Konfigurierbare Systemsoftware (KSS)

VL 5 – 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 15 – 2015-05-15
http://www4.informatik.uni-erlangen.de/Lehre/SS15/V_KSS

About this Lecture
Problem Space Solution Space

Specific Problem Specific Solution
Domain Expert Architect / Developer

Features and Dependencies

System User Configuration

intentional side

Architecture and Implementation

Variant

System User

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!
Typical Configurable Operating Systems...

one individual variant for each human being

320 optional, independent features

more variants than atoms in the universe!

ecos

1,250 features
Typical Configurable Operating Systems...

1,250 features

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 KCONFIG 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

- Feature Modeling 12,000 features
- Coarse-grained: KBUILD 31,000 source files
- Fine-grained: CPP 89,000 #ifdef blocks
- Language-level: GCC if(CONFIG_SMP) ...
- Link time: LD branches in linker scripts
- 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 Automatic Tailoring
5.6 Summary
5.7 References
Challenges with Implemented Variability

**Central declaration of configurability:**  
*KCONFIG*

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

**Configuration Consistency?**

**Configuration**

**Implementation**

**Coverage?**

Problem Analysis: Configuration Consistency

**Symbolic:**

```
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;
}
```

**Logic:**

Result: Fix for a critical bug

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:
    case CPU_DEAD:
    case CPU_DEAD_FROZEN:
      free_cpumask_var(cfd->cpumask);
      break;
    #endif
    }
  return NOTIFY_OK;
}
```

Agenda

5.1 Motivation
5.2 Variability in Linux
5.3 Configuration Consistency
   - Problem Analysis
   - Solution Approach
   - Results
5.4 Configuration Coverage
5.5 Automatic Tailoring
5.6 Summary
5.7 References
**Solution Approach: Consistency Validation**

Problem and solution space are analyzed for configuration points:

- 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

### 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!

How good is this, really?

c⃝ dl KSS (VL 5 | SS 15) 5 The VAMOS/CADOS Approach | 5.3 Configuration Consistency 5–18

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

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

KCONFIG features
- 11,691 [100%]

Option-like
- 10,907 [93.3%]

Value-like
- 784 [6.7%]

Boolean
- 6,024 [51.5%]

Tristate
- 4,883 [41.8%]

String
- 87 [0.7%]

Integer/Hex
- 697 [6%]

⇒ Almost all features in Linux are option-like

5.4 Configuration Coverage

5.1 Motivation
5.2 Variability in Linux
5.3 Configuration Consistency
5.4 Configuration Coverage
Where Have All the Features Gone?
Results
Extracting Variability from KBUILD
Improvements
Implementation Space Coverage
5.5 Automatic Tailoring
5.6 Summary
5.7 References
Linux v3.1: Coverage of arch-x86 / allyesconfig

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

<table>
<thead>
<tr>
<th>Feature</th>
<th>KCONFIG features</th>
<th>KBUILD interpreted</th>
<th>KBUILD only</th>
<th>KBUILD/CPP</th>
<th>CPP interpreted</th>
</tr>
</thead>
<tbody>
<tr>
<td>arch-x86</td>
<td>11,691 [100%]</td>
<td>7,749 [66.3%]</td>
<td>1,925 [16.5%]</td>
<td>1,899 [16.2%]</td>
<td>3,916 [33.5%]</td>
</tr>
<tr>
<td>non-arch-x86</td>
<td>3,915 [33.5%]</td>
<td>3,514 [75.5%]</td>
<td>125 [4.8%]</td>
<td>125 [4.8%]</td>
<td>51.5%</td>
</tr>
<tr>
<td>allyesconfig</td>
<td>5,482 [46.9%]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>non-allyesconfig</td>
<td>2,294 [19.6%]</td>
<td></td>
<td></td>
<td></td>
<td></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>Feature</th>
<th>KCONFIG features</th>
<th>KBUILD interpreted</th>
<th>KBUILD only</th>
<th>KBUILD/CPP</th>
<th>CPP interpreted</th>
</tr>
</thead>
<tbody>
<tr>
<td>arch-x86</td>
<td>11,691 [100%]</td>
<td>7,749 [66.3%]</td>
<td>1,925 [16.5%]</td>
<td>1,899 [16.2%]</td>
<td>3,916 [33.5%]</td>
</tr>
<tr>
<td>non-arch-x86</td>
<td>3,915 [33.5%]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>allyesconfig</td>
<td>5,482 [46.9%]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>non-allyesconfig</td>
<td>2,294 [19.6%]</td>
<td></td>
<td></td>
<td></td>
<td></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</th>
<th>KCONFIG features</th>
<th>KBUILD interpreted</th>
<th>KBUILD only</th>
<th>KBUILD/CPP</th>
<th>CPP interpreted</th>
</tr>
</thead>
<tbody>
<tr>
<td>arch</td>
<td>4,685 [38.9%]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>drivers</td>
<td>5,330 [44.3%]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sound</td>
<td>536 [4.5%]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>misc</td>
<td>447 [3.7%]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>net</td>
<td>530 [4.4%]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arch/ drivers/ sound/</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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)
**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, ...):
  
  ![process diagram]

  - 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 [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**

- **Kconfig** selection
- **Source files**
- **Build scripts**
  - `Makefile`
  - `arch/x86/init.c`
  - `arch/x86/init.c`
  - `arch/x86/...`
  - `lib/Makefile`
  - `kernel/sched.c`
- `kbuild`
Variability Extraction from KBUILD with GOLEM [2]

Basic idea: Systematic probing and inferring of implications

- 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!

Kernelversion found inferences

<table>
<thead>
<tr>
<th>Kernelversion</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>v2.6.37.3</td>
<td>10,145 (95.1%)</td>
</tr>
<tr>
<td>v3.2</td>
<td>11,050 (95.4%)</td>
</tr>
</tbody>
</table>

Case Study: Configuration Consistency

Without KBUILD constraints

<table>
<thead>
<tr>
<th>Without KBUILD constraints</th>
<th>Code defects</th>
<th>Referential defects</th>
<th>Logical defects</th>
<th>Sum: Σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code defects</td>
<td>1835</td>
<td>415</td>
<td>83</td>
<td>2333</td>
</tr>
</tbody>
</table>

With KBUILD constraints

<table>
<thead>
<tr>
<th>With KBUILD constraints</th>
<th>Code defects</th>
<th>Referential defects</th>
<th>Logical defects</th>
<th>Sum: Σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code defects</td>
<td>1835</td>
<td>439</td>
<td>299</td>
<td>2573</td>
</tr>
</tbody>
</table>

Result: +10%

Implementation Space Coverage

Issue: Decompositional Implementation of Variability

```c
#define CONFIG_NUMA
Block1
#else
Block2
#endif
```

Make sure that the submitted code...
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

<table>
<thead>
<tr>
<th>Software Project</th>
<th>allyesconf CE (%)</th>
<th>allyesconf CC (%)</th>
<th>Overhead: increase of GCC Invocations</th>
<th>GCC warnings</th>
<th>GCC errors</th>
<th>Σ Issues</th>
<th># Edits blocks per reported issue (hit)</th>
<th>Result: increase of GCC messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux/x86</td>
<td>78.0%</td>
<td>88.4%</td>
<td>21.5%</td>
<td>203 (17%)</td>
<td>1 (0)</td>
<td>202 (110)</td>
<td>26 (+15%)</td>
<td></td>
</tr>
<tr>
<td>hardware</td>
<td>76.0%</td>
<td>86.5%</td>
<td>21.0%</td>
<td>180 (15)</td>
<td>1 (0)</td>
<td>181 (82)</td>
<td>26 (+17%)</td>
<td></td>
</tr>
<tr>
<td>software</td>
<td>82.7%</td>
<td>92.4%</td>
<td>22.7%</td>
<td>21 (21)</td>
<td>0 (0)</td>
<td>21 (351)</td>
<td>0 (+0%)</td>
<td></td>
</tr>
<tr>
<td>Linu</td>
<td>x86e</td>
<td>59.0%</td>
<td>84.4%</td>
<td>22.1%</td>
<td>417 (204)</td>
<td>02 (15)</td>
<td>508 (46)</td>
<td>190 (+64%)</td>
</tr>
<tr>
<td>hardware</td>
<td>51.2%</td>
<td>80.1%</td>
<td>25.7%</td>
<td>380 (262)</td>
<td>02 (15)</td>
<td>471 (34)</td>
<td>194 (+70%)</td>
<td></td>
</tr>
<tr>
<td>software</td>
<td>83.6%</td>
<td>96.3%</td>
<td>19.5%</td>
<td>37 (29)</td>
<td>37 (102)</td>
<td>5 (5)</td>
<td>3 (24)</td>
<td></td>
</tr>
<tr>
<td>Linux/arm</td>
<td>54.5%</td>
<td>90.9%</td>
<td>22.0%</td>
<td>220 (187)</td>
<td>29 (1)</td>
<td>249 (85)</td>
<td>91 (+68%)</td>
<td></td>
</tr>
<tr>
<td>hardware</td>
<td>42.1%</td>
<td>86.2%</td>
<td>21.5%</td>
<td>174 (172)</td>
<td>17 (1)</td>
<td>191 (72)</td>
<td>69 (+77%)</td>
<td></td>
</tr>
<tr>
<td>software</td>
<td>79.9%</td>
<td>96.3%</td>
<td>23.2%</td>
<td>46 (36)</td>
<td>1 (0)</td>
<td>58 (128)</td>
<td>22 (+31%)</td>
<td></td>
</tr>
<tr>
<td>L4/FIASCO</td>
<td>90.1%</td>
<td>99.8%</td>
<td>see test</td>
<td>20 (5)</td>
<td>21 (16)</td>
<td>see test</td>
<td>16 (+32%)</td>
<td></td>
</tr>
<tr>
<td>Busybox</td>
<td>74.2%</td>
<td>97.3%</td>
<td>60.3%</td>
<td>44 (35)</td>
<td>0 (0)</td>
<td>44 (72)</td>
<td>9 (+26%)</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
   - trace invoked kernel code
   - address \( \rightarrow \) #ifdef block

Main idea: “measure” needed features

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

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

Automatic Tailoring: Results [4, 6, 9]

- 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
- 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
- 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

- arch: 33%
- block: 15%
- crypto: 71%
- drivers: 95%
- fs: 86%
- ipc: 38%
- kernel: 34%
- lib: 25%
- mm: 8%
- net: 87%
- sound: 100%
- others: 62%
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 → VAMOS/CADOS
  - already found thousands and fixed hundreds of defects and bugs
  - more to come!

Referenzen (Cont'd)


