A Case Study in Using ACL2 for Feature-Oriented Verification

Kathi Fisler and Brian Roberts
WPI Department of Computer Science
kfisler@cs.wpi.edu

November 8, 2004

Abstract

Feature-oriented software systems are built from components that encapsulate cohesive end-user features. Feature-oriented components are interesting because they cross-cut the system architecture, capturing behavior fragments from several system entities. Features support a plug-and-play style of software construction, in which several systems can be constructed from the same core set of features. This approach requires novel forms of formal verification that support modular reasoning about feature-based components. This paper presents a case study on modeling and verifying a feature-oriented email system in ACL2. The main goal of the study is to identify classes of theorems that should be proven about individual features so that properties about compositions of features can be derived from the theorems on the individual features. A secondary goal is to evaluate ACL2 as a modeling and verification framework for feature-oriented systems. We present our feature model and insights into verification, then discuss the strengths and weaknesses of ACL2 in this problem domain.

1 Introduction

Feature-oriented design is an increasingly common software development paradigm. This approach recognizes that many software systems consist primarily of user-specified *features*, where a feature is a product characteristic that allows customers to distinguish between products. Telecommunications systems are a popular example, offering features such as voice-mail, message forwarding, and call-waiting. Domains rich in features tend to lead to *product-lines* of related systems built from combinations or permutations of features from a common set [5, 9].

Feature-oriented software design offers unique challenges for verification. A product line can yield an exponential number of products relative to the number of features, making it infeasible to verify each new product from scratch. This demands a verification methodology in which properties specific to individual features are verified against those features and a lightweight analysis checks that feature-specific properties remain valid when features are composed into products. While this sounds like classical modular verification, features introduce three challenges to this standard problem. First, the composition-time analysis require reasoning about *open systems* because features may refer to shared propositions. Second, two or more features can collectively yield behavior that violates the expected behavior of a subset of those features, and *this is often desirable*. Consider a voice-mail feature with the property that it activates message recording if a call is unanswered after four rings. This property could be violated if the feature is included in a system that also provides call forwarding (if the call is forwarded before the voice-mail system is activated). This *feature interaction* problem is perhaps the most interesting technical issue in feature-oriented design (and one that is not always amenable to formal analysis [11]). Third, features *cross-cut* software architecture, meaning that a feature contributes code to several components in the underlying system. This means that the module (feature) that we wish to analyze in isolation is less cohesive at an implementation level than a conventional module (though more cohesive from a behavioral perspective), which violates some of the assumptions underlying traditional modular verification.

We are interested in developing verification strategies for feature-oriented systems. In prior work, Fisler *et al.* developed a modular approach to feature verification based on model-checking [4, 8, 12]. Because

finite state machines and temporal logic are not always appropriate formalisms, we are also interested in understanding how to do modular feature verification in other frameworks. This paper considers this problem in the context of ACL2, presenting a case study in modeling and modularly verifying a simple email application. Our long-term goal is to figure out how to effectively model and modularly verify feature-oriented systems in ACL2: this involves figuring out what the system architecture should look like to enable plug-and-play construction of multiple products from sets of features and what sorts of intermediate lemmas we need to prove about each feature to enable lightweight modular verification of products. Ideally, the product-level verification should also warn of potential feature interactions, despite the incompleteness of reasoning in ACL2. Identifying the lemmas is in some sense the most important aspect of this project, as they should carry over to a variety of verification frameworks in which models of features are given at the level of code (as opposed to state machine abstractions).

The work reported here is in its early stages, and has been done by two advanced-beginner-level users of ACL2. One of our goals in presenting this work at the ACL2 workshop is to gather feedback from the ACL2 community regarding approaches and techniques that might better realize our project. In turn, we present our initial findings about the strengths and weaknesses of ACL2 for this problem domain. We also hope that the ACL2 community will find this paper useful as an introduction to a growing trend in software development that will require support from the verification community in the long term.

2 The Case Study Example: An Email System

This paper uses a running example of an email product line that is originally due to Bob Hall of AT&T Labs [10]. The available features in Hall's example are basic mail delivery, digital signatures, forwarding, anonymous remailing, encryption, decryption, signature verification, auto-reply, filtering (based on sender's hostname), and mail hosting. Our study uses a representative subset of Hall's features that give rise to formalizable feature interactions. The *host* feature sends a "postmaster response" when an attempt is made to send a message to an unknown user. The *auto-response* feature is like Unix *vacation*: it sends an automatically-generated response to senders the first time they send a message to a user. The *encryption* and *decryption* features encrypt and decrypt mail, respectively. We use host for its simplicity, auto-response to represent features that do not modify the message being sent but can trigger other actions to be performed (mailing new messages), and *encryption* and *decryption* as features that affect incoming and outgoing messages. The properties of and interactions between these features that are relevant to this study are detailed as needed in the rest of the paper.

Architecturally, an email system contains several components: a database of information about individual users (such as their encryption keys, mail aliases, and auto-response message), a series of processes to run on outgoing messages (such as deciding whether to encrypt them) and a series of processes to run on incoming messages (such as deciding whether to send an auto-reply to them). A full-blown email application would also contain queues of incoming and outgoing messages; as the queues complicate verification without adding insight to feature verification, we currently assume that only one message is active in the system at a time. We implement an email system as a simulator that takes a list of valid commands from users (such as "send a message to Fred" or "change my auto-response message") and executes each one in turn. The result of executing a command could be a change to the user database, the issuance of a new command to continue processing a message, the delivery of a message to a user, or an error message if the command fails.

Individual features can contribute code to multiple components of an email system: an encryption feature, for example, introduces commands for enabling encryption and setting encryption keys, as well as the process for actually encrypting an outgoing message. This illustrates how features cross-cut conventional system architecture; Figure 1 shows this visually. Although having features (components) cross-cut the architecture breaks the homomorphism between components at the design and architectural levels, it may prove a benefit for verification: properties and features are both behavioral entities, arising from the perspective of end-users; as a result, feature modules often align more naturally with properties than conventional modules. Fortunately, features tend to compose sequentially rather than in parallel, so most inter-component communication occurs between fragments arising from the same feature; this vastly simplies verification.

Implementation Components

		Command loop	Database	Incoming	Outgoing
Features	auto-reply	set-msg, enable	response	reply code	
	encryption	set-key, enable	key	check encrypted	encrypt msg

Figure 1: Features as modules (shown horizontally) provide code fragments for system components (shown vertically).

3 Modeling and Building Products

The preceding overview of feature-oriented systems identifies a number of requirements for our models of features and products in ACL2. Specifically, the model must support:

- describing a feature independently from other features,
- describing the pieces of a feature that belong in each component independently of one another,
- easily changing the set and order of features that comprise a product,
- proving properties of individual features, and
- proving properties of compositions of features using theorems about individual features as lemmas.

Not surprisingly, we model features and the simulator as collections of functions. Our model relies on four main data structures. *Messages* are lists capturing the sender, recipient, message headers, and message contents. *Actions* capture commands to be executed; they consist of lists indicating the action and the data required to perform it. For example, the action to mail a message is a list of the form `('mail sender message)`, where the message captures the recipient. The *user environment* captures users' personal preferences (including address books and whether forwarding is set). The *host environment* stores information about which users have accounts on each host. To facilitate verification, the host and user environments also store the messages that are sent or received to each user and host.

The rest of the section describes the functions that we define for each feature and the mechanisms we use to build products from sets of features. Section 4 discusses verification.

3.1 Modeling Individual Features

Consider an auto-response feature: when a user enables this feature, it tracks which senders have already received an automated reply from the user and sends an automated reply when the user receives a message from a new sender. Adding this feature requires three kinds of extensions to an email system:

1. new actions that users can perform (for enabling auto-response and setting the contents of the generated message),
2. new types of information that must be stored in the user and host environments (the stock message to send and which users have received replies), and
3. new processing on incoming messages (check the list of previous senders and send an auto-reply if necessary).

Other features, such as encryption, could also introduce processing on outgoing messages.

These four kinds of changes drive our model for features. An individual feature is modeled as a set of four essential functions, one for each type of modification. The essential functions may use any number of

```
;; augment user environment with initialized fields needed for feature
(defun email-auto-init (env)
   (set-var 'already-answered '()
    (set-var 'user '()
     (set-var 'default-response '() env))))

;; code to execute new commands introduced by this feature
(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)))

;; just returns a comment since this feature does not process outgoing messages
(defun email-auto-outgoing (msg env)
  (begin (cw "  [email-auto-outgoing: ~x0]~%" (get-var 'user env))
         (act comment "[Outgoing events not handled]")))

;; the heart of the auto-response feature, which works on incoming
;; messages.  The act function is the constructor for actions.
(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 2: ACL2 source code of auto-response feature.

auxiliary functions, but the system model will only directly invoke the essential functions. The following table summarizes the essential functions. We indicate essential functions through a naming convention: a feature-specific prefix (such as `email-auto`) combined with the suffix listed in the table.

Function/Suffix	Description
`-init`	Initializes system data structures for the feature
`-command`	Introduces new commands related to the feature
`-outgoing`	Transforms messages sent from 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 2 shows the ACL2 model of the auto-response feature. Following the naming convention, 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 four functions go into a single file for the corresponding feature. We chose not to package the feature functions into ACL2 books because book certification introduced a more complicated step than seemed necessary for the exploratory nature of this project.

3.2 Modeling Products

An email product (simulator) consists of a base system that performs simple mail delivery and a set of hooks for adding optional features. Through the hooks, the base system must invoke each feature's essential functions to initialize environments, dispatch commands, and process all incoming and outgoing messages. Figure 3 shows the core of the simulator; the full code appears elsewhere [16][1]. The main function of a product is `simulate-network`, which invokes `do-actions`, the main processing loop of the simulator. The function `do-action-cond` dispatchs to one of several functions (`do-init`, `do-send`, `do-deliver`, `do-command`, and `do-mail`) depending on the type of the action to perform. The `do-mail` function implements basic mail delivery; the remaining four functions provide the hooks for inserting features (described in the next section). Figure 4 summarizes the call graph of the simulator, which shows where the essential functions fit into the system code.

3.3 Building Products from Features

To insert features into a simulator, we need to specify which features to include, then ensure that the hook functions invoke the essential functions for each feature. Figure 5 shows a representative fragment of the product-customization code. The `*features-present*` constant defines which features should be included or enabled in a product (in the figure, auto response and encryption). The `user-init` function provides the hook for the `-init` functions; it is called from `do-init` in Figure 3. The `fif` construct is a macro (short for "feature-if") that includes the code for a feature if the feature is enabled. `user-init` calls each enabled feature's initialization function in turn with the user environment as its argument. Each `-init` function will return a (possibly) modified environment, which `user-init` passes on as the argument to the next feature.

Using `fif` and `*features-present*`, we can easily customize which of the available features we include in a particular product. This approach is not as modular as we would ideally like, since it does involve modifying the core infrastructure code whenever we develop a new features. The usual solution to this problem in functional languages uses higher-order functions, which ACL2 does not support. Our chosen approach strikes a reasonable balance between quick customization of products and separation of features. We discuss this issue in more detail in Section 5.

The `do-command` hook function is similar in format and function to `do-init`. The feature functions for incoming and outgoing messages are invoked from the `do-deliver` and `do-send` functions called in `do-action-cond` in Figure 3. Function `do-send` pipes the message through each feature's `-outgoing` function; these functions may modify the message (to encrypt it, for example). After all features have processed the message, `do-send` calls appropriate functions to send the message to the specified host. On the message

[1] In order to keep the paper self-contained, some uses of simple functions and macros from the full code have been inlined.

```
;; parses actions into internal representation and initiates action processing
(defun simulate-network (actions)
  (do-actions (parse-actions actions) 20 *users* *hosts*))

;; runs each action in turn, returning updated user and host environments
;; the count variable is used to prove termination of the function
(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)))

;; for any action other than basic mail delivery, runs the command and
;; returns the updated user and host environments (and remaining actions).
;; Features are added to the system by extending the definitions of the helper
;; functions called in the cond cases of this function.
(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))))))
```

Figure 3: The core simulator code, modeling a system that contains no optional features. As features are added to the system, the `do-init`, `do-command`, `do-send` and `do-deliver` functions get extended to include the code for those features (see Section 3.3).

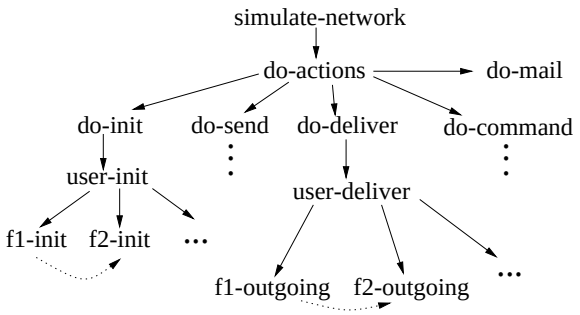

Figure 4: The call graph for the simulator, showing where the essential feature functions are invoked. Solid lines indicate calls from one function to another. Dashed lines indicate that the user or host environments resulting from one function are passed to another.

```
(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 5: User initialization source code over three possible features. `let-seq` is a macro that sequentially binds the variable in its first argument to the values of the remaining expressions, then returns the value of the last expression. In this example, it is used to accumulate changes to the data structure for a user over the individual feature initializations.

delivery side, `do-deliver` is similar to `do-send`. It passes the message through the features that could affect the message, and then delivers the message. We omit the code for these functions as it does not shed additional insight on our model or approach.

4 Verifying the System

Ideally, a modular verification technique for feature-oriented systems should allow a designer to verify properties of individual features, then use a lightweight, modular technique to confirm that feature-specific properties are not violated when features are composed into a product. In the context of ACL2, we interpret "lightweight" to mean that the composition-time proofs require little to no human interaction with the prover. As an example, if we have a product containing auto-response that satisfies the auto-response properties, and then we add an encyryption feature, we need to prove that the extended product still satisfies the auto-response properties; if our methodology is truly lightweight, we shouldn't need to prove unanticipated theorems about the functions that implement encryption. Even in the context of anticipated theorems, we don't want to have to prove theorems specifically about auto-response on the encryption feature because this would lead to a combinatorial explosion of theorems across all the features that could comprise a product. One of our goals with this case study is to understand what form these general theorems might take, if they exist at all. Furthermore, we are interested in whether ACL2 failing to prove a property theorem from the general theorems likely suggests a feature interaction (as opposed to a missing auxiliary lemma).

Fisler's prior work on using CTL model checking for modular feature verification partially verifies a property against a feature to automatically generate sufficient conditions under which the property would hold of the feature; the conditions are captured as CTL formulas. When features are composed into a product, another series of automated checks (akin to model checking and three-valued propositional checks) discharges the generated constraints; failure to discharge a constraint indicates a feature interaction. This approach relies on the decidability of model checking, in that it detects feature interactions through failure of the checks on constraints.

Reasoning about features and their composition in our ACL2 model is more subtle when reasoning about both individual features and feature compositions. The subtleties in proving properties about individual features arise in part because our model does not make individual features truly independent of the overall system. While a feature is defined through the four essential functions described in Section 3.1, those four functions only interact through the code for the core infrastructure (which calls the `-init` function through `do-init`, then the `-command` function through `do-command`, and so on). This makes it difficult to reason about an individual feature purely at the level of its essential functions, without including the core infrastructure. The subtleties in reasoning about feature compositions arise largely from the undecidability of ACL2.

4.1 Reasoning about Individual Features

To verify a property against an individual feature, we first form a simple product out of the feature and the core infrastructure (with no other features); we then prove a series of theorems that lift the property to the level of the `simulate-network` function in Figure 3. As an example, consider the auto-response feature, whose main property is that it should reply once and only once to all messages from active users of the system, but not including automated messages such as the postmaster. One key theorem towards this property states that sending a message to a user who has enabled auto-response adds a sender to the user's list of senders who have already received answers. This theorem appears as the first expression in Figure 6. The assumptions on the theorem are that the auto-response feature is enabled (`'default-response` is non-nil in the environment), the environment is a valid user environment (`'user` is non-nil), the sender of the message `msg` is equal to `sender`, and the sender has not already received an auto-response message. The consequence of the implication states that the user environment resulting from running `email-auto-incoming` includes the sender in the already-answered list.

The remaining two theorems in Figure 6 lift the theorem on `email-auto-incoming` to the functions that invoke `email-auto-incoming`, namely `user-deliver` (called directly from `do-deliver` in `simulate-network`) and `simulate-network`. The theorem on `simulate-network` confirms that the property holds in a product with only this feature; ACL2 needs the `user-deliver` theorem to find the proof for the `simulate-network` theorem.

Were we to extend the product with an additional feature, we would want to prove that the properties proven of the auto-response feature continue to hold in the new product. Obviously, we could simply re-verify the theorems; if the new feature has no impact on auto-response, the theorem proved of the smaller product should (hopefully) go through without additional human guidance to ACL2. This is not a truly modular approach, however, because ACL2 will re-verify some theorems rather than use the results of the previous verification: specifically, ACL2 needs to re-verify the `user-deliver` and `simulate-network` theorems because adding the new feature requires editing `user-deliver` (which `simulate-network` calls). In theory, the verification of the `email-auto-incoming` theorem need not be done again, as that code has not been edited. Putting the feature functions into a book could prevent this re-verification, but this approach would not scale to properties that require interaction between essential functions through the core infrastructure. Nonetheless, this approach would allow for some modular verification. Rather than discuss modular verification and our motivation for it further on this example, the next section explores it in the more interesting context of feature interaction.

4.2 Reasoning Modularly about Feature Interactions

The auto-response and host features yield a feature interaction when composed. When a message is sent to an invalid user on a host, the host feature sends a "user unknown" message from the `postmaster` account.

```
(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 6: Hierarchy of theorems to prove an auto-response property. The first part of the name of each theorem indicates to which function the theorem is lifting the property. The main difference between these theorems lies in the assumptions, which get more restrictive the closer the theorem gets to being about `simulate-network`.

```
(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)
                (member vacation *vacation-space*)
                (equal (get-var sender-name users) sender-env)
                (get-var default-response sender-env)
                (get-var 'user 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 new-users))
                       vacation))))
```

Figure 7: Theorem capturing the feature interaction between auto-response and message hosting.

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

Detecting this interaction is difficult because it does not violate properties about either the host or auto-response features individually. Detecting this interaction in a theorem-proving context is complicated further because theorem proving is generally undecidable. Even if we had anticipated the potential interaction and encoded its absence as a theorem, it would be difficult to determine whether the theorem fails because there is an interaction or because we failed to identify sufficient supporting lemmas.

In this case study, we are mainly concerned with figuring out what properties of the individual features could be used to signal the interaction. We therefore capture the interaction itself (rather than its absence) as an ACL2 theorem, then determine what intermediate lemmas we would need about each feature in isolation so that we could prove the interaction theorem without reasoning about an entire product containing both features. Intuitively, the theorem capturing the interaction says that given a valid user name and auto-response (vacation) message, 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; Figure 7 shows the code. While this approach of verifying that interactions exist is not reasonable in the long-term, it is nonetheless useful for developing a methodology for modular theorem proving about feature-oriented systems.

The interaction lemma follows from theorems about the behavior of the host and auto-response features. Specifically, the theorem depends on three theorems: 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, 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, and 3) if the user in the recipient field in not known on a host, then the `postmaster` will send an "unknown user" reply to the message sender. The first two capture the changes in environment and action resulting from auto-response, respectively. Although these theorems reflect core properties of the individual features, they would not signal an interaction in the absence of the interaction theorem.

Proving the interaction theorem becomes more interesting when we require that reasoning to be modular. The proof that the interaction theorem follows from the three feature-specific theorems uses the bodies of the functions that define the features. Truly modular reasoning should not need the feature implementations. Figuring out how to do truly modular reasoning is important because we want to determine whether we

```
; email-auto-outgoing returns a valid message when given a valid message
(defthm thm-email-auto-outgoing-returns-message-p
    (implies (message-p msg)
             (message-p (mv-nth 1 (email-auto-outgoing msg env)))))
```

Figure 8: Theorem characterizing constraints on functional outputs.

```
(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)
                 (has-var 'already-answered new-env)
                 (has-var 'user new-env)
                 (has-var 'default-response new-env)))))
```

Figure 9: Theorem characterizing which attributes are changed during function call.

can identify logically sufficient intermediate lemmas for modular verification, independently of ACL2. To simulate modular reasoning, we exploited ACL2's disabling feature and *hands-off* hints to prohibit expansion of the feature's functions at verification time. ACL2 now needs additional lemmas to discharge the interaction theorem, as we describe in the next section.

4.3 Characterizing the Intermediate Lemmas

The specific intermediate lemmas needed to prove interaction theorems modularly are not interesting in this paper, but their nature is instructive for our goal of developing a methodology for modular feature verification. Our work identified four general types of supporting lemmas:

1. *Input/output:* Constraints on the types or format of the functional inputs (arguments) and outputs (return values). Figure 8 shows an example.

2. *Changes:* Which attributes *might* and definitely do not change in the environment after executing the feature. Figure 9 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 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 10 shows an example. The theorem `auto-response-if-not-already-answered-mail-action` proves that if auto-response 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: if the sender is already in the `'already-answered` list, then no auto-response message is sent.

4. *Lifting:* Theorems needed to raise theorems proven about individual features to theorems about the `simulate-network` and its immediate helper functions.

```
;; 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))))

;; 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 10: Theorems characterizing messages resulting from the auto-response feature.

The first three kinds are proven against the essential functions for the individual features. The lifting theorems naturally involve the enclosing definitions referenced in the theorems themselves.

The changes and dependent changes lemmas most closely resemble the constraints that Fisler *et al.*'s model-checking technique generates to preserve properties of features. The main difference is that the ACL2 theorems frame changes in the context of environments, which is a data structure more appropriate to ACL2's functional models than to state machines. The input/output theorems are new to the ACL2 context; they were unnecessary in a model-checking context because all data was propositional, lacking both type and structure (messages there were captured as individual propositions for each message attribute). The lifting theorems are also novel to the ACL2 context, arising from ACL2's richer modelling framework as compared to state machines.

4.4 Verification Effort

The following table shows the distribution of theorems needed to prove the interaction theorem between the host and auto-response features. The feature-level theorems were generally easy to produce, and required less user time and effort to create and prove. The majority of the human effort was expended at the lifting theorems, particularly because of the number of theorems at that level. We used ACL2 version 2.5 for this project.

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

In our experience, the dependent-changes and lifter theorems required the most human intervention, often requiring hints to the theorem-prover or additional helper lemmas before finally proving. The input/output and changes theorems in the system generally went through with little additional intervention. As our experience with ACL2 and the problem domain increased, lifting became easier because the effort needed to lift different properties followed predictable patterns. Additional case studies would be needed before we could predict the human time needed to use our approach to add (and verify) new products to the system.

5 Perspective on Using ACL2 for Modular Feature Verification

Our main goals for this case study were twofold: first, we wanted to explore how difficult it would be to model and verify feature-oriented systems in ACL2; second, we wanted to understand what sort of lemmas are needed about individual features to enable modular verification and detection of certain kinds of feature interactions. This section reflects on our lessons learned toward both goals.

ACL2 has both strengths and weaknesses for modeling feature-oriented systems for modular verification. The procedural style of models in ACL2 naturally captures the high-level control flow in feature oriented systems because features generally interact sequentially, rather than in parallel. Features, however, cross-cut system entities. As such, a feature is not one function but several, the calls to which occur within some core infrastructure code. Composing features into a product therefore required that we modify the core infrastructure code in order to invoke all the necessary functions. ACL2 macros were very useful in managing the code modifications, but this approach is not modular due to our need to modify the infrastructure code. A standard modular model for feature oriented systems in functional languages passes functions that extend a system as arguments to the appropriate parts of the infrastructure code. ACL2's restriction to first-order functions fails to support this model. In addition, the cross-cutting functions for a single feature often execute concurrently, rather than sequentially as in this case study; we currently ignore this aspect of realistic feature-oriented systems.

A related consequence of connecting the essential functions for a single feature through the core infrastructure code is that ACL2 books fail to be appropriate for modularity. Ideally, we would want to create a book for each feature, putting all the code and theorems needed for modular reasoning into the book. This approach would work for theorems that involve only one essential function, but not for theorems involving the interaction between essential functions or for the lifting theorems. Putting the code for the core infrastructure in each book would result in multiple copies of that code; theorems would not be reusable across these copies. Our best approach to modularity thus far has been to disable traversal of features that had already been verified using ACL2's `hands-off` hints. Without `hands-off`, which is outside the scope of the core ACL2 logic, it is not clear how we could create a suitable modular verification framework within ACL2.

One might argue that whether proofs succeed with or without `hands-off` is irrelevant, as long as the proofs go through automatically. We believe, however, that being able to prove theorems at composition time while using `hands-off` hints provides a metric for assessing the lightweightness of our compositional reasoning approach: proofs that don't go through with `hands-off` hints indicate necessary feature-specific theorems that our methodology missed. We must conduct further case studies to determine whether this metric is meaningful in practice.

Fisler *et al.*'s modular model checking methodology for features was able to automatically generate sufficient constraints for preserving properties of each feature. These constraints serve the analogous role to the intermediate lemmas used for the interaction theorem in this case study. Automatic constraint generation is a significant advantage, as assumptions can be notoriously hard to get right in assume-guarantee reasoning. A natural question, then, is whether we might be able to automate generation of the intermediate lemmas in ACL2. As a first level of automated support for lemma generation, we exploited ACL2 macros to create theorems parameterized by details from the actual features. For example, Figure 11 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 the name of the function to which to apply the theorem and (optionally) any hints that should be provided to the theorem prover to aid in proving the theorem. This technique vastly simplified our task of maintaining and updating the system, and kept similar theorems consistent across features.

Our macro-based approach, while useful, still requires the product designer to decide which lemmas to create for each feature. In the longer term, we would prefer to push this automation even further, and use some formal analysis of the feature models to determine which theorems to generate in the first place. Furthermore, we would like to be able to predict which classes of lemmas ACL2 could generally discharge without guidance (beyond the hints that we build into the macros). Input/output style theorems mostly capture type annotations, and could be generated from a front-end feature modeling language with type annotations. The changes theorems, which characterize which variables definitely or sometimes change, could at least be approximated with standard flow analyses. We could probably generate skeletons of the lifting

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

functions, but those require additional assumptions based on the system's control flow that might be hard to generate accurately in practice. The dependent change theorems talk about the feature implementations, and are not good candidates for automation. A feature-oriented verification environment built around ACL2 could therefore provide some nontrivial designer support, but would still be subject to some of the usual complexities that arise in theorem proving. Naturally, we need to perform additional case studies to refine our views on how much we can automate in this domain.

6 Conclusion

The software engineering community is exploring many notions of programming with cross-cutting components. Work on aspect-oriented programming [1], mixin layers [2], units [7], subject-oriented programming [14] and others [3, 13, 15] all revolve around creating design-level components that span several implementation-level entities. This design trend raises new challenges for verification. In the context of features, the vast number of ways in which features can be composed into products mandates that this verification be somewhat modular, so that the majority of verification effort is amortized over several products; composition-time checks must be lightweight.

This paper describes an early effort at using ACL2 as a modeling and verification framework for feature-oriented systems. Using a case study of a feature-rich email system, we proposed a model of individual features and showed how to compose features into products. We showed how to verify properties against individual features, and identified four classes of lemmas that we need about individual features in order to reason modularly about their compositions. We demonstrated that these lemmas allow ACL2 to modularly prove that a particular email product contains an undesirable interaction between two features (an instance of so-called *feature interaction*). This gives us high confidence that our current findings are on the right track towards a feasible methodology for reasoning about feature-oriented systems through ACL2.

Our models are inspired by Hall's LISP-based descriptions of feature-oriented systems [10]. Hall manually searched his models for feature interactions, rather than use a verification-based approach as described here. We are developing more automated techniques that can modularly detect these interactions from formal descriptions of the behavior that they violate. Felty and Namjoshi [6] use temporal logic as a foundation for detecting feature interactions between feature specifications, but do not consider verifying properties of features or modular detection of interactions. We are not aware of any other efforts to detect feature interactions modularly through theorem proving.

Our work to date has made several simplifying assumptions that we need to relax in future work. Perhaps the most disconcerting is that modularly detecting a feature interaction currently involves proving a theorem stating that the interaction exists. In reality, feature interactions are *unexpected* behaviors; we would prefer to detect interactions when a composition of features fails to preserve a property proven of one of the features. Even this is challenging in an undecidable logic such as ACL2's, since failure to find a proof does not guarantee that a theorem is false. We are hopeful, however, that many feature-oriented systems will have a certain level of regularity that makes feature interaction detection feasible in practice. For example, the features in our current email suite all follow a pattern of making adjustments to the host and

user environments; the theorems that we prove modularly at composition time essentially confirm that the sequence of features respects the environment setting that each feature requires. These theorems appear to involve mostly straightforward rewriting and reduction; as such, we are hopeful that we can identify a sufficiently rich set of lemmas to prove about individual features so that composition theorems have high likelihood of succeeding without human guidance to the prover. Alternatively, we may be able to use hints more effectively to help trigger failure in the face of feature interaction. If we can achieve this, failure to prove a theorem would strongly suggest a feature interaction. This hypothesis is highly speculative, and will require additional case studies to confirm or deny.

Our work indicates that ACL2 has both strengths and weaknesses for modular feature verification. The `hands-off` hints are perhaps ACL2's greatest asset, as they allow us to simulate modular verification by disabling expansion of a feature's definition. ACL2 macros are also extremely helpful from a code management perspective; we use them extensively to configure different products out of features and to generate standard templates of theorems to prove about individual features. Most of ACL2's disadvantages reflect the standard limitations to automation associated with theorem proving. One ACL2-specific issue was that having only first-order functions meant that we had to manually inject features into a product, rather than externally linking features together through higher-order functions.

We close with a series of questions for more experienced ACL2 users:

1. How might we create a more genuinely modular model of a feature-oriented system? Even assuming we could write feature models such that the interaction between the core functions doesn't rely on the infrastructure code, is there a good way to "inject" a feature into a product without modifying the core infrastructure code?

2. How might books be most useful in this project? Our understanding is that books are essentially libraries of code plus theorems; including a book would still give the prover access to the feature implementations during verification, so we would still need to use hints (`hands-off` or `disable`) to enforce modular verification. Are we missing a characteristic of books that might be helpful here?

3. Has anyone had luck with determining sets of intermediate lemmas that make a class of theorems highly likely to go through? Which sorts of hints might we explore to get ACL2 to distinguish feature interactions from theorems that lack sufficient supporting theorems? Are we overly optimistic that regularity across features could be sufficient to yield high probability that theorems about property preservation would go through if they were indeed true?

References

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

[2] 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.

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

[4] Colin Blundell, Kathi Fisler, Shriram Krishnamurthi, and Pascal Van Hentenryck. Parameterized interfaces for open system verification of product lines. In *IEEE International Symposium on Automated Software Engineering*, 2004.

[5] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[6] Amy P. Felty and Kedar S. Namjoshi. Feature specification and automated conflict detection. *ACM Transactions on Software Engineering Methodology*, 12(1):3–27, 2003.

[7] 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.

[8] Kathi Fisler and Shriram Krishnamurthi. Modular verification of collaboration-based software designs. In *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 152–163, September 2001.

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

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

[11] 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.

[12] Harry Li, Shriram Krishnamurthi, and Kathi Fisler. Modular verification of open features through three-valued model checking. *Journal of Automated Software Engineering*, To appear.

[13] 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.

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

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

[16] Brian Roberts. Modular detection of feature interactions through theorem proving: A case study. Master's thesis, WPI Department of Computer Science, August 2003.