Konfigurierbare Systemsoftware (KSS)

VL 6 – Variability Management in the Large:
The VAMOS/CADOS Approach

Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität
Erlangen-Nürnberg

SS 16 – 2016-05-30

http://www4.informatik.uni-erlangen.de/Lehre/SS16/V_KSS

About this Lecture

Problem Space

Solution Space

Features and Dependencies

Architecture and Implementation

Specific Problem

Specific Solution

Domain Expert

Architect / Developer

System User

Variant

Configuration

intentional side

extensional side

intended properties

actual implementation

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!
Typical Configurable Operating Systems...

- 320 optional, independent features
- more variants than atoms in the universe!
- 1,250 features

one individual variant for each human being
Typical Configurable Operating Systems...

Challenges: → VAMOS/CADOS*

- How to maintain this?
- How to test this?
- Why so many features anyway?

* Variability Management in Operating Systems
Configurability-Aware Development of Operating Systems

The Linux Configuration and Generation Process

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

Dominancy and Hierarchy of Variability

Linux V3.2

- **l0**: Feature Modeling 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

- 6.1 Motivation
- 6.2 Variability in Linux
  - Variability Implementation in Linux
  - Challenges
- 6.3 Configuration Consistency
- 6.4 Configuration Coverage
- 6.5 Automatic Tailoring
- 6.6 Summary
- 6.7 References
Challenges with Implemented Variability

- Central declaration of configurability: KCONFIG
- Distributed implementation of configurability: MAKE, CPP, GCC, LD

Agenda

- 6.1 Motivation
- 6.2 Variability in Linux
- 6.3 Configuration Consistency
  - Problem Analysis
  - Solution Approach
  - Results
- 6.4 Configuration Coverage
- 6.5 Automatic Tailoring
- 6.6 Summary
- 6.7 References

Problem Analysis: Configuration Consistency

- Symbolic ☇ Logic ☇


```c
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:
  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

1. Feature DISCONTIGMEM implies feature NUMA
2. Inner blocks are not actually configuration-dependent
   - Block₂ is always selected → undead
   - Block₃ is never selected → dead

~ Linux contains superfluous #ifdef Blocks!

Result: Code cleanup

Solution Approach: Consistency Validation

Problem and solution space are analyzed for configuration points:

- DISCONTIGMEM → MEMORY Model
- DISCONTIGMEM → NUMA
- SPARSEMEM → MEMORY Model
- NUMA → MEMORY Model
- NUMA → 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!

Agenda

6.1 Motivation
6.2 Variability in Linux
6.3 Configuration Consistency
6.4 Configuration Coverage
   - Where Have All the Features Gone?
   - Results
   - Extracting Variability from KBUILD
   - Improvements
   - Implementation Space Coverage
6.5 Automatic Tailoring
6.6 Summary
6.7 References

Common Beliefs About Variability in Linux

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

⇒ Almost all features in Linux are option-like

Linux v3.1: Feature Distribution by Type

1. Most variability is expressed by boolean (or tristate) switches

<table>
<thead>
<tr>
<th>KCONFIG features</th>
<th>Option-like</th>
<th>Value-like</th>
</tr>
</thead>
<tbody>
<tr>
<td>11,691 [100%]</td>
<td>10,907 [93.3%]</td>
<td>784 [6.7%]</td>
</tr>
</tbody>
</table>

2. Option-like
   - Boolean 6,024 [51.5%]
   - Tristate 4,883 [41.8%]

3. Value-like
   - String 87 [0.7%]
   - Integer/Hex 697 [6%]
Linux v3.1: Coverage of arch-x86 / allyesconfig

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

\[\text{CONFIG features} 11,691 [100\%] \]

\[\text{arch-x86} 7,776 [66.5\%] \]

\[\text{non-arch-x86} 3,915 [33.5\%] \]

\[\text{allyesconfig} 5,482 [46.9\%] \]

\[\text{non-allyesconfig} 2,294 [19.6\%] \]

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

Linux v3.1: Distribution by Granularity

- Variability is mostly implemented with the CPP

\[\text{CONFIG features} 11,691 [100\%] \]

\[\text{KBUILD interpreted} 7,749 [66.3\%] \]

\[\text{KBUILD only} 1,925 [16.5\%] \]

\[\text{KBUILD/CPP} 3,916 [33.5\%] \]

⇒ KBUILD implements more than two thirds of all variation points

Linux v3.2: Distribution by HW/SW

- The Linux kernel is highly configurable

\[\text{CONFIG features} 12,038 [100\%] \]

\[\text{Software related} 1,487 [12.4\%] \]

\[\text{Hardware related} 10,551 [87.6\%] \]

\[\text{net} 530 [4.4\%] \]

\[\text{misc} 447 [3.7\%] \]

\[\text{drivers} 5,330 [44.3\%] \]

\[\text{sound} 536 [4.5\%] \]

\[\text{arch} 4,685 [38.9\%] \]

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

Linux Feature Growth over Time

- All features

- HW features: arch/ drivers/ sound/

- SW features (everything else)
Linux Feature Growth over Time  (#Features in arch, 2007–2012)

![Graph showing Linux feature growth over time from 2007 to 2012.](image)

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 CPP 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 ➞ which file is selected by which feature?
- Usual approach for variability extraction [7, 11] (KCONFIG, CPP, ...):
  - source ➞ parse & transform ➞ 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, and Wasowski [1]
    - Nadi parser by Nadi and Holt [5]
  - 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

[Diagram illustrating the Linux build process.](image)
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!

Kernel version found inferences

<table>
<thead>
<tr>
<th>Kernel Version</th>
<th>Found Inferences</th>
</tr>
</thead>
<tbody>
<tr>
<td>v2.6.25</td>
<td>6,274 (93.7%)</td>
</tr>
<tr>
<td>v2.6.28.6</td>
<td>7,032 (93.6%)</td>
</tr>
<tr>
<td>v2.6.33.3</td>
<td>9,079 (94.9%)</td>
</tr>
<tr>
<td>v2.6.37</td>
<td>10,145 (95.1%)</td>
</tr>
<tr>
<td>v3.2</td>
<td>11,050 (95.4%)</td>
</tr>
</tbody>
</table>

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%

Developer has to derive at least two configurations to ensure that the every line of code even compiles!

Issue: Decompositional Implementation of Variability

#ifdef CONFIG_NUMA
Block1
#else
Block2
@endif

Make sure that the submitted code...

"...has been carefully reviewed with respect to relevant KCONFIG combinations. This is very hard to get right with testing – brain-power pays off here."

Linux kernel patch submission checklist (Documentation/SubmitChecklist)
The VAMPYR Driver for Static Checkers

**Goal:** Maximize configuration coverage of existing tools
- Every configuration-conditional part should be covered at least once
- Statement coverage

⇒ Create a set of configurations and scan each individually

Results with GCC as Static Checker

**USENIX '14 [8]**

<table>
<thead>
<tr>
<th>Software Project</th>
<th>allyesconfig CC%</th>
<th>GCC invoss</th>
<th>GCC warnings</th>
<th>GCC errors</th>
<th>Issues</th>
<th>#/files per issue</th>
<th>#config</th>
<th>Overhead</th>
<th>GCC invoss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux/x86</td>
<td>78.6%</td>
<td>21.5%</td>
<td>203 (176)</td>
<td>1 (0)</td>
<td>232</td>
<td>110</td>
<td>4</td>
<td>26%</td>
<td></td>
</tr>
<tr>
<td>Hardware</td>
<td>76.0%</td>
<td>21.0%</td>
<td>180 (156)</td>
<td>1 (0)</td>
<td>181</td>
<td>82</td>
<td>0</td>
<td>26%</td>
<td></td>
</tr>
<tr>
<td>Software</td>
<td>82.7%</td>
<td>22.7%</td>
<td>21 (21)</td>
<td>0 (0)</td>
<td>21</td>
<td>351</td>
<td>0</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>Linux/arm</td>
<td>59.0%</td>
<td>22.7%</td>
<td>417 (204)</td>
<td>2 (15)</td>
<td>508</td>
<td>46</td>
<td>90</td>
<td>190%</td>
<td></td>
</tr>
<tr>
<td>Hardware</td>
<td>51.2%</td>
<td>23.4%</td>
<td>360 (252)</td>
<td>12 (15)</td>
<td>471</td>
<td>4</td>
<td>164</td>
<td>194%</td>
<td></td>
</tr>
<tr>
<td>Software</td>
<td>83.6%</td>
<td>19.5%</td>
<td>37 (62)</td>
<td>0 (0)</td>
<td>37</td>
<td>102</td>
<td>5</td>
<td>5%</td>
<td></td>
</tr>
<tr>
<td>Linux/mips64</td>
<td>54.5%</td>
<td>22.0%</td>
<td>220 (187)</td>
<td>29 (1)</td>
<td>249</td>
<td>85</td>
<td>91</td>
<td>91%</td>
<td></td>
</tr>
<tr>
<td>Hardware</td>
<td>42.1%</td>
<td>21.5%</td>
<td>174 (121)</td>
<td>17 (1)</td>
<td>191</td>
<td>72</td>
<td>69</td>
<td>69%</td>
<td></td>
</tr>
<tr>
<td>Software</td>
<td>79.3%</td>
<td>23.2%</td>
<td>46 (36)</td>
<td>1 (1)</td>
<td>58</td>
<td>128</td>
<td>22</td>
<td>22%</td>
<td></td>
</tr>
<tr>
<td>LA/FIASCO</td>
<td>98.1%</td>
<td>90.9%</td>
<td>see test</td>
<td>0 (0)</td>
<td>21</td>
<td>see test</td>
<td>16</td>
<td>16%</td>
<td></td>
</tr>
<tr>
<td>Bugbox</td>
<td>74.2%</td>
<td>63.0%</td>
<td>44 (35)</td>
<td>0 (0)</td>
<td>44</td>
<td>72</td>
<td>9</td>
<td>9%</td>
<td></td>
</tr>
</tbody>
</table>

Example: arch-arm

- Increased CC compared to allyesconfig from 60% to 84%
- 199 (+64%) additional issues reported by GCC
- 91 reported issues have to be considered as serious bugs
- 7 patches submitted – all got immediately accepted

Just by letting the compiler see all the code!
Idea: Automated Tailoring of Linux

- Distribution kernels today come with a **maximum** configuration
- As side-effect, this maximizes the **attack** surface!
- Each use-case needs its specific, ideal configuration

→ Automatically derive an **ideal** configuration for a given use case.

Automatic Tailoring: Approach

1. Prepare feature tracing
   - enable ftrace, or
   - patch source with flipper

2. Run test load, observe
   - address → #ifdef block

Main idea: “measure” needed features

- Start with standard distribution kernel
- Run use-case-specific test load → “observe” needed functionality
- Derive configuration for tailored kernel
Automatic Tailoring: Approach

1. Prepare feature tracing
   - enable ftrace, or
   - patch source with flipper
2. Run test load, observe
   - trace invoked kernel code
   - address \( \rightarrow \) #ifdef block
3. Map to partial config
   - blocks \( \rightarrow \) dependent blocks
   - blocks \( \rightarrow \) features

Specific scenario

- test load
- prepare
- self-reflective kernel
- baseline kernel

Automatic Tailoring: Results

4. x86-based server/workstation systems (LAMP, Desktop with NFS)
   - 90% fewer features, 9 entries on white list (out of 495–555)
   - 90% less executable code
   - 10% fewer functions with CVE entries
5. ARM-based low-cost appliances (raspBMC, Google Coder, Onionπ)
   - 70% fewer features, 14 entries on white list (out of 471–497)
   - 75% less executable code
6. ARM-based high-end ASIC (Nexus 4 with Ubuntu Phone)
   - 30% fewer features, 14 entries on white list (out of 850)
   - 25% less executable code

Evaluation: Reduction for LAMP

<table>
<thead>
<tr>
<th>Category</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>arch</td>
<td>33%</td>
</tr>
<tr>
<td>block</td>
<td>15%</td>
</tr>
<tr>
<td>crypto</td>
<td>71%</td>
</tr>
<tr>
<td>drivers</td>
<td>95%</td>
</tr>
<tr>
<td>fs</td>
<td>86%</td>
</tr>
<tr>
<td>ipc</td>
<td>38%</td>
</tr>
<tr>
<td>kernel</td>
<td>34%</td>
</tr>
<tr>
<td>lib</td>
<td>25%</td>
</tr>
<tr>
<td>mm</td>
<td>8%</td>
</tr>
<tr>
<td>net</td>
<td>87%</td>
</tr>
<tr>
<td>sound</td>
<td>100%</td>
</tr>
<tr>
<td>others</td>
<td>62%</td>
</tr>
</tbody>
</table>
Results: Automatic Tailoring

HotDep ’12: Tartler, Kurmus, Ruprecht, Heinloth, Rothberg et al. [9]

- TCB is significantly smaller
- Easy to use: process is fully automated
- If necessary, the tailoring can be guided with whitelists and blacklists
- Going further: Dynamic ASR
  - Even if present: Who is allowed to call what
  - At runtime: Block illegal invocations.

Summary

Real-world system software offers thousands of features
- eCos: 1,250 features}
- Linux: 12,000 features
} mostly induced by hardware!
- central declaration (ecosConfig, KCONFIG)
- distributed, multi-paradigm implementation (MAKE, CPP, GCC, ...)

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/CADOS
- already found thousands and fixed hundreds of defects and bugs
- more to come!

Referenzen


