already-answered sender
   (get-var recip-name
     (mv-nth 1 (simulate-network
                 (mk-action 'mail sender-name msg)
                 users hosts))))))
```

Figure 3.11: Hierarchy of theorems to prove auto-response property.

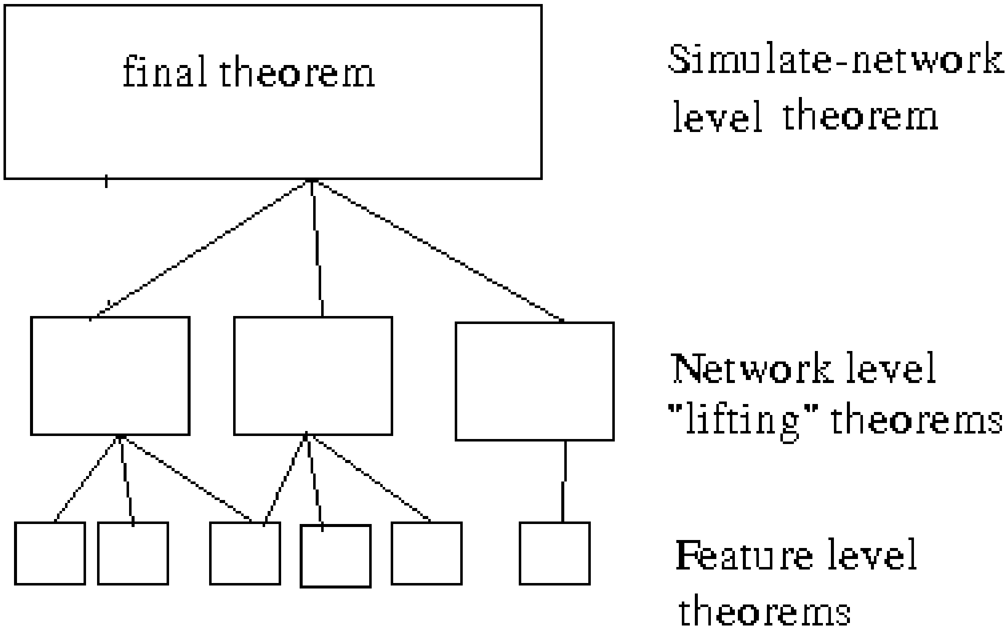
final theorem
Simulate-network level theorem
Network level "lifting" theorems
Feature level theorems

Figure 3.12: Levels of theorems.

messages. The second sends an undeliverable response from the `postmaster` when a message is sent to an unknown user on the server. The `postmaster` account is an unmonitored system account whose messages are often discarded.

One interaction that Hall has found is between these two features. As outlined in Section 3.2, when a message is sent to an invalid user on a host, it is customary for the `postmaster` of the host to reply with a 'user unknown' response. The `postmaster` account is an administrative address, and is not always monitored by a human. It would therefore be extraneous to send an auto-response message to the `postmaster`. While this situation does not indicate that something went wrong

1. `A` sends a message to `ZZZ@host` (an unknown user)
2. The mailing software of host determines that `ZZZ` is an unknown user.
3. The `postmaster` at `host` sends an 'user unknown' message to `A`.

Figure 3.13: Normal scenario of `Postmaster` sending a 'user unknown' message.

0. `A` sets his auto-response to reply to any incoming email.
1. *same as above in 3.13.*
2. *same as above in 3.13.*
3. *same as above in 3.13.*
4. The auto-response of `A` sends a reply to `Postmaster@host`. This reply is extraneous and is unexpected by the user.

Figure 3.14: Interaction scenario between auto-response and host features.

in the email system, it is considered an unwanted interaction within the feature-interaction community.

The scenarios in Figures 3.13 and 3.14 illustrate both the expected and unexpected behaviors of the auto-response and host features. The unexpected behavior is where the interaction occurs.

We can verify that this interaction is possible by describing the interaction as an ACL2 theorem. Intuitively, the premise of the theorem, is: given that the vacation message can only be a valid vacation message, and that the user name of the sender is a valid user name; sending a message to user who does not exist on a host should result in the postmaster receiving a vacation message from the original sender of the message. The formal theorem statement is listed in Figure 3.15.

Capturing the interaction as a theorem, rather that the absence of the interaction is less than ideal. It presupposes that the interaction was already identified in the system. The goal of this thesis was to recreate the problems that Hall identified in a modular fashion; this helps determine sufficient interface information to identify these problems. Theorem provers have known limitations in identifying the sources of failure in proofs. Working with the actual interactions made it easier to detect what interface information was points to the interactions. Carrying this over to work with the theorems that state that no interaction exists is deferred to future work.

Recall that the goal of doing this compositionally is to prove the theorems without referring the to contents of the auto-response and host features. Instead, we will prove these theorems using interface information about these two features.

What information is needed from the two features in order to prove that the interaction occurs? From the auto-response, it is sufficient to know that: 1) if the auto-response feature is enabled, and the sending user has not yet been responded to (i.e. not in `'already-responded`), then that user will be added to the `'already-responded` list, and 2) if the auto-response feature is enabled, and the sending user has not yet been responded to, then an auto-response reply message will be sent. The first captures the changes in the environment, the second capture the action that results. For the host system, it is sufficient to know that if the user

```
(defthm vacationer-autoresponds-to-postmaster-thm
  (implies
   (and
    (message-p msg)
    (equal (recipient msg) recipient)
    (equal (email-user (message-sender msg)) sender-name)
    (equal (email-user recipient) recipient-name)
    (equal (email-host recipient) recipient-host)
    (symbolp vacation)
    (equal (get-var sender-name users) sender-env)
    (get-var 'default-response sender-env)
    (get-var recipient-host hosts)
    (not (get-var recipient-name users))
)
  (mv-let (status new-users new-hosts)
       (simulate-network (mk-action 'mail sender-name msg)
                         users hosts)
    (eq (body-lines (get-user-var 'postmaster 'recv-msg users))
 vacation))))
```

Figure 3.15: ACL2 theorem for extraneous auto-response messages.

in the recipient field in not known on that host, then the `postmaster` will send an 'unknown user' reply to the message sender.

All of these constraints are easily captured in theorems that appear in Figures 3.9 and 3.10 . The first two theorems prove that the auto-response feature performs as expected, sending automated "vacation" responses to users once and only once. This requires the code of the auto-response feature, as well as the base system code. The last theorem proves that the host feature operates in the correct manner for sending 'unknown user' replies. The code of the host feature as well as the base system are required here. Together, these theorems capture the necessary interfaces for proving the larger auto-response interaction property in Figure 3.16. However, additional work must be done in order for the proofs to be accepted into the theorem proving environment.

```
(defthm simulate-network/auto-response-to-postmaster
 (implies
  (and
   (equal recip-name ’postmaster)
   (equal (get-var user users) recipient-env)
   (get-var ’default-response recipient-env)
   (get-var ’user             recipient-env)
   (not (member-equal recip-name
                      (get-var ’already-answered recipient-env)))
   (equal (email-user (message-sender msg)) sender-name)
   (equal (email-user (recipient msg))      recip-name)
   )
  (user-sent-auto-response-to
   sender-name
   recip-name
   (mv-nth 1 (simulate-network
                (mk-action ’mail sender-name msg)
                users
                hosts)))))
```

Figure 3.16: ACL2 theorem for ensuring an auto-response message will be sent.

In order to prove the interaction exists in the whole system, we must “lift” the properties of the features to a level where the features are able to interact. In the code for the whole system, the features interact by both being called starting from `simulate-network`. We therefore need to lift the interface properties proven about the features to theorems that refer to `simulate-network`. Figure 3.12 shows the hierarchy of theorem levels needed to prove a theorem, while Figure 3.11 shows those theorems for the the auto-response feature.

Treating the lifted theorems as lemmas, we can now prove that the interaction exists on the entire system. However, in doing so, we would be using the code from the entire system, which would be counter-productive to the modular verification goals of the thesis. The code for connecting the features are needed for the final proofs and cannot be removed from the system. However, the code contained within the individual features are not needed and **should not be used**, as the already-proven interfaces provide sufficient information for the verification. The theorem prover of ACL2 operates by expanding function definitions as needed to prove the theorems. Left to its core algorithms, ACL2 proves the interaction theorem, but looks at the contents of the features. If ACL2 did not look at the contents of the features, the resulting proof would be modular.

Fortunately, ACL2 provides a mechanism for preventing it from expanding certain functions when proving theorems. Using the ACL2 feature to "disable" expanding (or re-writing) the feature contents, leaves the connections between the feature intact, while making the proof modular. Figure 3.17 contains a small example illustrating how disabling a function affects its proof.

The function `f` returns `'a` if its argument is true, and `'b` otherwise. We wish to prove that `f` returns either `'a` or `'b` for all inputs. We first prove, via `f-of-true-returns-a` that `f` returns `'a` for all true inputs. Next, `f-of-false-returns-b` proves that `f` returns `'b` for all non-true inputs. Since these two facts will be

```
(defun f (x) (if x ’a ’b))

(defthm f-of-true-returns-a
  (implies
     x
     (equal (f x) ’a)))

(defthm f-of-false-returns-b
  (implies
     (not x)
     (equal (f x) ’b)))

; Disable expanding definition of f
(in-theory (disable f))

(defthm f-returns-a-or-b
  (member-equal (f x) ’(a b))
  :hints ((‘‘Goal’’ :use (f-of-true-returns-a
                          f-of-false-returns-b))))
```

Figure 3.17: ACL2 example of using `disable`.

sufficient for proving our property, we then `disable` the expansion of function `f` in all further proofs. That is, the definition of `f` is no longer consulted when attempting to prove the theorems. In our final theorem, `f-returns-a-or-b`, we use the results of the previous two theorems, to prove the original property.

We used the disabling technique when proving all the interaction theorems. Using this technique, we were able to duplicate the results obtained from verifying the whole complete system in Section 3.2.1. Using the disabling technique, we were able to duplicate the results obtained from verifying the whole complete system in

Section 3.2.1. This process and style of interface theorems are representative of the steps and theorems needed to prove the other interactions.

3.3 Summary of Case Study

By building the system up from the base system, we were able to clearly observe the effects of adding a feature on the system. Each feature added was a collection of functions which implemented the necessary functional interfaces. Adding a feature to the system was simple, thanks to the ample usage of macros in the main source code. Only minimal changes must be made to add a feature to the base system, this allowed me to experiment with a variety of features in rapid succession. Each feature was individually added to the base system, and verified to ensure that its major properties were preserved.

To ensure that the interactions occurred, an attempt was made to prove all the properties on the whole composed system; the system with all features enabled. The baseline proven by this attempt gave validation to the next section of the case study—proving the interactions occurred, modularly. By "disabling" the theorem prover from expanding the function definitions of the features, we could ensure that all theorems were proven on the basis of the previously proven verification interfaces. In other words, by obtaining the same proof results both with and without expanding the feature code, we modularly verified that the properties listed in Section 3.2.

Chapter 4

The Methodology

This chapter describes a methodology for creating and proving theorems about feature-oriented systems in the ACL2 theorem proving environment. In order to determine what theorems must be proven, it is paramount that 1) the base system and calling function for each feature have already been implemented, and 2) the set of properties to preserve and interactions to confirm are known. The methods presented here illustrate how to optimally set up the model in ACL2, and how to design different types of interface theorems. Additionally, hints are given on which parts of the model are needed for proving different types of theorems, and how effectively manage the proof process.

4.1 Setting up the Model

In considering flexibility, both the model and the implementation should be designed to easily allow new features to be added, with minimal or no changes needed. The model of the system was designed to be highly modular; the implementation of the system is modular as well, as each feature is contained in a separate file.

In implementing the system, we had two main goals:

1. To make it easy to enable or disable features so we could study the impact of the features on properties in the system.

2. To make adding a new feature (to the code, as well as the corresponding theorems) as simple as possible. Many of the theorems have a similar structure across features (differing only in the name of the calling function or an environment variable). Having a way to share this common structure would simplify both adding new features and altering the common structure as needed once we began using them for verification.

ACL2 macro facilities enabled me to meet both of these goals [Gra93].

For the first goal, we created a macro, `fif` (short for "feature-if"), which conditionally includes the code for a feature, as well as the necessary code in the network system, based on whether the feature is enabled. A feature is enabled by setting an ACL2 variable with the names of the features to be used. Figure 4.1 shows the same code as Figure 3.5, but using the `fif` macro to increase flexibility. The `*features-present*` line in the figure specifies that only the auto-response and encryption code should be included when macro expanded.

```
(defconst *features-present* '(auto encrypt))

(defund user-init (user)
  (let-seq user
   (fif encrypt (email-encrypt-init          user) user)
   (fif decrypt (email-decrypt-init          user) user)
   (fif auto    (email-auto-init             user) user)
           user))
```

Figure 4.1: User initialization source code, using the `fif` macro.

The second goal was similarly solved with macros. We created various macros to create theorems parameterized by details from the actual features. For example, Figure 4.2 shows the macro `make-*/mail-returns-message-p-thm` which creates a theorem to prove that the function's second return value is has the format of a valid message. The macro takes in one or more arguments: 1) the name of the function to which to apply the theorem, and (optionally) 2) any hints that should be provided to the theorem prover to aid in proving the theorem[1].

Obviously these macros do not affect the ability to prove theorems in the system. However, these macros are critical for reducing user overhead in proving the theorems. Without the simplifying macros, it would be difficult for the programmer to maintain and update the system, as well as keep the similar theorems consistent across features.

[1] Hints are a feature of ACL2 which provide the theorem prover with a suggest of how to best proceed in the proof.

```
(defmacro make-*/mail-returns-message-p-thm (func &optional hints)
  ‘(defthm ,(new-thm func ’returns-message-p)
     (implies
       (message-p msg)
      (message-p (mv-nth 1 (,func msg env))))
     :hints ,(inject-goals
        (list
          (list "Goal" ’:in-theory (list ’enable func))) hints)))

(make-*/mail-returns-message-p-thm     email-auto-outgoing)
```

Figure 4.2: Macro definition for `make-*/mail-returns-message-p-thm` and example usage.

4.2 Designing Interface Theorems

Theorems characterize the following:

1. *Input/output:* Constraints on the types or format of the functional inputs (arguments) and outputs (return values).

2. *Changes:* Which attributes *might* and definitely do not change in the environment after executing the feature. Figure 4.4 shows the theorem `thm-email-auto-init-adds-or-changes-only-x-variables`, which proves that only `’already-answered`, `’user` and `’default-response` are added by the function `email-auto-init`. Explicitly stating those variables that *are* changed enables the theorem to remain the same when additional variables are added

```
; Provided that the first argument to email-auto-outgoing
;  satisfies message-p, the second return value of
;  calling the function will always satisfy message-p.
(defthm thm-email-auto-outgoing-returns-message-p
    (implies (message-p msg)
             (message-p (mv-nth 1 (email-auto-outgoing msg env)))))
```

Figure 4.3: Constraints on functional outputs.

to the system—the number and content of the unknown variables in the system are irrelevant to this theorem.

3. *Dependent changes:* How the environment is changed and what values are returned, based on attributes of the inputs or variables in the environment. Figure 4.5 shows an example. The theorem `auto-response-if-not-already-answered-mail-action` proves that if the auto-response mechanism is enabled and the sender of the incoming message is not already in the `'already-answered` list, then an auto-response message is sent back. The theorem `no-auto-response- if-already-answered` proves the reverse—that if the sender is already in the `'already-answered` list, then no auto-response message is sent.

Some of the theorem types listed above have a great potential for being generated automatically. Writing theorems is a time-consuming task, so the automation both reduces the time spent, as well as the chances of the theorem being improperly

```
(defthm thm-email-auto-init-adds-or-changes-only-x-variables
 (implies
   (and (symbol-alistp env)
        (equal (get-var key env) var)
        (not (member key
                     '(already-answered user default-response))))
   (let
    ((new-env (email-auto-init env)))
    (and (equal (get-var key new-env) var)
        (and (has-var 'already-answered new-env)
             (has-var 'user             new-env)
             (has-var 'default-response new-env))))))
```

Figure 4.4: Which attributes are changed during function call.

stated. The list below characterizes the above list as to each class of theorem's potential for automatic generation.

1. *Input/output:* With minimal programmer input, these theorems could be automated. Programmers of strongly-typed languages are already accustomed to specifying the type of their variables. With simple type annotations on variables, the input and output types of functions could be automatically derived (generated) and proven.

2. *Changes:* Which simple code-analysis techniques, these theorems could also be largely automatically generated. A simple parsing of the possible paths in a function could determine which variables are always changed and which are only sometimes changed. Variables that are changed based on an input

```
;; auto-response-if-not-already-answered-mail-action
;   If enabled, first message from user results in an
;;    auto-response message
(defthm auto-response-if-not-already-answered-mail-action
  (implies
   (and
    (get-var ’default-response env)   ; autoresponder enabled
    (get-var ’user             env)
    (equal sender (message-sender msg))
    (not (member-equal sender (get-var ’already-answered env)))
    )
   (equal (mv-status (email-auto-incoming msg env)
          ’mail))))

;; no-auto-response-if-already-answered
;;   If enabled, subsequent messages from the same user result in
;;   no additional messages
(defthm no-auto-response-if-already-answered
  (implies
   (and
    (equal (message-sender msg) sender)
    (user-in-already-answered sender env)
    )
   (not
    (equal (mv-status (email-auto-incoming msg env))
   ’mail))))
```

Figure 4.5: Which attributes are changed during function call.

or environmental variable could be marked, and put into a theorem using the *dependent changes* method.

3. *Dependent changes:* The least likely candidate for automation. These theorems are highly dependent on the code contained in the functions used in the theorem.

4. *"Lifters":* These theorems that "lift" previously proven theorems to the next level of code also have potential for semi-automatic generation. The first theorem on a function that is lifted to the next level require human intervention to discover the exact methods used for related one level to the next. However, the subsequent theorems use largely the same techniques for the lifting, and could possibly be generated automatically.

Although using macros or generating theorems automatically can reduce the amount of effort necessary to initially produce a theorem, some theorems may require more user intervention to prove. In our experience, the *dependent changes* and *lifter* theorems required the most user intervention, often requiring hints to the theorem-prover or additional helper lemmas before finally proving. The *input/output* and *changes* theorems in the system required almost no changes.

4.3 Model Elements for Proving Theorems

Each class of theorem must be checked against different parts of the system model. For example, the *property* theorems need only be checked against the feature-level code, while the *interaction* theorems use only the top-most level. The following is a list explaining which parts of the model each theorem uses.

1. *Property, Input/Output, Changes, Dependent changes:* These classes of theorems use the feature-level code in the system. These require the feature and base system code, as illustrated in Section 3.2.2.

2. *Interaction:* Since interactions primarily occur between features, this class of theorems uses only the top-most level of code for its proofs. The feature-level code should be disabled during these proofs, and cannot be used to provide details for the proof. See Section 3.2.3 for a complete explanation.

3. *Lifting:* Acting as a bridge between the *interaction* and other types of theorems, this class could potentially use all the levels of code in the system. For this particular setup, we had three main levels of code: the feature- (calling functions), action- (`user-command`, `user-init` functions, etc), and top-level (the `simulate-network` function). The total number of levels could be different for other system models, but the nature of lifting theorems would remain the same.

4.4 Managing the Proof Process

One of the downsides of theorem-proving is the level of user interaction required to adequately prove a system. Ideally, the methodology for proving the system should minimize the amount of user time and effort needed.

With ACL2, you can automate the proof process in two main ways: 1) auxiliary proofs capture common lemmas about key relationships in the system; and 2) theorem-prover hints that tell ACL2 which lemmas to use at particular points in a proof. The lifting theorems are the main form of auxiliary theorems we used, which provided the necessary lemmas for the later proof of the interaction theorems. Hints provided a method in proving the lifting theorems for specifying which variables on one level corresponded with the variables on the next level of code. For the interaction theorems, hints provided a way to point ACL2 to the proper lemmas to use in the proof process. Regardless of the system implementation, hints will still be necessary to ensure that ACL2 uses the proper theorems. This part requires user intervention, and is unlikely to be automated.

4.5 Perspective on Methodology

4.5.1 Strengths

1. Easier handling of unknown values. By using a theorem proving environment, unknown values became a minor issue, in contrast to the major difficulty they presented by using a model checker.

2. Ability to reason about a list of actions, rather than just one action at a time. Li's model checker system used only one action at a time, which limits its ability to express rich theorems.

4.5.2 Limitations

1. The current methodology is not able to detect actual interactions, but instead had to rely on those already identified.

2. Theorems must be stated in the positive form. That is, that the interaction *does* exist. It is difficult in the theorem proving environment to prove a property does *not* hold; the failure of a theorem to prove could also signify that it was improperly stated, or does not have enough supporting proofs.

3. The user overhead of creating the theorems is high, but could be reduced significantly through the use of automating techniques. See Sections 5.2 and 4.2 for ideas on automating theorem generation.

Chapter 5

Conclusions

The goal of this research has been to determine what kinds of system model and interface theorems would allow us to reproduce Hall's interactions modularly. These in turn yield a proposed methodology for handling other, similar, systems. The primary focus of this work was not on finding the interactions, but rather finding theorems that proved them to exist: the most challenging part of this project was in coming up with the change and dependent change theorems, which characterize the information that one feature needs about another during modular verification. This characterization, and the overall system architecture, are the main and novel contributions of this work.

To conclude this thesis, we review the results of the case study, summarize the categories of theorems needed for the proofs, review the features of ACL2 that helped in verifying the system, and cite directions for future work.

5.1 Review of Case Study

Starting from a base system devoid of features, we individually modeled and implemented three additional features: the *host*, *auto-response*, and *en-/decrytion* features. The features were chosen as a representative sample of the features presented in Hall's paper. These features and their implementation is discussed in Section 3.1.2.

We combined two or more features together to create instances where features interacted in an interesting manner. Section 3.2.3 showed how to prove that interaction occurred between features. We were able to prove that the *auto-response* feature interacted with the *host* feature when sending spurious response messages to the host's postmaster messages. Additionally, we started the proof to show that the *en-/decryption* feature interacted with the *auto-response* feature by sending out previously encrypted text over the network in clear-text. Although we were unable to finish these proofs in the time available, problems appeared to be in minor assumptions and not in the methodology. Based on the initial successes, we are confident that all the feature interaction presented in Hall's paper could be similarly proven.

Section 4.2 categorized the types of theorems in several distinct categories. To show how common each of the types are, Figure 5.1 shows the number of theorems that were needed in proving the interactions. The feature-level theorems were generally easy to produce, and required less user time and effort to create and prove. The majority of the effort was expended at the lifting theorems, particularly because of the number of theorems at that level (as shown in Table 5.1).

Theorem Type/Level	**Number Proven**
Top Level	
- `simulate-network`	1
Lifting Level	
- lifting theorems	88
Feature Level	
- input/output	21
- changes	12
- dependent changes	8

Figure 5.1: Distribution of theorems proven.

5.1.1 ACL2 Features

The language chosen, ACL2, had several features that were particularly helpful the process of modeling the system, and proving the properties and interaction. This list highlights those features that were most helpful:

1. Disabling - The ability to disable functions from being “expanded” in the proof was crucial to successfully proving the theorems in a modular fashion. With out this feature, we would have be un-able to conclusively state that the theorems were proven without using the source code of the features.

2. Books - This feature of ACL2, although it proved to unwieldy to implement for our purposes, highlighted the need to separate out the modules into individual files.

Initially, the further up the theorems were “lifted,” [1] the harder the theorems were to prove. However, as our knowledge of working with ACL2 and proving

[1] See Section 3.2.2 for details on lifting theorems.

theorems increased, the opposite became true—finding the initial theorems for the feature interfaces became problematic, but once found, were easily lifted up to the network-level code verification. Given this learning overhead, it is difficult to estimate the amount of user time needed to use this methodology once a user becomes experienced with ACL2 and feature verification.

5.2 Future Work

Section 4.5 summarizes the strengths and limitations of this work. Three issues which could provide the most immediate benefit are:

1. Automatically generate theorems which pertain to changes that could occur in the environment. A program could analyze the feature source code, and determine which environment variables will definitely change, and which could possible change in the calling functions. These analyses could then easily be turned into theorems on the feature (see Section 4.2 for details).

2. Finish reproducing the interactions presented in Hall's study.

3. Perform another case study to determine what other kinds of interfaces can be or should be determined on the feature code. This would also have the benefit of validating the methodology proposed in Section 4.

Appendix A

Code Listing of System

A.1 mail.lisp

```
(include-book "/usr/share/acl2-2.6/books/data-structures/list-theory")
(include-book "/usr/share/acl2-2.6/books/data-structures/alist-theory")
(include-book "/usr/share/acl2-2.6/books/data-structures/set-theory")
(include-book "/usr/share/acl2-2.6/books/data-structures/structures")

; Enable all features implemented responder
(defconst *features-present* '(encrypt decrypt auto host))

(ld "const.lisp")
(ld "util.lisp")

(fif host    (ld "f-host.lisp"))
(fif encrypt (ld "f-encrypt.lisp"))
(fif decrypt (ld "f-decrypt.lisp"))
(fif auto    (ld "f-auto.lisp"))

(ld "base.lisp")
(ld "theorems.lisp"))
```

A.2 const.lisp

```
;;; CONSTANTS
;;; Used mainly for testing purposes.

;;*users*
;; Describes users are in the system
(defconst *users*
  '(
    (rjh (user . rjh)
 (already-answered)
 (default-response)
 (name . rjh))
    (bob (user . bob)
 (already-answered)
 (user)
 (default-response)
 (name . bob))
    (postmaster (already-answered)
(default-response)
(user . postmaster)
(name . postmaster))))

;;*hosts*
;; Describes hosts in system
(defconst *hosts*
  '(
    (host (users rjh bob postmaster)
  (users bob postmaster)
  (hostname . host)
  (hostname)
  (users postmaster)
  (name . host))
    (remail (hostname)
    (users postmaster)
    (name . remail))
    (sender (hostname)
    (users postmaster)
    (name . sender))))

(defconst *commands*
  '(init_user
    set_hostname
    set_user
    default_response))

(defconst *key-space* '(a b c d e))
(defconst *user-names* '(rjh bob))
(defconst *vacation-space* '("I'm on vacation"
    "I love my job, but i love fishing more"))

(defconst *init-env*
  (acons 'users *users*
  (acons 'hosts *hosts*
   '())))

(defconst *msg*
  (mk-message
   (mk-email 'bob 'host)
   (list (mk-email 'rjh 'host))
   '()
   (list "none")))
```

A.3 util.lisp

```
;;; MACROS
;;;

; To see what these macros expand out to, use:
;  (macroexpand '(<macro> <macro-args>) '() state)
; or
;  (expand (<macro> <args>))

;; (expand term)
;; expands macro to one level
(defmacro expand (macro)
  `(macroexpand1 (quote ,macro) '() state))

;;(begin list[code]) -> list[len(list)]
;;   returns results of last expression
(defmacro begin (exp &rest rst)
  (cond ((endp rst) exp)
        ((endp (cdr rst)) `(prog2$ ,exp ,(car rst)))
        (t `(prog2$
             ,exp
             (begin ,(car rst) ,@(cdr rst))))))

;;(lookup string [string]) -> value
;; macro to lookup variable in current env
(defmacro lookup (str1 &optional str2)
  `(if (not (null ,str2))
       (get-hash-var ,str2 ,str1 env)
     (get-var ,str1 env)))

;;(lookup string [string]) -> value
;; macro to lookup variable in current env
(defmacro lookupx (str &optional str2)
  (let ((s1 (string str))
        (s2 (string str2)))
    `(if (not (null ,s2)) (get-hash-var ,s2 ,s1 env)
       (get-var ,s1 env))))


;;(let-seq var list[code]) -> list[len(list)]
;; Sequentially binds var to results of list[i] code
;;  returns results of last expression
(defmacro let-seq (var &rest rst)
  (cond ((endp rst) (car rst))
        ((endp (cdr rst)) (car rst))
        (t `(let ((,var (,@(car rst))))
              (let-seq ,var ,@(cdr rst))))))
```

```
; mv-let-seq var list[code] -> list[len(list)]
; sequentially binds var to results of list[i] code
;  returns results of last expression
(defmacro mv-let-seq (vars &rest rst)
  (cond ((endp rst) (car rst))
        ((endp (cdr rst)) (car rst))
        (t ‘(mv-let ,vars (,@(car rst))
              (mv-let-seq ,vars ,@(cdr rst))))))

; mv-let-seq var list[code] -> list[len(list)]
; sequentially binds var to results of list[i] code
;  returns results of last expression
(defmacro mv-let-seq-use (vars &rest rst)
  (cond ((endp rst)       ‘())
        ((endp (cdr rst)) ‘(begin ,@vars (,@(car rst))))
        (t ‘(mv-let ,vars  (begin ,@vars (,@(car rst)))
              (mv-let-seq-use ,vars ,@(cdr rst))))))



;;(defalias f1 f2) -> (defmacro f1 ...)
;;defines alias for function
;;Macro-defining macro from "On Lisp" by Paul Graham
(defmacro defalias (f1 f2)
  ‘(defmacro ,f1 (&rest args)
     ‘(,’,f2 ,@args)))

;;(fif feature if_stmt else_stmt) -> statement
;;if feature is enabled, use if_stmt, otherwise else_stmt
(defmacro fif (feature if_stmt &optional else_stmt)
  (cond ((member-equal feature *features-present*)
          ‘,if_stmt)
        (t ‘,else_stmt)))

;;(fif-all features if_stmt else_stmt) -> statement
;;if features are enabled, use if_stmt, otherwise else_stmt
(defmacro fif-all (features if_stmt &optional else_stmt)
  (cond ((null features) ‘,if_stmt)
((member-equal (car features) *features-present*)
 (let ((f2 (cdr features)))
   `(fif-all ,f2 ,if_stmt ,else_stmt)))
        (t `,else_stmt)))

;;(fif-any features if_stmt else_stmt) -> statement
;;if any feature is enabled, use if_stmt, otherwise else_stmt
(defmacro fif-any (features if_stmt &optional else_stmt)
  (cond ((null features) (begin (cw "(t: ~x0)~%" if_stmt) `,else_stmt))
((member-equal (car features) *features-present*)
 `,if_stmt)
        (t
 (let ((f2 (cdr features)))
       `(fif-any ,f2 ,if_stmt ,else_stmt)))))

;;; ENV FUNCTIONS
(defun envp (env)
 (declare (xargs :guard t))
  (symbol-alistp env))

;;(set-var symbol value env) -> env
;; Returns a new list with 'var changed to 'val
(defun set-var (var val env)
;  (declare (xargs :guard (and (symbolp var)
;       (symbol-alistp env)
;   )))
  (acons var val env))

;;(get-var symbol env) -> value
;; Return val of 'var in list
(defun get-var (var env)
  (declare (xargs :guard (and (symbolp var)
      (symbol-alistp env))))
  (cdr (assoc var env)))

;;(has-var symbol env)-> boolean
;; checks whether symbol is defined in alop
;(defalias has-var assoc)
```

```
(defun has-var (var env)
  (declare (xargs :guard (and (symbolp var)
      (symbol-alistp env))))
  (assoc var env))

(defun symbol2-alist2p (var1 var2 alist)
  (declare (xargs :guard t)) ;(and (symbolp var1)
     ; (symbolp var2)
     ; (symbol-alistp alist))))
  (and (symbol-alistp alist)
       (symbolp var1)
       (let ((alist2 (get-var var1 alist)))
 (and
  (symbolp var2)
  (symbol-alistp alist2)))))

;;(set-hash-var symbol value symbol env) -> env
;; sets var to val in hash contained in env
(defun set-hash-var (var val hash env)
;  (declare (xargs :guard (and (symbolp var)
;       (symbolp hash)
;       (symbol-alistp env)
;       (symbol2-alist2p hash var env)
;       )))
  (set-var hash                               ; Set hash var
           (set-var var val                   ;  after setting var->val
                    (get-var hash env)) env);  after getting hash
  )

;;(get-hash-var symbol symbol env) -> value
;; get value of var in hash contained in env
(defun get-hash-var (var hash env)
  (declare (xargs :guard (and (symbolp var)
      (symbol-alistp env)
      (symbol2-alist2p hash var env))))
  (get-var var                ; Set hash var
           (get-var hash env))) ;  after getting hash
```

```
(defun host-envp (env)
 (declare (xargs :guard t))
 (and (envp env)
      (atom-listp (lookup 'users))))

(defun user-envp (env)
 (declare (xargs :guard t))
 (and (envp env)
      (listp (lookup 'already-answered))))


(defun env-subset (?sub ?super)
  (declare (xargs :guard (and
  (symbol-alistp ?super)
  (symbol-alistp ?sub))))
   (subsetp-equal (domain ?sub) (domain ?super)))

;;(comment string env) -> env
;; prints out a comments, returns env
(defun comment (str env)
  (declare (xargs :guard (and (stringp str)
      (symbol-alistp env))))
  (or
   (cw str)   ; returns '()
   (cw "~%")  ; newline
   env))

;;(actionp message) -> boolean
; predicate for checking whether variable is an action
(defun actionp (act)
  (declare (xargs :guard t))
  (and (symbol-alistp   act)
       (get-var 'command act)
       (has-var 'args    act)
       (has-var 'env     act)))


;;; HELPER FUNCTIONS
```

```
;;;

;;(stringify-helper <anything> boolean -> string
;; helper function to stringify
(defun stringify-helper (list in-list?)
  (declare (xargs :guard t))
  (cond ((null list) ")")
((stringp list)    (string list))
((characterp list) (string list))
((symbolp list)    (string list))
((consp   list)
 (string-append
  (if in-list? " " "(")
  (string-append
   (stringify-helper (car list) nil)
   (string-append " "
  (stringify-helper (cdr list) t)))))
(t "<error>")))


;;(stringify ?) -> string
;; stringifies list or variable
(defun stringify (list)
  (declare (xargs :guard t))
  (stringify-helper list nil))

;;(los-adjoin item list) -> list
;; Add item to list if item is not already part of list
(defun los-adjoin (item list)
  (declare (xargs :guard (and (atom item)
      (atom-listp list))))
  (if (member-equal item list)
      list
    (cons item list)))

(defstructure email
  (user (:assert (and (symbolp user)
      (not (null user)))))
  (host (:assert (and (symbolp host)
      (not (null host)))))
  (:options :guards))

(defun email-listp (lst)
  (declare (xargs :guard t))
  (if (ATOM LST)
      (EQ LST NIL)
    (AND (email-P (CAR LST))
 (email-LISTP (CDR LST)))))

(defun recips-p (r)
  (declare (xargs :guard t))
  (and (listp r)
       (email-p (car r))
       (email-listp (cdr r))))

(defstructure message
  (sender  (:assert (email-p sender)))
  (recips  (:assert (recips-p recips)))
  (headers (:assert (symbol-alistp headers)))
  (body    (:assert (listp body)))
  (:options :guards)
)

(defun recipient (msg)
  (declare (xargs :guard (and (message-p msg)
      (not (null (message-recips msg))))))
  (car (message-recips msg)))

(defmacro mk-email (user host)
  `(make-email :user ,user :host ,host))

(defmacro mk-message (s r h b)
  (if (listp b)
      `(make-message :sender ,s
     :recips ,r
     :headers (set-var 'from ,s
       (set-var   'to   ,r ,h))
     :body   ,b)
          `(make-message :sender ,s
     :recips ,r
     :headers (set-var 'from ,s
       (set-var   'to   ,r ,h))
     :body   '(,b))))


;;(wrap-msg-body message symbol -> message)
;; wraps body of message in wrapper, for example, encryption
(defun wrap-msg-body (msg wrapper)
;  (declare (xargs :guard (and (messagep msg)
;       (symbolp       wrapper))))
  (update-message msg
  :body
  (list wrapper (message-body msg))))

;;(get-msg-wrapper message) -> symbol
;; gets the wrapper around the message
(defun get-msg-wrapper (msg)
  (begin (cw "~x0 ~%" (stringify msg))
 (let ((wrapper (message-body msg)))
   (and (consp wrapper)
(car wrapper)))))


;;(get-msg-header string message -> header
;; get the header which contains
(defun get-msg-header (header msg)
  (get-var header (message-headers msg)))

;;(set-msg-header string string message -> message
;; add (replace) the header starting with <header> with <value>
(defun set-msg-header (header value msg)
  (update-message msg
  :headers
  (set-var header value (message-headers msg))))
```

```
;;(subject message) -> string
;; returns subject of message
(defun subject (msg)
  (get-msg-header 'subject msg))


(defun ret3 (status arg env)
  (declare (xargs :guard (and (symbolp status)
      (message-p arg)
      (symbol-alistp env))))
  (mv status arg env))

;; (return3 symbol msg env) -> (mv symbol msg env)
;;  syntactical sugar to return from feature
(defmacro act (status &optional arg env)
  (if (equal 'comment status)
      `(begin
(cw "Comment: ~x0~%" ,arg) ; output comment
(ret3 ',status msg env))
    (if env
`(ret3 ',status ,arg ,env)
`(ret3 ',status ,arg env))))

(defun action-commandp (action)
  (and (listp action)
       (listp (cdr action))
       (listp (cddr action))
       (listp (cdddr action))
  (let ((who  (car action))
(what (cadr action))
(arg1 (caddr action))
(arg2 (cadddr action)))
    (and (or (has-var who *hosts*)
     (has-var who *users*))
 (equal 'command what)
 (and (symbolp arg1)
      (memberp arg1 *commands*))
 (listp arg2)))))

(defun msg-type-p (msg)
  (declare (xargs :guard t))
  (and (listp msg)
       (listp (cdr msg))
       (listp (cddr msg))
       (listp (cdddr msg))
       (listp (cddddr msg))
       (let ((head (car msg))
     (from (cadr msg))
     (to   (cadddr msg))
     (body (cadddr (cdr msg))))
 (and
  (equal 'smessage head)
  (listp from)
  (listp to)
  (or (stringp body)
      (atom    body)
      (listp   body))))))

(defun action-sendp (action)
  (and (listp action)
       (listp (cdr action))
       (listp (cddr action))
       (let ((who  (car action))
     (what (cadr action))
     (msg  (caddr action)))
 (begin (cw "~x0 - ~x1 - ~x2~%" who what msg)
 (and (or (has-var who *hosts*)
  (has-var who *users*))
      (equal 'send what)
      (msg-type-p msg))))))

(defun actions-helper (actions status)
  (if (null actions)
      (equal 'end status)
    (if (listp actions)
 (let ((action (car actions))
       (rest   (cdr actions)))
   (if (equal 'init status)
       (if (equal 'init (car action))
   (actions-helper rest 'command)
 nil)
     (if (equal 'command status)
 (if (action-commandp action)
     (actions-helper rest 'command)
   (if (action-sendp action)
       (actions-helper rest 'end)
     nil))
       nil)))
      nil)))

(defun actions-listp (actions)
  (begin
  (actions-helper actions 'init)
  )
)

(defun get-vars (vals alist)
  (cond ((null vals) nil)
(t (and (listp vals)
(let ((val (car vals)))
  (acons
   val
   (get-var val alist)
   (get-vars (cdr vals) alist)))))))

(defun equal-or-one-off (l1 l2)
  (or (equal l1 l2)
      (equal (cdr l1) l2)))

(defun wrap-list (lst wrapper)
  (cons wrapper lst))

(defun unwrap-list (lst wrapper)
  (if (and (consp lst)
           (equal wrapper
                  (car lst)))
      (cdr lst)
    lst))

(defmacro mv-status  (lst) `(mv-nth 0 ,lst))
(defmacro mv-msg     (lst) `(mv-nth 1 ,lst))
(defmacro mv-env     (lst) `(mv-nth 2 ,lst))

(defmacro mv-actions (lst) `(mv-nth 0 ,lst))
(defmacro mv-users   (lst) `(mv-nth 1 ,lst))
(defmacro mv-hosts   (lst) `(mv-nth 2 ,lst))

(defmacro action-msg (act) `(action-arg1 ,act))
```

A.4 f-host.lisp

```
;;; EMAIL-HOST
;;; ---------

;;(email-host-init env) -> env
;; initializes environment
(defund email-host-init (env)
  (begin
   (cw "  [email-host-init]~%")
  (set-var 'hostname '()
  (set-var 'users '(postmaster) env))))


;;(email-host-command string list[args] env) ->
;;
  (begin
   (cw "  [email-host-command: ~x0 (~x1)]~%" (lookup 'hostname) cmd)
  (cond ((equal cmd 'SET_HOSTNAME)
         (let ((hn (car args)))
           (set-var 'hostname hn env)))
```

```
        ((equal cmd ’INIT_USER)
         (let ((uname (car args)))
    (set-var ’users (los-adjoin uname (lookup ’users)) env)))
        ((equal cmd ’DELETE_USER)
         (let ((uname (car args)))
           (set-var ’users (remove-equal uname (lookup ’users)) env)))
        (t env))))

;;(email-host-outgoing message env) ->
;;
(defund email-host-outgoing (msg env)
  (begin
   (cw "  [email-host-outgoing: ~x0]~%" (lookup ’hostname))
   (let ((recip (recipient msg))) ;
     (begin
      (cw ".....~x0~%" (stringify recip))
      (if (equal (lookup ’hostname)
 (email-host recip))
  (begin  (cw "delivering...")
  (let ((r-user (email-user recip)))
    (if (member-equal r-user (lookup ’users))
(begin (cw "..to ~x0~%" r-user)
       (if (message-p msg)
   (act deliver msg)
 (begin (cw "Message not msgp ~x0~%" msg)
(act deliver msg))))
      (begin
       (cw "no user~%")
       (let ((pm-addr (mk-email ’Postmaster (lookup ’hostname))))
 (act mail
      (mk-message pm-addr
  (list (message-sender msg))
  nil
  (list
   "Unknown user: "
   (stringify recip)
   "Following message undeliverable:"
   (stringify (message-headers msg))
   "e"
   (stringify (message-body msg))))))))))
(act mail msg))))))

;;(email-host-incoming message env) ->
(defund email-host-incoming (msg env)
  (begin
   (cw "  [email-host-incoming: ~x0]~%" (lookup 'hostname))
   (act comment "[Incoming events not handled]")))
```

A.5 f-encrypt.lisp

```
;;; EMAIL-ENCRYPT
;;; -------------

(defun encrypt-headers (headers key )
  (if (endp headers)
      nil
    (let* ((header  (car headers))
   (field   (car header))
   (content (cdr header)))
      (cond ((equal 'subject field)
     (cons (cons field (wrap-list content (list 'encrypted key)))
   (encrypt-headers (cdr headers) key )))
     (t (cons header
      (encrypt-headers (cdr headers) key )))))))

(defun encrypt-body (body key)
  (wrap-list body (list 'encrypted key)))

(defun encrypt-message (msg key nonce)
  (declare (ignore nonce))
  (make-message
     :sender (message-sender msg)
     :recips (message-recips msg)
     :headers (encrypt-headers (message-headers msg) key)
     :body    (encrypt-body    (message-body    msg) key)))
```

```
; email-encrypt-init env -> env
; initializes encryption variables
(defun email-encrypt-init  (env)
   (set-var 'e-corrs '()
   (set-var 'nonce-counter 0 env)))

(defun email-encrypt-command (cmd args env)
  (begin
   (cw "  [email-encrypt-command:  ~x0]~%" (lookup 'user))
   (cond ((equal 'SET_CORRESPONDENT_KEY cmd)
  (let ((corr (car args))
(key  (cadr args)))
    (if (member-equal corr (lookup 'e-corrs))
(if (equal '() key)
    (set-var 'e-corrs
     (remove-equal corr (lookup 'e-corrs))
     env)
  (set-hash-var corr key 'correspondent-key env))
      (set-var 'e-corrs (cons corr (lookup 'e-corrs))
       (set-hash-var corr key 'correspondent-key env)))))
 (t env))))

(defun email-encrypt-outgoing (msg env)
  (let ((?recip (recipient msg)))
    (begin
     (cw "  [email-encrypt-outgoing: ~x0]~%" (lookup 'user))
     (if (member-equal ?recip (lookup 'e-corrs))
 (let ((key   (lookup  'correspondent-key ?recip))
       (nonce (lookup 'nonce-counter)))
   (act mail
(encrypt-message msg key nonce)
(set-var  'nonce-counter (1+ nonce) env)))
       (act mail msg env)))))

(defun email-encrypt-incoming (msg env)
  (begin
   (cw "  [email-encrypt-incoming: ~x0]~%" (lookup 'user))
   (act comment "[Incoming events not handled]")))

(defun body-encrypted? (body)
  (and
   (listp body)
   (listp (car body))
   (equal (caar body)
          'encrypted)))

(defun encrypted? (msg)
  (and
   (message-p msg)
   (let ((body (message-body msg)))
     (body-encrypted? body))))
```

A.6 f-decrypt.lisp

```
;;; EMAIL-DECRYPT
;;; -------------

(defun decrypt-headers (headers key )
  (if (endp headers)
      nil
    (let* ((header  (car headers))
   (field   (car header))
   (content (cdr header)))
      (cond ((equal 'subject field)
     (cons (cons field (unwrap-list content (list 'encrypted key)))
   (decrypt-headers (cdr headers) key)))
     (t (cons header
      (decrypt-headers (cdr headers) key )))))))

(defthm unwrap-list-undoes-wrap-list
  (implies
   (equal key unkey)
  (equal
   (unwrap-list (wrap-list lst key) unkey)
   lst)))

(in-theory (enable unwrap-list wrap-list))

(defthm decrypt-headers-undoes-encrypt-headers
 (implies
  (and
   (symbol-alistp headers)
   )
  (equal
   (decrypt-headers
    (encrypt-headers headers key)
    key)
   headers)))

(defun decrypt-body (body key)
 (unwrap-list body (list 'encrypted key)))

(defun decrypt-message (msg key)
  (update-message msg
   :headers (decrypt-headers (message-headers msg) key)
   :body    (decrypt-body    (message-body    msg) key)))

(defthm body-encrypted-by-encrypt-body
 (implies
  (and
   body
   )
  (body-encrypted? (encrypt-body body key))))


(fif comment
(thm ;defthm body-encrypted-by-encrypt-body
 (implies
  (and
   (message-p msg)
   key
   nonce
   )
   (encrypt-message msg key nonce)))
)

(fif comment
(thm
 (implies
  (and
   (message-p msg)
   )
  (equal
   (decrypt-message
    (encrypt-message msg key nonce)
    key)
   msg)))
)

(defun invalid-decrypt-message-result (msg)
  (if (endp msg) nil
    (if (equal ’encrypted
       (cadr msg))
t
      (invalid-decrypt-message-result (cdr msg)))))

(defun email-decrypt-init (env)
  (set-var ’user ’()
  (set-var ’own-key ’() env)))

(defun email-decrypt-command (cmd args env)
   (cond ((equal ’SET_USER cmd)
  (let ((?user (nth 0 args)))
    (set-var ’user ?user env)))
 ((equal ’SET_OWN_KEY cmd)
  (let ((key (nth 0 args)))
    (set-var ’own-key key env)))
 (t env)))

(defun email-decrypt-outgoing (msg env)
   (act comment "[Outgoing events not handled"))

(defun email-decrypt-incoming (msg env)
  (let ((?user (LOOKUP 'user))
(?ok   (LOOKUP 'own-key)))
    (if (equal "" ?user)
(act comment "[No user set --> message discarded]")
      (if (equal ?ok "")
  (act deliver msg env)
(let ((?dmsg (DECRYPT-MESSAGE msg ?ok)))
  (if (invalid-decrypt-message-result ?dmsg)
      (act deliver msg env)
    (act deliver ?dmsg env)))))))
```

A.7 f-auto.lisp

```
;;; EMAIL-AUTO
;;; -----------------

;;(email-auto-init env) -> env
;; initialize envronment
(defund email-auto-init (env)
   (set-var 'already-answered '()
    (set-var 'user '()
     (set-var 'default-response '() env))))

;;(email-auto-command string list[<anything>] env -> env
;;
(defund email-auto-command (cmd args env)
  (cond ((equal 'SET_USER cmd)
         (let ((?user (car args)))
            (set-var 'user ?user env)))
        ((equal 'SET_DEFAULT_RESPONSE cmd)
 (let ((resp (car args)))
   (set-var 'already-answered '()
           (set-var 'default-response resp env))))
        (t env)))
```

```
;;(email-auto-outgoing message env) -> action
;; what transpires when an message is sent
(defund email-auto-outgoing (msg env)
  (begin
   (cw "  [email-auto-outgoing: ~x0]~%" (lookup 'user))
   (act comment "[Outgoing events not handled]")))

;;(email-auto-incoming message env) -> action
;; when a message is received
(defund email-auto-incoming (msg env)
   (if (equal '() (lookup 'user))
(act comment "    [User not set yet --> no action]")
     (begin
      (let ((?from (message-sender msg)))
 (if (member-equal ?from (lookup 'already-answered))
      (act comment "[No autoresponse, already answered -->
                  no further action]")
   (if (equal '() (lookup 'default-response))
       (act comment "[No default autoresponse --> no further action]~%")
     (let ((?recip    (recipient msg))
   (?response (lookup 'default-response)))
       (begin
(cw "  respond '~x0'to ~x1~%" ?response ?recip)
(act mail
     (mk-message
      ?recip
      (list (message-sender msg))
      (set-var 'subject (list 're (subject msg)) '())
      (cons ?response '()))
     (set-var
      'already-answered
      (cons ?from (lookup 'already-answered))
      env))))))))))
```

A.8 base.lisp

```
;;; SIMULATE NETWORK
;;; ----------------

;(defmacro glue (feature &rest ))

;(defmacro funcname (&rest names)
;  ‘(pack-intern "ACL" (,@names)))

;(defmacro callme (name &rest args)
;  (let ((call
;  (cons (funcname name) args)))
;    ‘(QUOTE ,call)))

; Usage:
;  (glue outgoing
;    (decrypt (deliver verify))
;
; Well, try this instead:
;  (glue
;    (email-decrypt-outgoing
;      (deliver email-verify-outgoing)
;    (email-verify-outgoing
;      (deliver email-auto-outgoing))

(defun msgtomessage (msg)
  (begin
   (cw "msgtomessage")
  (let ((head (car msg))
(from (cadr msg))
(to   (caddr msg))
(body (cadddr msg)))
    (if (equal ’smessage head)
(mk-message
 (mk-email (car from) (cadr from))
 (mk-email (car to)   (cadr to  ))
 ’()
 (list body))
      ’()))))

(defstructure action
  (type (:assert (member type ’(command init send mail deliver
                                  ok deliver-ok))))
  (name (:assert (and (not (null name)) (symbolp name))))
  (arg1 (:assert (or (equal ’init type)
     (if (member-equal type
       ’(send mail deliver))
 (message-p arg1)
       (not (null arg1))))))
  (arg2 (:assert (or (not (equal type ’command))
     (atom-listp arg2))))
  (:options :guards))


(defun action-listp (lst)
  (if (ATOM LST)
      (EQ LST NIL)
    (AND (action-P (CAR LST))
 (action-LISTP (CDR LST)))))

(defun mk-action4 (type name arg1 arg2)
  (make-action :type type
       :name name
       :arg1 arg1
       :arg2 arg2))

(defmacro mk-action (type &optional name arg1 arg2)
  (mk-action4 type name arg1 arg2))

(defun send-ok (msg user)
  (begin
   (cw "send-ok: ~%")
   (mv ’deliver msg (set-var ’sent-msg msg user))))

(defun send-abort (msg user)
  (mv 'abort msg user))

(defun deliver-ok (msg user)
  (mv 'deliver-ok msg (set-var 'recv-msg msg user)))

(defalias deliver-abort deliver-ok)

(defun recv-ok (msg user)
  (mv 'ok msg user))

(defun recv-abort (msg user)
  (mv 'abort msg user))

(fif address
     (defalias user-send-begin  user-send-address)
     (defalias user-send-begin  user-send-address-next))

(fif sign
     (defalias user-send-address-next user-send-sign)
     (defalias user-send-address-next user-send-sign-next))

(fif encrypt
     (defalias user-send-sign-next user-send-encrypt)
     (defalias user-send-sign-next user-send-encrypt-next))

(defalias user-send-encrypt-next user-send-ok)

(fif decrypt
     (defalias user-deliver-begin user-deliver-decrypt)
     (defalias user-deliver-begin user-deliver-decrypt-next))

(fif verify
     (defalias user-deliver-decrypt-next user-deliver-verify)
     (defalias user-deliver-decrypt-next user-deliver-verify-next))

(fif auto
     (defalias user-deliver-verify-next user-deliver-auto)
     (defalias user-deliver-verify-next user-deliver-auto-next))
```

```
(defalias user-deliver-auto-next user-deliver-ok)


(fif host
     (defalias host-send-begin host-send-host)
     (defalias host-send-begin host-send-host-next))

(defalias host-send-host-next host-send-ok)

;;(user-init user) -> user
;;
(defun user-init (user)
;  (declare (xargs :guard (symbol-alistp user)))
  (let-seq user
           (fif sign    (email-sign-init             user) user)
           (fif verify  (email-verify-init           user) user)
   (fif encrypt (email-encrypt-init          user) user)
   (fif decrypt (email-decrypt-init          user) user)
   (fif auto    (email-auto-init             user) user)
   (fif address (email-address-init          user) user)
           user))

(defun user-command-dummy (cmd args user)
  (declare (ignore cmd args))
  user)

;;(user-command user list[<anything>]) -> user
;;run through all commands
(defun user-command (act user)
;  (declare (xargs :guard (and (symbol-alistp user)
;       (true-listp    args))))
  (let ((cmd  (action-arg1 act))
(args (action-arg2 act)))
    (let-seq user
     (fif address (email-address-command cmd args user)
  user)
     (fif auto    (email-auto-command    cmd args user)
  user)
     (fif sign    (email-sign-command    cmd args user)
  user)
     (fif encrypt (email-encrypt-command cmd args user)
  user)
     (fif decrypt (email-decrypt-command cmd args user)
  user)
     (fif verify  (email-verify-command  cmd args user)
  user)
     (user-command-dummy cmd args user)
     user)))


;;;(user-send-*
;;;send user message

(defun user-send-ok (msg user)
  (begin
  (cw "user-send-ok: Message mailed (~x0): ~x1~%" msg user)
  (send-ok msg user)))

(defun user-send-abort (msg user)
  (begin
   (cw "Message aborted~%")
   (send-abort msg user)))

(fif encrypt
     (defund user-send-encrypt (msg user)
       (begin (cw "user-send-encrypt: ~x0 ~% ~x1~%" msg user)
              (mv-let (status new-msg new-user)
                      (email-encrypt-outgoing msg user)
                      (cond ((equal 'mail status)
                             (user-send-encrypt-next new-msg new-user))
                            (t (user-send-abort new-msg new-user)))))))


(fif sign
     (defun user-send-sign (msg user)
       (mv-let (status msg user)
       (email-sign-outgoing msg user)
       (let ((user (set-var 'sent-msg msg user)))
 (if (equal 'mail status)
                      (user-send-sign-next msg user)
      (user-send-abort     msg user))))))

(fif address
     (defun user-send-address (msg user)
       (mv-let (status msg user)
       (email-address-outgoing msg user)
       (let ((user (set-var 'sent-msg msg user)))
 (user-send-address-next msg user)))))

;;(user-send user msg) -> user
;;
(defun user-send (msg user)
      (begin (cw "user-send: ~x0 / ~x1~%" user (stringify msg))
  (user-send-begin msg user)))

;;;(user-deliver
(defun user-deliver-ok (msg user)
  (begin (cw "Message delivered~%")
    (deliver-ok msg user)))
;    (deliver-ok msg (set-var 'recv-msg msg user))))

(defun user-deliver-abort (msg user)
   (deliver-abort msg user))
;   (deliver-abort msg (set-var 'recv-msg msg user)))

(defmacro do-options (on-abort options)
  (let ((opt (car options))
(rst (cdr options)))
    (cond ((endp rst) `(if (equal ',(car opt) status)
   (,(cadr opt) msg user)
 (,on-abort msg user)))
   (t `(if (equal ',(car opt) status)
   (,(cadr opt) msg user)
 (do-options ,on-abort ,rst))))))

(defmacro do-next (fn callee on-abort options)
  `(defun ,fn (msg user)
     (mv-let (status msg user)
     (,callee msg user)
     (do-options ,on-abort ,options))))

(fif auto
     (defun user-deliver-auto-reply (msg user)
       (begin (cw "[user-deliver-auto-reply] Auto ~x0~%" msg)
      (mv 'mail msg user))))

(fif auto
     (defun user-deliver-auto (msg user)
       (mv-let (status new-msg new-user)
       (email-auto-incoming msg user)
       (begin
(cond ((equal 'deliver status)
       (user-deliver-auto-next  new-msg new-user))
      ((equal 'mail status)
       (user-deliver-auto-reply new-msg (set-var 'recv-msg msg new-user)))
      (t
       (user-deliver-abort      new-msg new-user))))))
     )

; should do the same as above...
;(do-next user-deliver-auto
;  email-auto-incoming
;  user-deliver-abort
;  ((deliver user-deliver-auto-next)
;   (mail    user-deliver-auto-reply)))

(fif verify
     (defun user-deliver-verify (msg user)
       (mv-let (status msg user)
       (email-verify-incoming msg user)
       (let* ((user (set-var 'recv-msg msg user)))
 (user-deliver-verify-next msg user)))))

(fif decrypt
     (defund user-deliver-decrypt (msg user)
       (mv-let (status new-msg new-user)
               (email-decrypt-incoming          msg user)
               (if status
                   (user-deliver-decrypt-next       new-msg new-user)
                   (user-deliver-decrypt-next       new-msg new-user)))))



;;(user-deliver symbol message) -> message
;;
(defun user-deliver (msg user)
  (begin
    (cw "    [user-deliver]~x0 to ~x1~%" msg user)
    (user-deliver-begin msg user)))

;;(clean-msg message) -> message
;;
;(defun clean-msg (msg)
;  (begin
;   ;(cw (string-append (string-append "MSG-->" (stringify msg)) "~%"))
;   (if (message-p msg)
;       msg
;     (let ((msg msg))
;       (begin
;        ;(cw (string-append (string-append "MSG2-->" (stringify msg)) "~%"))
;        (make-message
;         (nth 1 msg)
;         (nth 2 msg)
;         '()
;         (nth 3 msg))))))



;;(init-users list[users]) -> list[users]
```

```
;;
(defun init-users (users)
  (if (not (endp users))
      (let ((name (caar users))
            (user (cdar users)))
        (acons name (user-init user)
               (init-users (cdr users))))
    nil))


;;(clean-act act ->
;(defun clean-act (act)
;  (if (equal 'mail (car act))
;      (let ((msg (cadr act)))
; (begin
;  (cw "msg: ~x0--~x1~%" msg (car (sender msg)))
;  (list (car (sender msg))
;        'send
;        msg))
; )
;    act))

;;;; HOST FUNCTIONS
;;;;

(defun host-init (host)
  (let-seq host
           (fif host (email-host-init host) host)
           host))

(defun host-command (act host)
  (let ((cmd  (action-arg1 act))
(args (action-arg2 act)))
    (let-seq host
     (fif host
  (email-host-command  cmd args host)
  host)
     host)))
```

```
(defun host-send-ok (msg host)
  (begin
    (cw "MAILED~%")
    (send-ok msg host)))

(defun host-send-delivered (msg host)
  (begin
    (cw "host-send-delivered DELIVERED~%")
    (mv ’deliver msg (set-var ’recv-msg msg host))))

(defun host-send-abort (msg host)
  (send-abort msg host))

(fif host
     (defun host-send-host (msg host)
       (mv-let (status new-msg new-host)
       (email-host-outgoing msg host)
;        (let ((host (set-var ’sent-msg msg host)))
       (cond ((equal ’deliver status)
      (host-send-delivered new-msg new-host))
     ((equal ’mail    status)
      (host-send-host-next new-msg new-host))
     (t (host-send-abort new-msg new-host)))))
)

(defun host-send (msg host)
  (host-send-begin msg host))

; init-hosts list[hosts] -> list[hosts]
;
(defun init-hosts (hosts)
  (if (not (endp hosts))
      (let ((name (caar hosts))
            (host (cdar hosts)))
        (acons name (host-init host)
               (init-hosts (cdr hosts))))
    nil))
```

```
;;;; OTHER

(defun add-user-mail-action (type msg rest)
  (cons (mk-action
 type
 (email-user (message-sender msg))
 msg)
rest))

(defun add-sender-mail-action (type msg rest)
  (cons (mk-action
 type
 (email-user (message-sender msg))
 msg)
rest))


(defun add-host-mail-action (type msg rest)
  (cons (mk-action
 type
 (email-host (recipient msg))
 msg)
rest))

;;(do-init action list[actions] list[user] list[host]) ->
;;
(defun do-init (action rest users hosts)
  (declare (ignore action))
  (begin (cw "do-init~%")
  (mv rest
      (init-users users)
      (init-hosts hosts))))

;;(do-command action list[actions] list[user] list[host]) ->
; (BOB COMMAND "SET_USER" ("bob"))
(defun do-command (action rest users hosts)
  (begin
   (cw "do-command ~x0~%" action)
  (let* ((name (action-name  action)))
    (cond ((get-var name users)  ; USER COMMAND
           (let* ((user      (get-var name users))
                  (new-user  (user-command action user))
                  (new-users (set-var name new-user users)))
             (mv rest new-users hosts)))
          ((get-var name hosts)  ; HOST COMMAND
           (let* ((host      (get-var name hosts))
                  (new-host  (host-command action host))
                  (new-hosts (set-var name new-host hosts)))
             (mv rest users new-hosts)))
          (t (mv rest users hosts))))))

(defun do-user-send (action rest users hosts)
  (let* ((name (action-name action))
 (msg  (action-arg1 action)))
    (mv-let (status new-msg new-user)
    (user-send msg (get-var name users))
    (let ((new-users (set-var name new-user users)))
      (begin
       (cw "will send(~x0) ~x1, user: ~x2~%" status new-msg
   (email-host (recipient new-msg)))
       (if status
   (mv (add-host-mail-action 'send new-msg rest)
       new-users
       hosts)
 (mv rest new-users hosts)))))))



(defun do-host-send (action rest users hosts)
  (let* ((name (action-name action))
 (msg  (action-arg1 action)))
    (mv-let (status new-msg new-host)
    (host-send msg (get-var name hosts))
    (let ((new-hosts (set-var name new-host hosts)))
      (cond ((equal 'deliver status)
     (mv (add-user-mail-action ’deliver new-msg rest)
      users
      new-hosts))
    ((equal ’mail    status)
     ; "mail" to another host == "deliver" on other host
       (mv (add-user-mail-action ’deliver new-msg rest)
      users
      new-hosts))
    (t (mv rest users new-hosts))))))


;;(do-send action list[actions] list[user] list[host]) ->
; (BOB SEND (SMESSAGE (bob host) (rjh host) "Body line 1")))))
(defun do-send (action rest users hosts)
  (begin
   (cw "do-send:  ~x0~%" (stringify action))
   (let* ((name (action-name action))
  (msg  (action-arg1 action)))
    (cond ((get-var name users)  ; USER SEND
   (begin (cw "trying ~x0 ~x1~%" (get-var name users) msg)
  (do-user-send action rest users hosts)))
          ((get-var name hosts)
   (do-host-send action rest users hosts))
          (t (mv rest users hosts))))))

(defun add-mail-action (msg rest)
  (cons (mk-action
 ’mail
 (email-user (message-sender msg))
 msg)
rest))

;;(do-deliver action list[actions] list[user] list[host]) ->
(defun do-deliver (action rest users hosts)
  (begin
   (cw "Doing delivery of ~x0~%" (stringify action))
  (let* ((name (action-name action))
         (msg  (action-arg1 action)))
    (cond ((get-var name users)  ; USER DELIVER
           (mv-let (status new-msg new-user)
                   (user-deliver msg (get-var name users))
                   (let ((new-users (set-var name new-user users)))
                     (if (equal ’mail status)
 (begin
  (cw "acted as ~x0 ~x1, user: ~x2~%" status new-msg
      (email-user (message-sender msg)))
;   (mv (add-user-mail-action ’mail new-msg rest)
  (mv (add-user-mail-action ’send new-msg rest)
      new-users
      hosts))
                     (mv rest new-users hosts)))))
          (t (mv rest users hosts))))))

;;;(do-mail action list[actions] list[user] list[host]) ->
(defun do-mail (action rest users hosts)
  (let* ((name (action-name action))
         (msg  (action-arg1 action)))
    (cond ((get-var name users)  ; USER MAIL
           (mv-let (status msg new-user)
                   (user-send msg (get-var name users))
                   (let ((new-users (set-var name new-user users)))
                     (if status
 (begin
  (cw "here be dragons (~x2) ~x0 with ~x1 ~%" (message-sender msg) msg status)
  (mv
   (if (equal status ’mail)
       (add-sender-mail-action ’mail
       msg
       rest)
     (if (equal status ’deliver)
  (add-user-mail-action ’deliver
msg
rest)
       rest))
   new-users
   hosts))
```

```
       (mv rest new-users hosts)))))
          (t (mv rest users hosts)))))


(defun do-action-cond (action rest users hosts)
  (let ((type   (action-type action)))
    (cond ((equal ’init    type)
   (do-init    action rest users hosts))
  ((equal ’send    type)
   (do-send    action rest users hosts))
  ((equal ’deliver type)
   (do-deliver action rest users hosts))
  ((equal ’mail type)
   (do-mail    action rest users hosts))
  ((equal ’command type)
   (do-command action rest users hosts))
  (t (begin
      (cw "Can’t execute command ~x0~%" action)
      (mv rest users hosts))))))

;;(do-actions action list[actions] list[user] list[host]) ->
(defun do-actions (actions count users hosts)
  (declare (xargs :measure (acl2-count count)))
  (if (and (> count 0) (integerp count))
      (begin
       (cw "do-actions: ~x0~%" (stringify (car actions)))
      (cond ((endp actions) (mv ’end users hosts))
            (t (let* ((action (car         actions))
                      (rest   (cdr         actions)))
 (mv-let (new-actions new-users new-hosts)
 (do-action-cond action rest users hosts)
 (do-actions new-actions (- count 1) new-users new-hosts ))))))
    (mv ’error users hosts)))

(defun parse-actions (actions)
  (begin
   (cond ((endp actions) ’())
 (t
  (cons
   (let ((act (car actions)))
     (if (equal 'init (car act))
 (mk-action 'init)
       (if (equal 'command (cadr act))
   (mk-action
    'command
    (car act)
    (caddr act)
    (cadddr act))
 (mk-action
  'send
  (car act)
  (msgtomessage (caddr act))))))
   (parse-actions (cdr actions))))))

;;(simulate-network list[actions])
;;sets up initial env and calls recursive simulation func
(defun simulate-network (actions)
  (do-actions (parse-actions actions) 20 *users* *hosts*))

;;(simulate-network-with-env list[action] env)
;;
(defun simulate-network-with-env (actions env)
  (do-actions (parse-actions actions)
      20
              (get-var 'users env)
              (get-var 'hosts env)
              ))
```

A.9 theorems.lisp

```
(set-ignore-ok t)
(set-match-free-default :all)

(ld "p-macros.lisp")
(ld "p-util.lisp")
```

```
(fif host (ld "p-host.lisp"))
(fif auto (ld "p-auto.lisp"))

(ld "p-other.lisp")
(ld "p-interaction.lisp")
```

A.10 p-macros.lisp

```
(defun inject-goals (new-goals goals)
  (cond ((endp new-goals) goals)
(t (let* ((goal (car new-goals))
  (key  (car goal))
  (rest (cdr new-goals)))
     (begin
      (cw "you have ~x0 and ~x1~%" goals new-goals)
     (inject-goals
      rest
      (put-assoc-equal key
       (append (cdr (assoc-equal key goals))
       (cdr goal))
       goals)))))))

;; (new-thm symbol symbol) -> acl2-internal-symbol
;; returns concatenated thm-x-y as new acl2 internal symbol,
;; suitable for using as defthm or defun name
(defun new-thm (x y)
  (intern (concatenate 'string
       "THM-"
       (stringify x)
       "-"
       (stringify y))
  "ACL2"))

(defun catsym (x y)
  (intern (concatenate 'string
       (stringify x)
       (stringify y))
  "ACL2"))

;; (make-returns-symbol-alistp-thm symbol) -> thm-func-returns-symbol-alistp
;; Makes theorem which theorizes that a feature function returns
;; a symbol-alistp
(defmacro make-*/mail-returns-symbol-alistp-thm (func &optional hints)
  `(defthm ,(new-thm func 'returns-symbol-alistp)
     (implies
       (and
(message-p msg)
(symbol-alistp env)
)
       (mv-let (s new-msg new-env)
       (,func msg env)
       (symbol-alistp new-env)))

     :hints ,(inject-goals (list (list "Goal"
                 ':in-theory (list 'enable func))) hints)))


(defmacro make-*/mail-returns-symbolp-thm (func &optional hints)
  `(defthm ,(new-thm func 'returns-symbolp)
       (mv-let (s new-msg new-env)
       (,func msg env)
       (symbolp s))
     :hints ,(inject-goals (list (list "Goal"
                 ':in-theory (list 'enable func))) hints)))

;; (make-returns-same-env-thm symbol) -> thm-func-returns-same-env
;; Makes theorem which theorizes that a feature function returns the
;; same env passed into it.
;;
(defmacro make-*/mail-returns-same-env-thm (func &optional hints)
  `(defthm ,(new-thm func 'returns-same-env)
     (implies
      (and
       (message-p msg)
       (symbol-alistp env))
      (mv-let (s new-msg new-env)
      (,func msg env)
      (equal new-env env)))
     :hints ,(inject-goals (list (list "Goal"
               ':in-theory (list 'enable func))) hints)))

;; (make-*/mail-returns-message-p-thm symbol) -> thm-func-returns-message-p-thm
;; Makes theorem which theorized that a feature function returns
;; always a message-p message.
(defmacro make-*/mail-returns-message-p-thm (func &optional hints)
  `(defthm ,(new-thm func 'returns-message-p)
     (implies
      (and
       (message-p msg)
       )
      (message-p (mv-nth 1 (,func msg env))))
     :rule-classes (:rewrite (:FORWARD-CHAINING :TRIGGER-TERMS
                        ((mv-nth 1 (,func msg env)))))
     :hints ,(inject-goals (list (list "Goal"
            ':in-theory (list 'enable func))) hints)))

;; (add-has-vars-conditions list[symbol]) ->
;; Helper function for make-adds-x-variables-thm
(defmacro add-has-var-conditions (vars)
  (let ((variable (car vars))
(rst      (cdr vars)))
  (cond ((endp rst) `(has-var (quote ,variable)  new-env))
(t `(and (has-var (quote ,variable) new-env)
 (add-has-var-conditions ,rst))))))

;; (make-*/init-adds-x-variables-thm symbol list[symbol])
;; Makes theorem which theorizes that feature function adds
;; only particular variables to env
(defmacro make-*/init-adds-x-variables-thm (func vars &optional hints)
  `(defthm ,(new-thm func 'adds-x-variables)
     (implies
      (symbol-alistp env)
      (let ((new-env
     (,func env)))
(add-has-var-conditions ,@(cdr vars))))
     :hints ,(inject-goals (list (list "Goal"
                ':in-theory (list 'enable func))) hints)))


;; (make-changes-only-x-variables-thm symbol list[symbol])
;; Makes theorem which theorizes that feature function does not change
;; variables other than specified
(defmacro make-*/init-changes-only-x-variables-thm (func variables)
  `(defthm ,(new-thm func 'changes-only-x-variables)
     (implies
      (and
       (symbol-alistp env)
       (equal (get-var key env)
      var)
       (not (member key ,variables))
      )
     (let ((new-env
    (,func env)))
       (equal (get-var key new-env)
      var)))))

;; (make-add-and-changes-only-x-variables-thm symbol list[symbol])
;; Makes theorem which theorizes that feature function adds only
;; variables specified, and that no other variables are added
(defmacro make-*/init-add-and-changes-only-x-variables-thm
                           (func variables &optional hints)
  `(defthm ,(new-thm func 'adds-and-changes-only-x-variables)
     (implies
      (and
       (symbol-alistp env)
       (equal (get-var key env)
      var)
       (not (member key ,variables))
      )
     (let ((new-env
    (,func env)))
```

```
       (and
(equal (get-var key new-env)
       var)
(add-has-var-conditions ,@(cdr variables))
       )))
     :hints ,(inject-goals (list (list "Goal"
        ':in-theory (list 'enable func))) hints)))


(defmacro make-f/comm-add-and-changes-only-x-variables-thm
      (func variables &optional hints)
  `(defthm ,(new-thm func 'adds-and-changes-only-x-variables)
     (implies
      (and
       (symbol-alistp env)
       (equal (get-var key env)
      var)
       (not (member key ,variables))
      )
     (let ((new-env
    (,func cmd args env)))
       (and
(equal (get-var key new-env)
       var)
(add-has-var-conditions ,@(cdr variables))
       )))
     :hints ,(inject-goals (list (list "Goal"
         ':in-theory (list 'enable func))) hints)))

(defmacro make-*/mail-returns-same-message-thm (func &optional hints)
  `(defthm ,(new-thm func 'returns-same-message)
     (implies
      (and
       (symbol-alistp user)
       msg)
      (mv-let (new-status new-msg new-user)
       (,func msg user)
       (equal new-msg msg)))
     :hints ,(inject-goals (list (list "Goal"
              ’:in-theory (list ’enable func))) hints)))


;(defmacro make-user-*-returns-action-listp-thm (func &optional hints)
; un-needed due to not dealing with action-list

(defmacro make-d/that-returns-action-listp-thm (func &optional hints)
  ‘(defthm ,(new-thm func ’returns-action-listp)
     (implies
      (and
       (action-p action)
       (action-listp rest)
       (symbol-alistp users)
       (symbol-alistp hosts)
       )
      (mv-let (new-rest new-users new-hosts)
      (,func action rest users hosts)
      (action-listp new-rest)))
     :hints ,(inject-goals (list (list "Goal"
       ’:in-theory (list ’enable func))) hints)))


(defmacro make-d/that-returns-action-listp-if-message-p-thm
   (func &optional hints)
  ‘(defthm ,(new-thm func ’returns-action-listp)
     (implies
      (and
       (action-p action)
       (message-p (action-arg1 action))
       (action-listp rest)
       (symbol-alistp users)
       (symbol-alistp hosts)
       )
      (mv-let (new-rest new-users new-hosts)
      (,func action rest users hosts)
      (action-listp new-rest)))
     :hints ,(inject-goals (list (list "Goal"
       ’:in-theory (list ’enable func))) hints)))

(defmacro make-*/mail-returns-superset-of-user-thm (func &optional hints)
  ‘(defthm ,(new-thm func ’returns-superset-of-user)
     (implies
      (and
       (symbol-alistp user)
       msg
       )
       (mv-let (new-status new-msg new-user)
       (,func msg user)
       (env-subset user new-user)))
     :rule-classes :forward-chaining
     :hints ,(inject-goals (list (list "Goal"
         ’:in-theory (list ’enable func))) hints)))


(defmacro make-f/comm-returns-superset-of-user-thm
        (func &optional hints)
  ‘(defthm ,(new-thm func ’returns-superset-of-user)
     (implies
      (and
       (symbol-alistp user)
       args
       cmd
)
       (let ((new-user (,func cmd args user)))
 (env-subset user new-user)))
     :rule-classes :forward-chaining
     :hints ,(inject-goals (list (list "Goal"
        ’:in-theory (list ’enable func))) hints)))



(defmacro make-d/that-returns-superset-of-users-thm (func &optional hints)
  ‘(defthm ,(new-thm func ’returns-superset-of-users)
     (implies
      (and
       (symbol-alistp users)
       (symbol-alistp hosts)
       msg)
       (mv-let (new-rest new-users new-hosts)
       (,func action rest users hosts)
       (env-subset users new-users)))
     :hints ,(inject-goals (list (list "Goal"
        ':in-theory (list 'enable func))) hints)))


(defmacro make-d/that-returns-symbol-alistp-users-thm (func &optional hints)
  `(defthm ,(new-thm func 'returns-symbol-alistp-users)
     (implies
      (and
       (symbol-alistp users)
       (symbol-alistp hosts)
       (symbolp (action-name action))
       ;msg
       )
       (mv-let (new-rest new-users new-hosts)
       (,func action rest users hosts)
       (symbol-alistp new-users)))
     :hints ,(inject-goals (list (list "Goal"
        ':in-theory (list 'enable func))) hints)))


(defmacro make-*/init-returns-symbol-alistp-thm (func &optional hints)
  `(defthm ,(new-thm func 'returns-symbol-alistp)
     (implies
      (and
       (symbol-alistp env)
       msg)
       (let ((new-env (,func env)))
 (symbol-alistp new-env)))
     :hints ,(inject-goals (list (list "Goal"
        ':in-theory (list 'enable func))) hints)))
```

```
(defmacro make-d/that-returns-symbol-alistp-users-thm (func &optional hints)
  `(defthm ,(new-thm func 'returns-symbol-alistp-users)
     (implies
      (and
       (symbol-alistp users)
       (symbol-alistp hosts)
       (symbolp (action-name action))
       )
       (mv-let (new-rest new-users new-hosts)
       (,func action rest users hosts)
       (symbol-alistp new-users)))
     :hints ,(inject-goals (list (list "Goal"
         ':in-theory (list 'enable func))) hints)))


(defmacro make-d/that-returns-symbol-alistp-hosts-thm (func &optional hints)
  `(defthm ,(new-thm func 'returns-symbol-alistp-hosts)
     (implies
      (and
       (symbol-alistp hosts)
       (symbol-alistp users)
       (symbolp (action-name action))
       )
       (mv-let (new-rest new-users new-hosts)
       (,func action rest users hosts)
       (symbol-alistp new-hosts)))
     :hints ,(inject-goals (list (list "Goal"
        ':in-theory (list 'enable func))) hints)))



(defmacro make-d/that-returns-superset-of-hosts-thm (func &optional hints)
  `(defthm ,(new-thm func 'returns-superset-of-hosts)
     (implies
      (and
       (or rest (null rest))
       (or action (null action))
       (symbol-alistp users)
       (symbol-alistp hosts)
       msg)
       (mv-let (new-rest new-users new-hosts)
       (,func action rest users hosts)
       (env-subset hosts new-hosts)))
     :hints ,(inject-goals (list (list "Goal"
        ':in-theory (list 'enable func))) hints)))



(defmacro make-*/init-returns-superset-of-env-thm (func &optional hints)
  `(defthm ,(new-thm func 'returns-superset-of-env)
     (implies
      (and
       (symbol-alistp env)
       )
      (let ((new-env (,func env)))
(env-subset env new-env)))
     :hints ,(inject-goals (list (list "Goal"
        ':in-theory (list 'enable func))) hints)))



(defmacro make-*/mail-returns-same-user-thm (func &optional hints)
  `(defthm ,(new-thm func 'returns-same-user)
     (implies
      (and
       (symbol-alistp user)
       )
       (mv-let (new-status new-msg new-user)
       (,func msg user)
       (equal new-user user)))
     :hints ,(inject-goals (list (list "Goal"
        ':in-theory (list 'enable func))) hints)))


(defmacro make-*/mail-returns-superset-of-user-if-message-p-thm
  (func &optional hints)
  `(defthm ,(new-thm func 'returns-superset-of-user-if-message-p)
     (implies
      (and
       (symbol-alistp user)
       (message-p msg))
       (mv-let (new-status new-msg new-user)
       (,func msg user)
       (env-subset user new-user)))
     :hints ,(inject-goals (list (list "Goal"
        ':in-theory (list 'enable func))) hints)))


(defmacro make-d/that-returns-no-actions (func &optional hints)
  `(defthm ,(new-thm func 'returns-no-actions)
     (implies
      t
      (mv-let (new-rest new-users new-hosts)
      (,func action rest users hosts)
      (equal new-rest rest)))
     :hints ,hints))

(defmacro make-d/that-returns-at-most-one-action (func &optional hints)
  `(defthm ,(new-thm func 'returns-no-actions)
     (implies
      t
      (mv-let (new-rest new-users new-hosts)
      (,func action rest users hosts)
      (equal-or-one-off new-rest rest)))
     :hints ,hints))

(defun sender-becomes-recipient (a b)
  (equal
   (message-sender    a)
   (recipient         b)))
```

A.11 p-util.lisp

```
; Utility functions for theorems
;

(defthm env-subset-reflexive
 (implies
  (and
   (symbol-alistp l)
   )
  (env-subset l l)))

(defthm env-subset-acons
 (implies
  (and
   (env-subset a b)
   )
  (env-subset a (acons k v b))))

(defthm env-subset-set-var
 (implies
  (and
   (env-subset a b)
   )
  (env-subset a (acons k v b))))


(defthm env-subset-transitive
 (implies
  (and
   (env-subset a b)
   (env-subset b c)
   )
  (env-subset a c))
 :hints
   (("Goal'" :use (:instance subsetp-equal-transitive
     (a (domain a))
     (b (domain b))
     (c (domain c))))))

(defthm same-is-subset
  (implies
   (and
    (symbol-alistp e)
    (symbol-alistp e2)
    (equal e e2)
    )
   (env-subset e e2)))

(defthm set-var-is-subset
   (implies
    (and
     (symbol-alistp env)
     (symbol-alistp e2)
     (env-subset env e2)
     (symbolp k)
     )
    (env-subset env (set-var k v e2))))

(defthm equal-is-subset
   (implies
    (and
     (symbol-alistp env)
;     (symbol-alistp e2)
;     (equal env e2)
;     (subset env e2)
     )
    (env-subset env env)))

(defthm equal-set-var-is-subset
   (implies
    (and
     (symbol-alistp env)
     (symbolp k)
     )
    (env-subset env (set-var k v env))))
```

```
(defthm mk-action-returns-action-p-thm
  (implies
  (and
   (message-p msg)
   (symbolp user)
   (not (null user))
   )
  (action-p (mk-action 'send user msg arg2))))

(defthm mk-email-returns-email-p-thm
  (implies
   (and
    (and
     (symbolp to)
     (not (null to)))
    (and
     (symbolp host)
     (not (null host)))
    )
   (email-p (mk-email to host))))

(defthm message-sender-returns-email-p
  (implies
   (and
    (message-p msg)
    )
   (email-p
    (message-sender msg))))

(defthm mk-message-returns-message-p-thm
  (implies
   (and
    (email-p       to)
    (email-p     from)
    (equal (list to) tos)
    (symbol-alistp headers)
    (listp         body)
    )
   (message-p
    (mk-message from
tos
headers
body))))

(defthm add-mail-action-is-action-listp
 (implies
  (and
   (message-p msg)
   (action-listp rest))
  (action-listp
   (add-mail-action msg rest))))



(defthm add-user-mail-action-arg1
  (equal (action-msg (car (add-user-mail-action type msg rest)))
 msg))

(defthm add-user-mail-action-returns-action-listp
 (implies
  (and
   (member-equal type '(deliver mail send))
   (message-p msg)
   (action-listp rest))
  (action-listp (add-user-mail-action type msg rest))))

(defthm add-sender-mail-action-returns-action-listp
 (implies
  (and
   (member-equal type '(deliver mail send))
   (message-p msg)
   (action-listp rest))
  (action-listp (add-sender-mail-action type msg rest))))

(defthm add-host-mail-action-returns-action-listp
 (implies
  (and
   (member-equal type '(deliver mail send))
   (message-p msg)
   (action-listp rest))
  (action-listp (add-host-mail-action type msg rest))))
```

A.12 p-host.lisp

```
(make-*/mail-returns-symbol-alistp-thm email-host-outgoing)
(make-*/mail-returns-same-env-thm      email-host-outgoing)
(make-*/mail-returns-message-p-thm     email-host-outgoing)

(make-*/mail-returns-symbol-alistp-thm email-host-incoming)

(make-*/mail-returns-same-env-thm      email-host-incoming)
(make-*/mail-returns-message-p-thm     email-host-incoming)

(make-*/init-add-and-changes-only-x-variables-thm email-host-init
   '(hostname users))
```

A.13 p-auto.lisp

```
(make-*/mail-returns-symbol-alistp-thm email-auto-outgoing)
(make-*/mail-returns-same-env-thm      email-auto-outgoing)
(make-*/mail-returns-message-p-thm     email-auto-outgoing)
(make-*/mail-returns-symbolp-thm       email-auto-outgoing)

(make-*/mail-returns-symbolp-thm       email-auto-incoming)
(make-*/mail-returns-symbol-alistp-thm email-auto-incoming)
(make-*/mail-returns-message-p-thm     email-auto-incoming)
(make-*/mail-returns-superset-of-user-thm email-auto-incoming)

(make-*/init-add-and-changes-only-x-variables-thm email-auto-init
   '(already-answered user default-response))
```

```
;; (user-in-already-answered user env) -> boolean
;; checks whether user in in 'already-answered list
(defun user-in-already-answered (user env)
  (member-equal user (get-var 'already-answered env)))

; example usage of user-in-already-answered
(user-in-already-answered (email 'user 'host)
  (set-var 'already-answered
   (list (email 'u 'h)
    (email 'user 'host))
   '()))

;;;;;;;;email-auto-incoming/*;;;;;;;;;;;;;

;; email-auto-incoming/auto-response-if-not-already-answered
;;   if autoresponder enabled, then a message sent to user
;;   will result in one auto-response
(defthm email-auto-incoming/auto-response-if-not-already-answered
  (implies
   (and
    (lookup 'default-response)             ; autoresponder enabled
    (lookup 'user)
    (equal sender (message-sender msg))
    (not (member-equal sender (lookup 'already-answered)))
    )
   (mv-let (s new-msg new-env)
   (email-auto-incoming msg env)
   (and (equal s 'mail)
(equal (recipient new-msg) sender)
)
))
  :hints (("Goal" :in-theory (enable email-auto-incoming))))

(defthm email-auto-incoming/auto-response-if-not-already-answered-mail-action
  (implies
   (and
    (lookup 'default-response)             ; autoresponder enabled
    (lookup 'user)
    (equal sender (message-sender msg))
    (not (member-equal sender (lookup ’already-answered)))
    )
   (mv-let (s new-msg new-env)
   (email-auto-incoming msg env)
   (equal s ’mail)
))
  :hints (("Goal" :in-theory (enable email-auto-incoming))))


;; email-auto-incoming/no-auto-response-if-already-answered
;;   If enabled, subsequent messages from the same user result in
;;   no additional messages
(defthm email-auto-incoming/no-auto-response-if-already-answered
  (implies
   (and
    (equal (message-sender msg) sender)
    (user-in-already-answered sender env)
    )
   (not
    (equal (mv-status (email-auto-incoming msg env))
   ’mail)))
  :hints (("Goal" :in-theory (enable email-auto-incoming))))

;; email-auto-incoming/auto-response-adds-sender-to-already-answered
;;   When sending an autoresponse, sender address is added to
;;   ’already-answered list and will thus not receive additional
;;   auto-responses
(defthm email-auto-incoming/auto-response-adds-sender-to-already-answered
  (implies
   (and
    (lookup ’default-response)
    (lookup ’user)
    (equal sender (message-sender msg))
    (not (member-equal sender (lookup ’already-answered)))
    )
   (user-in-already-answered sender (mv-env (email-auto-incoming msg env))))
  :hints (("Goal" :in-theory (enable email-auto-incoming))))
```

```
(defthm email-auto-incoming/something
  (implies
   (and
    (lookup 'default-response)             ; autoresponder enabled
    (lookup 'user)
    (equal sender (message-sender msg))
    (not (member-equal sender (lookup 'already-answered)))
    )
   (mv-let (s new-msg new-env)
   (email-auto-incoming msg env)
   (equal s 'mail)))
  :hints (("Goal" :in-theory (enable email-auto-incoming))))

(defthm email-auto-incoming/auto-response-recipient-becomes-message-sender
  (implies
   (and
    (lookup 'default-response)             ; autoresponder enabled
    (lookup 'user)
    (equal sender (message-sender msg))
    (not (member-equal sender (lookup 'already-answered)))
    )
    (equal
     (message-sender (mv-msg (email-auto-incoming msg env)))
     (recipient msg)))
  :hints (("Goal" :in-theory (enable email-auto-incoming))))

(defthm email-auto-incoming/auto-response-message-sender-becomes-recipient
  (implies
   (and
;    (message-p msg)
    (get-var 'default-response env)   ; autoresponder enabled
    (get-var 'user             env)
    (equal (message-sender msg) sender)
    (not (member-equal sender (get-var 'already-answered env)))
    )
   (sender-becomes-recipient
     msg
```

```
      (mv-msg (email-auto-incoming msg env))))
;    (equal
;     (recipient (mv-msg (email-auto-incoming msg env)))
;     sender))
  :hints (("Goal" :in-theory (enable email-auto-incoming))))
(user-deliver
 (mk-message
  (mk-email 'rjh 'host)
  (list (mk-email 'bob 'host))
  '()
  '(a body))
 (set-var 'default-response 'res
 (set-var 'user             'rjh
 (set-var 'already-answered '()
     '())))))

(defthm email-auto-incoming/auto-response-message-rewrite
  (implies
   (and
    (message-p msg)
    (get-var 'default-response env)   ; autoresponder enabled
    (get-var 'user             env)
    (equal (message-sender msg) sender)
    (equal (recipient msg) recip)
    (not (member-equal sender (get-var 'already-answered env)))
    )
    (equal
     (mv-msg (email-auto-incoming msg env))
     (mk-message recip (list sender)
 (set-var 'subject (list 're (subject msg)) '())
 (cons (get-var 'default-response env) '())))))
  :hints (("Goal" :in-theory (enable email-auto-incoming)))
  :rule-classes (:rewrite :forward-chaining))

(defthm email-auto-incoming/already-answered-added
  (implies
   (and
    (lookup 'default-response)             ; autoresponder enabled
    (lookup 'user)
    (equal sender (message-sender msg))
    (not (member-equal sender (lookup 'already-answered)))
    )
   (equal
    (mv-env (email-auto-incoming msg env))
    (set-var 'already-answered
     (cons sender (get-var 'already-answered env))
     env)))
  :hints (("Goal" :in-theory (enable email-auto-incoming))))
```

```
;; email-auto-incoming/auto-response-adds-sender-to-already-answered
(fif comment
    (defthm email-auto-incoming/xxxx
  (implies
   (and
    (lookup 'default-response)
    (lookup 'user)
    (equal sender (message-sender msg))
    (not (member-equal sender (lookup 'already-answered)))
    (not (lookup k))
    (symbolp k)
    )
   (mv-let (s new-msg new-env)
   (email-auto-incoming msg env)
   (equal (get-var k env)
  (get-var k new-env))))
  :hints (("Goal" :in-theory (enable email-auto-incoming))))
)
```

A.14 p-other.lisp

```
(make-*/mail-returns-superset-of-user-thm send-ok)
(make-*/mail-returns-superset-of-user-thm user-send-ok
  (("Goal'" :use (:instance thm-send-ok-returns-superset-of-user))))
```

```
(make-*/mail-returns-superset-of-user-thm user-send
  (("Goal'" :use (:instance thm-user-send-ok-returns-superset-of-user))))

(make-*/mail-returns-superset-of-user-thm email-host-outgoing)
(make-*/mail-returns-same-user-thm email-host-outgoing)
(make-*/mail-returns-same-user-thm email-host-incoming)
(make-*/mail-returns-same-env-thm send-abort)
(make-*/mail-returns-superset-of-user-thm send-abort)
(make-*/mail-returns-superset-of-user-thm host-send-abort
   (("Goal'" :use (:instance thm-send-abort-returns-superset-of-user))))
(make-*/mail-returns-superset-of-user-thm host-send-delivered)
(make-*/mail-returns-same-user-thm send-abort)
(make-*/mail-returns-same-user-thm host-send-abort)
(make-*/mail-returns-superset-of-user-thm host-send-ok
  (("Goal'" :use thm-send-ok-returns-superset-of-user)))

(make-*/mail-returns-symbol-alistp-thm email-host-outgoing)
(make-*/mail-returns-superset-of-user-thm email-host-outgoing)
(make-*/mail-returns-superset-of-user-thm host-send-delivered)
(expand (make-*/mail-returns-superset-of-user-thm host-send-ok))
(make-*/mail-returns-superset-of-user-thm host-send-host)
;   (("Subgoal 3" :use (thm-host-send-delivered-returns-superset-of-user
;        thm-email-host-outgoing-returns-message-p))))
(make-*/mail-returns-superset-of-user-thm host-send)

;(expand (make-*/mail-returns-superset-of-user-thm host-command))
;(make-f/comm-returns-superset-of-user-thm  email-host-command)
(make-f/comm-returns-superset-of-user-thm  email-auto-command)

(make-*/init-returns-symbol-alistp-thm init-hosts)
(make-d/that-returns-symbol-alistp-hosts-thm do-init)

(make-*/mail-returns-message-p-thm send-ok)
(make-*/mail-returns-message-p-thm user-send-ok)
(make-*/mail-returns-message-p-thm user-send)
(make-*/mail-returns-message-p-thm host-send-delivered)
(make-*/mail-returns-message-p-thm email-host-outgoing)
(make-*/mail-returns-message-p-thm host-send-ok)
```

```
(make-*/mail-returns-message-p-thm send-abort)
(make-*/mail-returns-message-p-thm host-send-abort)
(make-*/mail-returns-message-p-thm host-send-host)
(make-*/mail-returns-message-p-thm host-send)

(make-*/init-returns-superset-of-env-thm init-hosts)
(make-d/that-returns-superset-of-hosts-thm do-init )
(make-d/that-returns-superset-of-hosts-thm do-send)
(make-d/that-returns-superset-of-hosts-thm do-deliver)
(make-d/that-returns-superset-of-hosts-thm do-mail)
(make-d/that-returns-superset-of-hosts-thm do-command)
(make-d/that-returns-superset-of-users-thm do-init);
(make-d/that-returns-superset-of-users-thm do-send)
(make-d/that-returns-superset-of-users-thm do-deliver)
(make-d/that-returns-superset-of-users-thm do-mail)
(make-d/that-returns-superset-of-users-thm do-command)

(make-d/that-returns-symbol-alistp-hosts-thm do-deliver)
(make-d/that-returns-symbol-alistp-hosts-thm do-mail)
;(make-d/that-returns-symbol-alistp-hosts-thm do-send)

(make-d/that-returns-symbol-alistp-users-thm do-init)
(make-d/that-returns-symbol-alistp-users-thm do-send)
(make-d/that-returns-symbol-alistp-users-thm do-deliver)
(make-d/that-returns-symbol-alistp-users-thm do-mail)
(make-d/that-returns-symbol-alistp-users-thm do-command)

(make-d/that-returns-action-listp-thm do-init)
(make-d/that-returns-action-listp-thm do-command)


(make-*/mail-returns-superset-of-user-thm email-auto-incoming)
(make-*/mail-returns-superset-of-user-thm deliver-ok)
(make-*/mail-returns-superset-of-user-thm user-deliver-abort)
(make-*/mail-returns-superset-of-user-thm user-deliver-auto-reply)
(make-*/mail-returns-symbol-alistp-thm user-deliver-auto-reply)
(make-*/mail-returns-symbol-alistp-thm deliver-ok)
(make-*/mail-returns-symbol-alistp-thm user-deliver-ok)
```

```
(make-*/mail-returns-symbol-alistp-thm user-deliver-abort)
(make-*/mail-returns-symbol-alistp-thm email-auto-incoming)
(make-*/mail-returns-symbol-alistp-thm email-auto-incoming)
(make-*/mail-returns-message-p-thm email-auto-incoming)
(make-*/mail-returns-message-p-thm email-auto-outgoing)
(make-*/mail-returns-symbol-alistp-thm user-deliver-auto)
(make-*/mail-returns-same-message-thm deliver-ok)
(make-*/mail-returns-same-message-thm user-deliver-ok)
(make-*/mail-returns-same-message-thm user-deliver-abort)
(make-*/mail-returns-same-message-thm user-deliver-auto-reply)
(make-*/mail-returns-symbol-alistp-thm user-deliver)
(make-*/mail-returns-symbolp-thm user-deliver)
(make-*/mail-returns-message-p-thm user-deliver)
(make-*/mail-returns-superset-of-user-thm user-send)


(defthm thm-user-command-returns-superset-of-user
  (implies (and (symbol-alistp user)
(action-p action)
cmd
args
(equal (action-arg1 action) cmd)
(equal (action-arg2 action) args)
)
   (let ((new-user
 (user-command user action )))
   (env-subset user new-user))
)
  :hints
  (("Goal''" :use (:instance
               thm-email-auto-command-returns-superset-of-user
               (cmd (action-arg1 action))
               (args (action-arg2 action))
               (user user)))))

(defthm thm-do-actions-returns-superset-of-hosts-on-zero
        (implies (and (or rest (null rest))
       (integerp count)
       (= count 0)
                       (symbol-alistp users)
                       (symbol-alistp hosts)
       (or action (null action))
       )
                  (mv-let (new-rest new-users new-hosts)
                          (do-actions action count users hosts)
  (and (symbol-alistp new-hosts)
       (symbol-alistp new-users)
       new-rest
       )))
;                          (env-subset hosts new-hosts)))
  :rule-classes (:forward-chaining)
         :hints
         (("goal" :in-theory (enable do-actions))))


(defthm thm/mk-action->action-p
 (implies
  (and
   (member-equal type '(command init send mail deliver ok deliver-ok))
   (symbolp type)
   (and (symbolp name)
name)
   (message-p msg)
   msg
   )
  (action-p (mk-action type name msg '()))
  ))

(defthm message-p-implies-symbolp-email-host-recipient
 (implies
  (and
   (message-p msg)
   )
  (symbolp (email-host (recipient msg)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(make-*/mail-returns-symbol-alistp-thm host-send)
(make-*/mail-returns-symbol-alistp-thm user-send)

(make-d/that-returns-symbol-alistp-hosts-thm do-user-send)
(make-d/that-returns-symbol-alistp-users-thm do-user-send)
(make-d/that-returns-symbol-alistp-users-thm do-host-send)
(make-d/that-returns-symbol-alistp-hosts-thm do-send)

(make-d/that-returns-symbol-alistp-hosts-thm do-command)
(make-d/that-returns-symbol-alistp-users-thm do-command)

(make-*/mail-returns-symbolp-thm host-send)
(make-*/mail-returns-symbolp-thm user-send)

(make-d/that-returns-action-listp-if-message-p-thm do-host-send
  (("Goal" :hands-off (host-send add-user-mail-action
                       add-host-mail-action))))

(make-d/that-returns-action-listp-if-message-p-thm do-user-send
  (("Goal" :hands-off (user-send add-user-mail-action
                       add-host-mail-action))))

(make-d/that-returns-action-listp-if-message-p-thm do-send
  (("Goal" :hands-off (do-user-send do-host-send))
   ("Subgoal 3" :use thm-do-user-send-returns-action-listp)
   ("Subgoal 1" :use thm-do-host-send-returns-action-listp)))

(make-d/that-returns-action-listp-if-message-p-thm do-deliver
(("Goal" :hands-off (user-deliver add-user-mail-action))))

(make-d/that-returns-action-listp-if-message-p-thm do-mail
  (("Goal" :hands-off (do-send add-sender-mail-action
       add-user-mail-action))))

(make-d/that-returns-action-listp-thm do-action-cond
  (("Goal" :hands-off (do-init do-send do-deliver do-mail do-command)
           :use (thm-do-init-returns-action-listp
 thm-do-send-returns-action-listp
 thm-do-deliver-returns-action-listp
 thm-do-mail-returns-action-listp
 thm-do-command-returns-action-listp)))
  )

(make-d/that-returns-symbol-alistp-hosts-thm do-action-cond
 (("Goal" :hands-off (do-deliver do-send do-mail  do-init))))

(make-d/that-returns-symbol-alistp-users-thm do-action-cond
 (("Goal" :hands-off (do-deliver do-send do-mail  do-init))))


(defmacro make-d/that-do-action-cond-returns-same-as-do-thm (type)
  ‘(defthm ,(new-thm  ’action-cond-returns-same-as-do type)
     (implies
      (equal (quote ,type)
     (action-type action))
      (equal
       (do-action-cond      action rest users hosts)
       (,(catsym ’do- type) action rest users hosts)))))

(make-d/that-do-action-cond-returns-same-as-do-thm init)
(make-d/that-do-action-cond-returns-same-as-do-thm command)
(make-d/that-do-action-cond-returns-same-as-do-thm send)
(make-d/that-do-action-cond-returns-same-as-do-thm mail)
(make-d/that-do-action-cond-returns-same-as-do-thm deliver)

(defmacro make-action-type-has-message-p-thm (type)
  ‘(defthm ,(new-thm type ’action-type-has-message-p)
     (implies
      (and
       (equal (quote ,type)
      (action-type action))
       (action-p action)
       )
      (message-p (action-arg1 action)))))

(make-action-type-has-message-p-thm send)
```

```
(make-action-type-has-message-p-thm mail)
(make-action-type-has-message-p-thm deliver)
;(make-action-type-has-message-p-thm init)
;(make-action-type-has-message-p-thm command)
```

```
(defthm thm-do-actions-returns-symbol-alistp-users
  (implies
   (and (symbol-alistp users)
(symbol-alistp hosts)
(integerp count)
(action-listp actions)
)
   (mv-let (new-rest new-users new-hosts)
   (do-actions actions count users hosts)
   (symbol-alistp new-users)))
  :hints
  (("goal" :hands-off (do-action-cond))
   ("subgoal *1/5" :use (:instance thm-do-action-cond-returns-action-listp
      (action (car actions))
      (rest   (cdr actions))
      (users users)
      (hosts hosts)))
   ("subgoal *1/3'" :use (:instance
                          thm-do-action-cond-returns-symbol-alistp-hosts
    (action (car actions))
    (rest   (cdr actions))
    (users users)
    (hosts hosts)))
   ("subgoal *1/2'" :use (:instance
                         thm-do-action-cond-returns-symbol-alistp-users
    (action (car actions))
    (rest   (cdr actions))
    (users users)
    (hosts hosts)))
   ))
```

```
(defthm thm-do-actions-returns-symbol-alistp-hosts
  (implies
   (and (symbol-alistp users)
                (symbol-alistp hosts)
                (integerp count)
                (action-listp actions)
                )
           (mv-let (new-rest new-users new-hosts)
                   (do-actions actions count users hosts)
                   (symbol-alistp new-hosts)))
  :hints
  (("goal" :hands-off (do-action-cond))
   ("subgoal *1/5" :use (:instance thm-do-action-cond-returns-action-listp
      (action (car actions))
      (rest   (cdr actions))
      (users users)
      (hosts hosts)))
   ("subgoal *1/3’" :use (:instance
  thm-do-action-cond-returns-symbol-alistp-hosts
    (action (car actions))
    (rest   (cdr actions))
    (users users)
    (hosts hosts)))
   ("subgoal *1/2’" :use (:instance
  thm-do-action-cond-returns-symbol-alistp-users
    (action (car actions))
    (rest   (cdr actions))
    (users users)
    (hosts hosts)))
   ))

(defun end-or-error (sym)
  (or (equal ’end sym)
      (equal ’error sym)))

(defthm thm-do-actions-returns-end-or-error
  (implies
   (and (symbol-alistp users)
                (symbol-alistp hosts)
                (integerp count)
                (action-listp actions)
                )
           (mv-let (new-rest new-users new-hosts)
                   (do-actions actions count users hosts)
   (end-or-error new-rest)))
  :hints
  (("goal" :hands-off (do-action-cond))
   ("subgoal *1/5" :use (:instance thm-do-action-cond-returns-action-listp
      (action (car actions))
      (rest   (cdr actions))
      (users users)
      (hosts hosts)))
   ("subgoal *1/3’" :use (:instance
  thm-do-action-cond-returns-symbol-alistp-hosts
    (action (car actions))
    (rest   (cdr actions))
    (users users)
    (hosts hosts)))
   ("subgoal *1/2’" :use (:instance
  thm-do-action-cond-returns-symbol-alistp-users
    (action (car actions))
    (rest   (cdr actions))
    (users users)
    (hosts hosts)))
   ))

; Takes ~30 seconds to prove
(defthm do-action-cond-returns-action-listp
 (implies
  (and
   (action-listp rest)
   (action-p action)
   (equal (action-arg1 action) msg)
   (message-p msg)
   (equal (email-user (message-sender msg)) sender-user)
   (equal (email-host (message-sender msg)) sender-host)
   (equal (email-user (recipient msg))      recipient-user)
   (equal (email-host (recipient msg))      recipient-host)
   (equal (get-var sender-user users) sender-user-env)
   (equal (get-var sender-host hosts) sender-host-env)
   (equal (get-var recipient-user users) recipient-user-env)
   (equal (get-var recipient-host hosts) recipient-host-env)
   (not (member-equal (email-user (message-sender msg))
      (get-var 'already-answered recipient-user-env)))
   (member-equal sender-user (get-var 'users (get-var sender-host hosts)))
   (has-var sender-user    users)
   (has-var sender-host    hosts)
   (member-equal recipient-user
         (get-var 'users (get-var recipient-host hosts)))
   (has-var recipient-host hosts)
   (equal (get-var 'hostname sender-host-env) sender-host)
   (equal (get-var 'hostname recipient-host-env) recipient-host)
   (symbol-alistp users)
   (symbol-alistp hosts)
   )
  (action-listp (mv-actions (do-action-cond action rest users hosts)))
  )
 :hints (("Goal" :in-theory (enable do-action-cond)
  :hands-off (do-init do-send do-deliver do-mail do-command)
  :use (thm-do-init-returns-action-listp
thm-do-send-returns-action-listp
thm-do-deliver-returns-action-listp
thm-do-mail-returns-action-listp
thm-do-command-returns-action-listp))))
```

A.15 p-interaction.lisp

```
;;; Proof of the feature interaction:

(defthm user-deliver/auto-response-adds-sender-to-already-answered
 (implies
  (and
   (get-var 'default-response recipient-env)
   (get-var 'user             recipient-env)
   (equal (message-sender msg) sender)
   (not (member-equal sender (get-var 'already-answered recipient-env)))
   )
  (user-in-already-answered
   sender
   (mv-env (user-deliver msg recipient-env)))
  )
 )

(defthm user-deliver/auto-response-recipient-becomes-message-sender
  (implies
   (and
    (get-var 'default-response recipient-env)
    (get-var 'user             recipient-env)
    (message-p msg)
    (equal (message-sender msg) sender)
    (not (member-equal sender (get-var 'already-answered recipient-env)))
   )
    (equal
     (message-sender (mv-msg (user-deliver msg recipient-env)))
     (recipient msg))))


(defthm user-deliver/adds-already-answered-and-recv-msg
  (implies
   (and
    (get-var 'default-response recipient-env)
    (get-var 'user             recipient-env)
    (equal (message-sender msg) sender)
    (not (member-equal sender (get-var 'already-answered recipient-env)))
    (not (member-equal k '(already-answered recv-msg)))
    )
   (equal
    (get-var k (mv-env (user-deliver msg recipient-env)))
    (get-var k recipient-env)))
  :hints (("Goal" :in-theory (enable user-deliver))))
```

```
(defthm user-deliver-auto/auto-response-message-sender-becomes-recipient
  (implies
   (and
    (get-var 'default-response env)
    (get-var 'user             env)
    (equal (message-sender msg) sender)
    (not (member-equal sender (get-var 'already-answered env)))
    )
   (sender-becomes-recipient
      msg
      (mv-msg (user-deliver-auto msg env))))
  :hints (("Goal" :in-theory (enable user-deliver-auto user-deliver)
          :hands-off (sender-becomes-recipient)
  :use (email-auto-incoming/auto-response-message-sender-becomes-recipient))))

(defthm user-deliver/auto-response-message-sender-becomes-recipient
  (implies
   (and
    (get-var 'default-response env)
    (get-var 'user             env)
    (equal (message-sender msg) sender)
    (not (member-equal sender (get-var 'already-answered env)))
    )
   (sender-becomes-recipient
      msg
      (mv-msg (user-deliver msg env))))
  :hints (("Goal" :in-theory (enable user-deliver-auto user-deliver)
          :hands-off (sender-becomes-recipient)
  :use (email-auto-incoming/auto-response-message-sender-becomes-recipient))))



(defthm user-deliver/auto-response-if-not-already-answered-mail-action
  (implies
   (and
    (lookup 'default-response)             ; autoresponder enabled
    (lookup 'user)
    (equal sender (message-sender msg))
    (not (member-equal sender (lookup ’already-answered)))
    )
   (mv-let (s new-msg new-env)
   (user-deliver msg env)
   (equal s ’mail)
))
  :hints (("Goal" :in-theory (enable email-auto-incoming))))


(defthm do-deliver/auto-response-message-sender-becomes-recipient
  (implies
   (and
    (action-listp  rest)
    (action-p      action)
    (equal (action-arg1 action) msg)
    (equal (get-var (action-name action) users)
   env)
    env
    (get-var ’default-response env)
    (get-var ’user             env)
    (equal (message-sender msg) sender)
    (not (member-equal sender (get-var ’already-answered env)))
    )
   (sender-becomes-recipient
      msg
      (action-arg1
       (car (mv-actions (do-deliver action rest users hosts))))))
  :hints (("Goal" :in-theory (enable do-deliver)
          :hands-off (user-deliver get-var add-user-mail-action)
  :use (user-deliver/auto-response-message-sender-becomes-recipient))))

(defthm do-deliver/auto-response-message-sender-becomes-recipient
  (implies
   (and
    (action-listp  rest)
    (action-p      action)
    (equal (action-arg1 action) msg)
    (equal (get-var (action-name action) users)
   env)
    env
    (get-var ’default-response env)
    (get-var ’user             env)
    (equal (message-sender msg) sender)
    (not (member-equal sender (get-var ’already-answered env)))
    )
   (sender-becomes-recipient
      msg
      (action-arg1
       (car (mv-actions (do-deliver action rest users hosts))))))
  :hints (("Goal" :in-theory (enable do-action-cond)
          :hands-off (do-deliver user-deliver get-var add-user-mail-action)
  :use (user-deliver/auto-response-message-sender-becomes-recipient))))
```

Bibliography

[Asp01] Aspect oriented programming (article series). *Communcations of the ACM*, 44(10), October 2001.

[BA01] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. In *Communications of the ACM*, October 2001.

[BJMvH02] Don Batory, Clay Johnson, Bob MacDonald, and Dale von Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. *ACM Transactions on Software Engineering and Methodology*, 11(2):191–212, April 2002.

[BKM96] B. Brock, Matt Kaufmann, and J S. Moore. Acl2 theorems about commercial microprocessors. In M. Srivas and A. Camilleri, editors, *Proceedings of Formal Methods in Computer-Aided Design (FM-CAD '96)*. Springer-Verlag, November 1996.

[BLHM02] Don Batory, Roberto Lopez-Herrejon, and Jean-Phillipe Martin. Generating product-lines of product- families. In *Automated Software Engineering Conference*, 2002.

[BM96] Bob Boyer and J Moore. Mechanized formal reasoning about programs and computing machines. In R. Veroff, editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*. MIT Press, 1996.

[BO92] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[Boe81] Barry Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[CGP00a] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2000.

[CGP00b] Edward M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2000.

[CGR93] D. Craigen, S. Gerhart, and T.J. 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-V1 & NIST GCR 93-626-V2 (Order numbers: PB93-178556/AS & PB93-178564/AS), National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161, USA, 1993.

[CNea98] Paul Clements, Linda M. Northrup, and et al. A framework for software product line practice. Technical report, Software Engineering Institute, September 1998.

[FF98] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. *ACM SIGPLAN International Conference on Functional Programming*, pages 94–104, 1998.

[FN98] A. P. Felty and K. S. Namjoshi. Feature specification and automated conflict detection. *ACM Transactions on Software Engineering and Methodology*, 12(1):3–27, January 1998.

[Gra93] Paul Graham. *On Lisp*. Prentice Hall, 1993.

[Gri00] Martin Griss. Implementing product-line features by composing component aspects. In *First International Software Product-Line Conference*, August 2000.

[Hal98a] Robert J. Hall. Feature combination and interaction detection via foreground/backgro und models. In *Feature Interactions in Telecommunications Syst ems*. IOS Press, 1998.

[Hal98b] Robert J. Hall. Feature combination and interaction detection via foreground/background models. In *Proc. 5th International Workshop on Feature Interactions in Telecommunications and Software Systems*. IOS Press, 1998.

[Hal00] Robert J. Hall. Feature interactions in electronic mail. In *Proc. 6th International Workshop on Feature Interactions in Telecommunications and Software Systems*. IOS Press, 2000.

[HC01] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

[KK98] Dirk O. Keck and Paul J. Kuehn. The feature and service interaction problem in telecommunications systems: A survey. *IEEE Transactions on Software Engineering*, 24(10):779–796, October 1998.

[KM97] Matt Kaufmann and J. Strother Moore. An industrial strength theorem prover for a logic based on common lisp. *Software Engineering*, 23(4):203–213, 1997.

[KMM00] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000.

[LKF02a] Harry Li, Shriram Krishnamurthi, and Kathi Fisler. Interfaces for modular feature verification. In *International Conference on Automated Software Engineering*, September 2002.

[LKF02b] Harry Li, Shriram Krishnamurthi, and Kathi Fisler. Verifying cross-cutting features as open systems. In *International Conference on Foundations of Software Engineering*, November 2002.

[LLM99] Karl Lieberherr, David Lorenz, and Mira Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, March 1999.

[Mey91] Bertrand Meyer. *Eiffel: The Language.* Prentice Hall PTR, October 1991. ISBN: 0132479257.

[MLK] J Strother Moore, Tom Lynch, and Matt Kaufmann. A mechanically checked proof of the correctness of the kernel of the amd5k86 floating-point division algorithm.

[Moo99] J. Moore. Proving theorems about java-like byte code. In E.-R. Olderog and B. Steffen, editors, *Correct System Design – Issues, Methods and Perspectives.* LNCS, 1999.

[OT99] H. Ossher and P. Tarr. Multi-dimensional separation of concerns in hyperspace. Technical Report RC 21452(96717), IBM, April 1999.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Par79] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–38, March 1979.

[Pnu85] Amir Pnueli. In transiton from global to modular temporal reasoning about programs. In K. Apt, editor, *Logics and Models of Concurrent Systems*, volume F-13 of *NATO Advanced Summer Institutes*, pages 123–144. Springer-Verlag, 1985.

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *Proceedings of ECOOP'97*. Springer-LNCS, 1997.

[Pre02] Christian Prehofer. Feature interactions in statechart diagrams or graphical composition of components. In *Second International Workshop on Aspect-Oriented Modeling with UML*, September 2002.

[Ray03] Eric Steven Raymond. *The Art of Unix Programming.* Thyrsus Enterprises, 2003.

[RF00] David M. Russinoff and Arthur Flatau. Rtl verification: A floating-point multiplier. In Matt Kaufmann, Panagiotis Manolios, and J S Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 13. Kluwer, 2000.

[Rus99] D.M. Russinoff. A mechanically checked proof of correctness of the amd-k5 floating point square root microcode. *Formal Methods in System Design*, 14:75–125, 1999).

[SE98] N. F. Schneidewind and C. Ebert. Preserve or redesign legacy systems? *IEEE Software*, 15(4):14–17, July/Aug 1998.

[Uni] Unix Programmer's Manual. *vacation(1).*