Konfigurierbare Systemsoftware (KSS)

VL 5 – Variability Management in the Large: The VAMOS Approach

Daniel Lohmann
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität
Erlangen-Nürnberg
SS 12 – 2012-06-13
http://www4.informatik.uni-erlangen.de/Lehre/SS12/V_KSS

About this Lecture

Problem Space
Solution Space

Domain Expert

Features and Dependencies

Architecture and Implementation

System User

Specific Problem

Specific Solution

Features and Dependencies

Variant

System User

intentional side

extensional side

Implementation Techniques: Classification

Decompositional Approaches
- Text-based filtering (untyped)
- Preprocessors

Compositional Approaches
- Language-based composition mechanisms (typed)
- OOP, AOP, Templates

Generative Approaches
- Metamodel-based generation of components (typed)
- MDD, C++ TMP, generators

Real-world software uses them all!
5.1 Motivation
5.2 Variability in Linux
5.3 Configuration Consistency
5.4 Configuration Coverage
5.5 Summary
5.6 References

Typical Configurable Operating Systems...

one individual variant for each human being

320 optional, independent features

more variants than atoms in the universe!

1,250 features
Typical Configurable Operating Systems...

1,250 features

Challenges: → VAMOS
- How to maintain this?
- How to test this?
- Why so many features anyway?

* Variability Management in Operating Systems

12,000 features

The Linux Configuration and Generation Process

1️⃣ Configuration with a CONFIG frontend
2️⃣ Compilation of a subset of files
3️⃣ Selection of a subset of CPP Blocks
4️⃣ Linking of the kernel and loadable kernel modules

Dominancy and Hierarchy of Variability

l0: Feature Modelling 12,000 features
l1: Coarse-grained: KBUILD 31,000 source files
l2: Fine-grained: CPP 89,000 #ifdef blocks
l3: Language-level: GCC → if(CONFIG_SMP) ...
l4: Link-time: LD → branches in linker scripts
l5: Run-time: INSMOD, MODPROBE, ...

Agenda

5.1 Motivation
5.2 Variability in Linux
- Variability Implementation in Linux Challenges
5.3 Configuration Consistency
5.4 Configuration Coverage
5.5 Summary
5.6 References
Challenges with Implemented Variability

Central declaration of configurability: `CONFIG`

Distributed implementation of configurability: `MAKE`, `CPP`, `GCC`, `LD`

Problem Analysis: Configuration Consistency

- Symbolic
- Logic

Problem Analysis: Symbolic Inconsistency

```
config HOTPLUG_CPU
  bool "Support for hot-pluggable CPUs"
  depends on SMP && HOTPLUG
  ---help---
static int hotplug_cfd(struct notifier_block *nfb, unsigned long action, void *hcpu)
{
  // [...]
  switch (action) {
    case CPU_UP_PREPARE:
    case CPU_UP_PREPARE_FROZEN:
      // [...]
    #ifdef CONFIG_CPU_HOTPLUG
    case CPU_UP_CANCELED:
    case CPU_UP_CANCELED_FROZEN:
      free_cpumask_var(cfd->cpumask);
      break;
    #endif
    case CPU_DEAD:
    case CPU_DEAD_FROZEN:
      free_cpumask_var(cfd->cpumask);
      break;
    #endif
  }
  return NOTIFY_OK;
}
```

Result:
Fix for a critical bug
Problem Analysis: Logic Inconsistency

- Feature DISCONTIGMEM implies feature NUMA
- Inner blocks are not actually configuration-dependent
  - Block$_2$ is always selected $\rightarrow$ undead
  - Block$_3$ is never selected $\rightarrow$ dead

Linux contains superfluous #ifdef Blocks!

Result:
Code cleanup

Solution Approach: Consistency Validation

Problem and solution space are analyzed for configuration points:

\[ C = \text{(FLATMEM} \rightarrow \text{MEMORY\_MODEL}) \]
\[ \land \text{(DISCONTIGMEM} \rightarrow \text{MEMORY\_MODEL}) \]
\[ \land \text{(SPARSEMEM} \rightarrow \text{MEMORY\_MODEL}) \]
\[ \land \text{(NUMA} \rightarrow \text{MEMORY\_MODEL}) \]
\[ \land \text{(DISCONTIGMEM} \rightarrow \text{NUMA}) \]

Implementation: The UNDERTAKER

Job: Find (and eventually bury) dead #ifdef-code!

We have found 1776 configurability defects in Linux v2.6.35

Submitted 123 patches for 364 defects

20 are confirmed new bugs (affecting binary code)

Cleaned up 5129 lines of cruft code
Implementation: The UNDERTAKER

Job: Find (and eventually bury) dead #ifdef-code!

![Graph showing new and fixed configuration defects over Linux releases]

Common Beliefs About Variability in Linux

1. Most variability is expressed by boolean (or tristate) switches.
2. arch-x86 is the largest and allyesconfig selects most features.
3. Variability is mostly implemented with the CPP.
4. The Linux kernel is highly configurable.

Linux v3.1: Feature Distribution by Type

- Most variability is expressed by boolean (or tristate) switches

<table>
<thead>
<tr>
<th>Feature Type</th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>KCONFIG features</td>
<td>11,691</td>
<td>100%</td>
</tr>
<tr>
<td>Option-like</td>
<td>10,907</td>
<td>93.3%</td>
</tr>
<tr>
<td>Value-like</td>
<td>784</td>
<td>6.7%</td>
</tr>
<tr>
<td>Boolean</td>
<td>6,024</td>
<td>55.2%</td>
</tr>
<tr>
<td>Tristate</td>
<td>4,883</td>
<td>44.8%</td>
</tr>
<tr>
<td>String</td>
<td>87</td>
<td>11.1%</td>
</tr>
<tr>
<td>Integer/Hex</td>
<td>697</td>
<td>88.9%</td>
</tr>
</tbody>
</table>

⇒ Almost all features in Linux are option-like
Linux v3.1: Coverage of arch-x86 / allyesconfig

- arch-x86 is the largest and allyesconfig selects most features

<table>
<thead>
<tr>
<th>Category</th>
<th>Features</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>arch-x86</td>
<td>11,691</td>
<td>100%</td>
</tr>
<tr>
<td>non-arch-x86</td>
<td>3,915</td>
<td>33.5%</td>
</tr>
<tr>
<td>allyesconfig</td>
<td>5,482</td>
<td>46.9%</td>
</tr>
<tr>
<td>non-allyesconfig</td>
<td>2,294</td>
<td>19.6%</td>
</tr>
</tbody>
</table>

⇒ arch-x86/allyesconfig is not nearly a full configuration

Linux v3.1: Distribution by Granularity

- Variability is mostly implemented with the CPP

<table>
<thead>
<tr>
<th>Configuration Type</th>
<th>KCONFIG Features</th>
<th>KBUILD only</th>
<th>KBUILD/CPP</th>
<th>CPP only</th>
</tr>
</thead>
<tbody>
<tr>
<td>interpreted</td>
<td>11,691 [100%]</td>
<td>7,749 [66.3%]</td>
<td>1,925 [16.5%]</td>
<td>3,916 [33.5%]</td>
</tr>
<tr>
<td>only</td>
<td>1,209 [10.3%]</td>
<td>24.5%</td>
<td>48.5%</td>
<td>51.5%</td>
</tr>
</tbody>
</table>

⇒ KBUILD implements more than two thirds of all variation points

Linux v3.2: Distribution by HW/SW

- The Linux kernel is highly configurable

<table>
<thead>
<tr>
<th>Feature Group</th>
<th>KCONFIG Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software related</td>
<td>12,038 [100%]</td>
</tr>
<tr>
<td>Hardware related</td>
<td>10,551 [87.6%]</td>
</tr>
<tr>
<td>net</td>
<td>530 [4.4%]</td>
</tr>
<tr>
<td>misc</td>
<td>447 [3.7%]</td>
</tr>
<tr>
<td>drivers</td>
<td>5,330 [44.3%]</td>
</tr>
<tr>
<td>sound</td>
<td>536 [4.5%]</td>
</tr>
<tr>
<td>arch</td>
<td>4,685 [38.9%]</td>
</tr>
</tbody>
</table>

⇒ Software features account for only twelve percent of all variation points

Linux Feature Growth over Time (#Features, 2007–2012)

- All features
- HW features
  - arch/
  - drivers/
  - sound/
- SW features (everything else)
**Linux Feature Growth over Time**  
(#Features in arch, 2007–2012)

---

**Results: Where Have all the Features Gone?**

1. Most variability is expressed by boolean (or tristate) switches  
   - more than 93 percent of all features are option-like  
   - it is acceptable for tools to ignore value-type features  
   ✔

2. arch-x86 is the largest and allyesconfig selects most features  
   - more than 53 percent are not covered by this configuration  
   - other parts of Linux are probably less tested and error-prone!  
   ✗

3. Variability is mostly implemented with the CPP  
   - more than 66 percent of all features are handled by the build system, only 17 percent are handled by C++ only  
   - variability extraction from KBUILD is necessary  
   ✗

4. The Linux kernel is highly configurable  
   - only 12 percent of all features configure software only  
   - variability is mostly induced by advances in hardware  
   - complexity will increase further  
   ✗

---

**Challenges: Variability Extraction from the Build System**

- Variability extraction $\rightarrow$ which file is selected by which feature?  
- Usual approach for variability extraction [5, 7] (KCONFIG, CPP, ...):  
  
  ```
  source \rightarrow \text{parse} \& \text{transform} \rightarrow \text{propositional formula}
  ```

- Parsing does not work well for MAKE-languages  
  - declarative and Turing-complete languages  
  - special features, like shell, foreach, eval, addprefix, ...

- Linux’s KBUILD is built on top of (GNU) MAKE  
  - nevertheless, researchers have tried parsing to extract variability  
    - KBUILDMINER by Berger, She, Czarnecki, et al. [1]  
    - Nadi parser by Nadi and Holt [4]  
  - resulting tools are too brittle at best  
    - work for a (few) Linux version(s) only  
    - each usage of a special feature requires manual tailoring  

---

**Linux Build Process Revisited**

- `SMP=n`  
- `PM=y`  
- `APM=m`  

  ```
  CONFIG_SMP := n  
  CONFIG_PM := y  
  CONFIG_APM := m
  ```

  ```
  #undef CONFIG_SMP  
  #define CONFIG_PM 1  
  #undef CONFIG_APM  
  #define CONFIG_APM_MODULE 1
  ```
Variability Extraction from KBUILD with GOLEM [2]

Basic idea: Systematic probing and inferring of implications
SPLC '12: Dietrich, et al. [2]

- Dancing Makefiles
- Identification of KCONFIG references
- Recursion into subdirectory while considering constraints
- Robust with respect to architecture and version

no adaptations on or for KBUILD!

KCONFIG references

<table>
<thead>
<tr>
<th>Kernel version</th>
<th>found inferences</th>
<th>(93.7%)</th>
<th>(94.9%)</th>
<th>(95.1%)</th>
<th>(95.4%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>v2.6.25</td>
<td>6,274</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v2.6.28.6</td>
<td>7,032</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v2.6.33.3</td>
<td>9,079</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v2.6.37</td>
<td>10,145</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v3.2</td>
<td>11,050</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Case Study: Configuration Consistency

Configuration defects in Linux v3.2:

Without KBUILD constraints
- Code defects: 1835
- Referential defects: 415
- Logical defects: 83
  Sum: $\Sigma 2333$

With KBUILD constraints
- Code defects: 1835
- Referential defects: 439
- Logical defects: 299
  Sum: $\Sigma 2573$  Result: $+10\%$

Summary

Real-world system software offers thousands of features

- eCos: 1,250 features
- Linux: 12,000 features
  mostly induced by hardware!

This imposes great challenges for management and maintenance
- how to ensure configurability consistency?
- how to ensure configuration coverage?
- how to keep pace with the constant feature increase?

A strong call for adequate tool support $\Rightarrow$ VAMOS
- already found thousands and fixed hundreds of defects and bugs
- more to come!
Referenzen


