A Case Study in Using ACL2 for Feature-Oriented Verification

Kathi Fisler and Brian Roberts
WPI Computer Science
Configurations of Features

- verify signature
- encrypt
- addressbook
- decrypt
- filter
- mailhost
- auto-respond
- signing
- forward
- remail

[Hall, 2000]
## Feature-Oriented Design

Modules encapsulate features, not objects

<table>
<thead>
<tr>
<th>Features</th>
<th>Command loop</th>
<th>User Pref Database</th>
<th>Incoming Messages</th>
<th>Outgoing Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto-reply</td>
<td>set-msg, enable</td>
<td>response</td>
<td>check/send reply</td>
<td></td>
</tr>
<tr>
<td>encryption</td>
<td>set-key, enable</td>
<td>key</td>
<td>check encrypted</td>
<td>encrypt message</td>
</tr>
</tbody>
</table>

Components
Feature-Rich Systems

- Telecommunications industry
- NASA’s next-generation software base
- Symbian
- Aspects

Still greatly lacking in verification tools
Verification Challenges

- Exponential number of possible products!
  - verify individual features once
  - verify compositions cheaply

- Feature interactions
  - does voice mail always engage after 4 rings?

- Features can share data
The Case Study

- Model an email system with four features
  - Host/postmaster (report unknown users)
  - Auto-response (aka *vacation*)
  - Encryption
  - Decryption

- Determine lemmas to modularly
  - prove properties of individual features
  - confirm properties and detect interactions
A Basic Email System

simulate-network (hostenv, userenv, actions)

↓

do-actions (...) → do-mail
## Modeling Features

<table>
<thead>
<tr>
<th></th>
<th>Command loop</th>
<th>User Pref Database</th>
<th>Incoming Messages</th>
<th>Outgoing Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>auto-reply</strong></td>
<td>set-msg, enable</td>
<td>response</td>
<td>check/send reply</td>
<td></td>
</tr>
<tr>
<td><strong>encryption</strong></td>
<td>set-key, enable</td>
<td>key</td>
<td>check encrypted</td>
<td>encrypt message</td>
</tr>
</tbody>
</table>

One function for each extension to the system

- add new actions
- add user info
- add processing on incoming messages
- add processing on outgoing messages
A Basic Email System

simulate-network (hostenv, userenv, actions)

  → do-actions (…)

    → do-mail

    ↓

  do-init

  ↓

  email-auto-init

  ↓

  …

do-command

  ↓

  email-auto-incoming

  ↓

  …

do-send

  ↓

  host-incoming

  ↓

  …

do-deliver
(defconst *features-present* '(auto encrypt))

(defund do-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)))
Verifying Features

If user has auto-response enabled and sender not in prev-recip list, send message

- Needs -init and -incoming functions

- Verify against product containing base system and auto-response feature
  - theorem refers to simulate-network
  - not really modular
Lightweight Product Verification

Add host to product with auto-response:
prove auto-response property still holds

- build (new) product including host feature
- prove \textit{simulate-network} theorem again

Lightweight means proof shouldn’t require unanticipated lemmas

Ideally warn of likely feature interactions
Detecting Feature Interactions

- Sample interaction: Auto-reply message sent to postmaster
- Often violates no properties of features
- Incompleteness makes more difficult
- Capture interaction as theorem, determine lemmas needed to confirm
  - Hope: failure to prove under lemmas indicates likely interaction
Supporting Modular Verification

- Lemmas about individual features crucial
  - make product verification lightweight
  - help detect feature interactions

- Four kinds of lemmas helpful
  - type/format of inputs and outputs
  - environment info that might/won’t change
  - conditions characterizing changes
  - lifting lemmas through call-graph hierarchy

- Ideally automate lemma creation
Why Modularity?

Reviewer: modularity irrelevant for ACL2

We disagree

- modularity key part of design process
- features provide new form of modularity
- Research goal goes beyond ACL2
Reflections on ACL2

- Procedural-style natural match for features
  - features capture functional/behavioral information
- First-order limitation inhibits plug-and-play
  - Implementations use higher-order functions/classes
- Macros crucial
  - generate products and standard lemmas
- Books too restrictive for some feature lemmas
- *Hands-off* and *disable* hints simulate modular environment
Questions for Experts

- Better way to achieve plug-and-play?

- Way to use books for all feature lemmas?

- Results on lemma generation that we should know about?