Software Product Lines (SPLs)

(Claus Brabrand, Associate Professor, IT University of Copenhagen)

Software Product Lines (SPLs)

Many companies do not have just one product they develop, maintain, and sell, but a whole ***family of (related!) products.*** (Apple, for instance, has a *family of related products*; e.g., `iPhones` and `iPads` with both similarities and differences.) The code of such a "program family" is often represented using a **mix** of ***"shared parts"*** (code fragments that are *shared* among similar products) and ***"distinct parts"*** (code fragments that are specific to particular products). The distinct parts are often mixed into the program using **`#ifdef`**'s to include extra pieces of code only for certain products or feature combinations (e.g., add extra code if "`DeLuxe`" version is *enabled* and "`TrialVersion`" is *disabled*). From a software perspective, such a family of products gives rise to a ***family of programs,*** which is often referred to as ***"Software Product Line" (aka, SPL):***

Figure 1: ***Software Product Line (aka, "Program Family"):*** **A family of cars** (above)**,**
a family of phones (below)**; even Linux** (right) **is essentially a *family* of program variants.**

(http://www.version2.dk/artikel/dansk-it-forsker-vinder-it-sikkerhedspris-banebrydende-analysesoftware-70913)

Programming Software Product Lines

In terms of **programming** a software product line (family of programs), let us consider a *very simple* toy example which has N=2 features, **INC** and **NEG**, both of which can be *enabled* or *disabled* which means that we have $2^N = 4$ distinct products/program variants (as shown below):

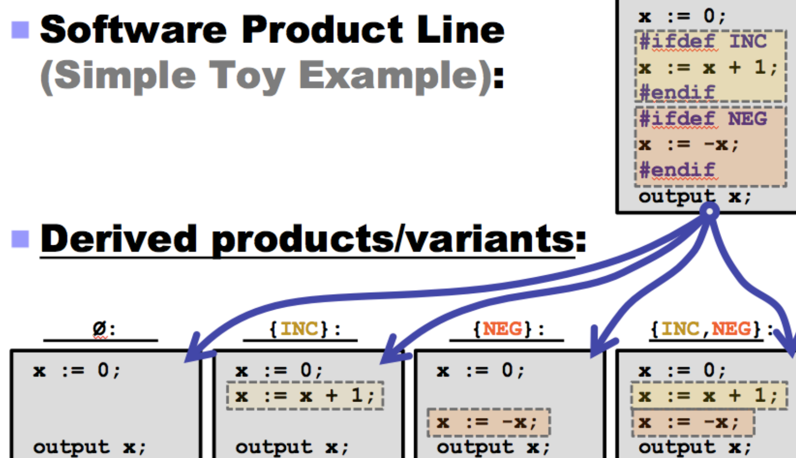

Figure 2: ***Software Product Line*** (above) **with its *four program variants* from the family** (below)**.**

Software Product Lines have many advantages; in particular, by maximizing code reuse among products (and programs).

Analyzing all Variants of a SPL

Importantly, however, most tools (incl. code analyzers) are ***not*** able to cope with the **`#ifdefs`** and hence analyze all the variants of the SPL. Instead, developers have to resort to the brute-force "generate-and-analyze" approach (i.e., generate and analyze all variants, one-by-one):

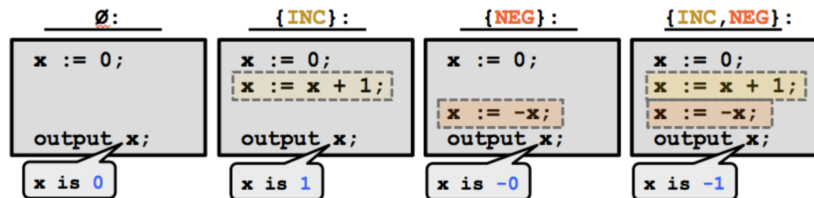

Figure 3: Brute force analysis of all four ($2^{N=2}$) program variants of our toy example SPL.

However, since the number of program variants is ***exponential****** in the number of features (N features yield 2^N variants), the generate-and-analyze strategy is infeasible for even moderately sized SPLs (for some SPLs, exhaustive analysis could take several thousands of years). This means that, currently, problems and errors (even in `iPhones`) are only discovered later when a particular combination of features (aka, configuration) happens to be selected and the corresponding product is derived and compiled/run/tested.

Analyzing a Software Product Line Directly

Recently, we have shown that it is possible to analyze a Software Product Line directly without generating all the program variants explicitly (i.e., analyzing the program family with **`#ifdefs`**):

- ***"Intraprocedural Dataflow Analysis for Software Product Lines"*** :-)

Furthermore, we have shown that it is possible to run such analyses efficiently: in minutes instead of years (actually in as little as 12 minutes):

- ***"SPL^{LIFT}: Statically Analyzing Software Product Lines in Minutes instead of Years"*** :-)

Note: Won German IT-Security Award (second place and € 60.000 EUR) in November 2014. :-)

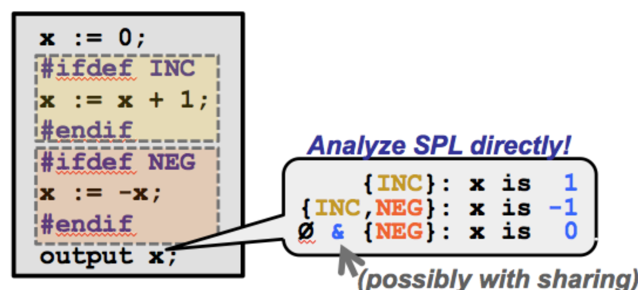

Figure 4: Analyzing a Software Product Line directly (possibly with sharing).

The above analysis runs only *once* as opposed to 2^N times in the generate-and-analyze (see Fig. 3).

With this, let me present my **M.Sc. Project Proposals on Software Product Lines**:

---

* **On combinatorial explosion:** It suffices to have 33 independent features to yield more variants than people on the planet (2^{33}). As little as 320 features yield more configurations than the number of atoms in the known universe. The Linux kernel code has more than 10 000 features!

M.Sc. Project Proposals on Software Product Lines

(Claus Brabrand, Associate Professor, IT University of Copenhagen)

#1) "EVALUATE the 'One-Disabled' Testing and Analysis Strategy on LINUX"

In recent research into **finding bugs in Linux**, we got an indication that testing/analyzing variants where ***exactly-one****-feature-has-been-****disabled*** might be a highly effective strategy for finding bugs:

- *"42 Variability Bugs in the Linux Kernel: A Qualitative Study"* (esp. OBS 10 and TABLE 3)

The idea of this project is to try to evaluate whether or not this is indeed the case. Comparing this new strategy against current practice; ***"random variant testing"*** (i.e., testing a similar amount of random variants) and the so-called ***"2-way testing"*** (i.e., testing all combinations of two features enabled), will reveal whether or not this is indeed a better error-finding strategy.

[**Note:** this project can be combined with #2 (below).]

- **Main tasks:** *devise experimental set up*, *evaluation*, and *data analysis*.
- **Programming:** probably little *programming*, but more *evaluation* and *analysis*.

If you are interested, come ask me about this project!

#2) "Use the brute-force 'generate-and-analyze' strategy to FIND BUGS in LINUX"

The idea of this project is to ***find and deploy a collection of errors finding tools on Linux*** using the brute-force generate-and-analyze strategy. We have indications that simply compiling program variants will find many ***compile-time errors*** and that certain analyses (esp., *uninitialized variables*, *buffer overflows*, and *null pointers*) might be effective for finding ***runtime-errors in Linux***. One of the challenges is to figure out which combinations of features (aka, configurations) and parts of Linux to maximize the chance of finding real bugs. Since configurations are independent, it is possible to run analyses in parallel (which could speed up analysis tremendously). Potentially, one might even be able to speculate about the ***error frequencies*** in Linux?

[**Note:** this project can be combined with #1 (above).]

- **Main tasks:** *devise experimental set up*, *evaluation*, and *data analysis*.
- **Programming:** infrastructure for deploying analyzers on configurations.

If you are interested, come ask me about this project!

#3) "VISUALIZATION of Software Product Lines"

Software Product Lines are ***much more complex*** than normal single-product programs because embody a whole program-family of up to 2^N programs! So far, not many tools are available for getting an overview and navigating the variability of a Software Product Line. The idea of this project is to try to figure out how to ***help SPL developers*** by appropriately visualizing the code. ***A visualizer could help*** identify complex areas and allow developers to "see" the relation between products, and/or a single product and the whole program family?

[**Note:** this project could be combined with #4 (below).]

- **Main tasks:** *visualization* and *evaluation of the visualization*.
- **Programming:** SPL visualizer.

If you are interested, come ask me about this project!

#4) "Variability METRICS (e.g., VARIATIONAL CYCLOMETRIC COMPLEXITY)"

Lots of software metrics are available for measuring various aspects of normal programs. However, these are not able to cope with `#ifdefs`. The idea of this project is to ***identify interesting metrics*** (especially metrics of complexity) and to lift them to work on the Software Product Lines (i.e., with the `#ifdefs`). In particular, it would be interesting to ***lift*** the ***"Cyclometric Complexity"*** measure—intended to measure how complex a program fragment is for a developer—to Software Product Lines. Presumably, complexity grows exponentially for a developer when dealing with 2^N programs instead of just one?

[**Note:** this project could be combined with #3 (above) or #5 (below).]

- **Main tasks:** *identify metrics*, *lift metrics to SPLs*, and *evaluate proposed metrics*.
- **Programming:** tool for measuring various aspects of an SPL (especially complexity).

If you are interested, come ask me about this project!

#5) "N features => 2^N Products: Effect on Developers"

In recent research, we have found 42 so-called ***"variability bugs" in Linux*** that are due to combinations of features having to be enabled and disabled:

- *"42 Variability Bugs in the Linux Kernel: A Qualitative Study"*

In that work, we derived ***simplified versions*** of each of the bugs. We thus have a collection of 42 simplified bugs. Instead of just having two versions of each bug (the real code and the simplified version), it would be interesting to take a few of these bugs and turn them into ***families of progressively harder versions of a bug***. (Indeed such a bug family could itself be represented via `#ifdefs`.) This could make it possible to ***study correlations*** between bug complexity and difficulty for developers. This could provide insights into how much more difficult SPLs are for programmers to cope with compared to single programs (and which aspects are difficult)?

[**Note:** this project can be combined with #4 (above).]

- **Main tasks:** develop progressively harder versions of a bug and evaluate bug difficulty.
- **Programming:** limited programming.

If you are interested, come ask me about this project!

#6) "Implement lifted dataflow analyses to FIND BUGS in JAVA SPLs"

Recently, we showed how to lift dataflow analyses to work on SPLs (i.e., with `#ifdefs`):

- *"Intraprocedural Dataflow Analysis for Software Product Lines"*
- *"SPL^LIFT: Statically Analyzing Software Product Lines in Minutes instead of Years"*

The idea of this project is to implement and deploy ***lifted analyses*** to try to ***find errors*** in five Java Software Product Lines (GPL, Mobile Media, Lampiro, Prevayler, and BerkeleyDB). Note that this project requires knowledge on dataflow analyses.

- **Prerequisite:** knowledge on **dataflow analysis required!**
- **Main tasks:** implement lifted analyses and use analyses to find errors.
- **Programming:** implement lifted dataflow analyses.

If you are interested, come ask me about this project!

#7) "Fix in Code -vs- Fix in Mapping -vs- Fix in Model"
When programming Software Product Lines, there are three layers at work: • *code* (solution domain); • *feature model* (problem domain); and • *mapping* between the two (problem <--> solution). Often, these are programmed in different languages; in particular, for the Linux Kernel, the languages involved are: C for the *code*, `KConfig` for the *feature model*, and `CPP` and `Make` for the *mapping*. See more here: • *"42 Variability Bugs in the Linux Kernel: A Qualitative Study"* It would be interesting to devise an experiment to quantify the differences between the three layers and languages. For instance, one could presumably compare the time spent and difficulties involved in e.g. debugging some of the 40 bugs (see above paper).
• **Main tasks:** *devise experimental set up*, *evaluation*, and *data analysis*. • **Programming:** probably little *programming*, but more *evaluation* and *analysis*.
If you are interested, come ask me about this project!

#8) "Analysis abstracton via Source-to-Source Transformation before Analysis"
Recently, we found out that it's possible to move "analysis abstractions" (that trade precision for speed) out of the analysis. In fact, it is possible to apply such abstractions to source programs – *independent of* – and even *before* the program is analyzed. We have made such a tool (called the "Reconfigurator") which takes a software product line as input along with an abstraction (trading analysis precision for speed) and applies the abstraction directly to the input program in such a way that when subsequently analyzed, the abstracted program will behave as if the analysis itself had been abstracted. It would be interesting to experiment with this "reconfigurator" tool in the context of other types of analyses (in particular, model-checking) for which there are lifted analyses (analyses that work at the SPL level).
• **Main tasks:** *investigate the "reconfigurator" source-to-source transformation in other contexts.* • **Programming:** reprogram the reconfigurator to work with other languages/formalisms.
If you are interested, come ask me about this project!

#9) "Transforming Undisciplined #ifdefs into Disciplined #ifdefs (in The Linux Kernel)"
Many SPLs use so-called undisciplined (aka lexical) #ifdefs such as the following example shows:

`if (x>0) f("pos");`
`#ifdef POS_AND_NEG`
`else f("neg");`
`#endif`
	`-->`
	`disciplined`
		`#ifdef POS_AND_NEG`
		`if (x>0) f("pos");`
		`else f("neg");`
		`#else`
		`if (x>0) f("pos");`
		`#endif`

Undisciplined #ifdefs add code fragments that only span parts of a syntactic category; in this case only "half" of a statement (the fragment in red). In most cases, it is possible to discipline them via source-to-source program transformation as also shown above (to the right). This project is about attempting to identify and discipline the top 10-20 most common cases occuring in Linux.
• **Main tasks:** *idenfify common undisciplined #ifdef patterns and how to discipline them* • **Programming:** mainly a source-to-source program transformation
If you are interested, come ask me about this project!