Timeline Variability

The Variability of Binding Time of Variation Points

Eelco Dolstra
Gert Florijn
Eelco Visser

Technical Report UU-CS-2003-052
Institute of Information and Computing Sciences
Utrecht University

January 2003

Preprint of
E. Dolstra, G. Florijn, and E. Visser. Timeline Variability: The Variability of Binding Time of Variation Points. In Workshop on Software Variability Modeling (SVM'03), number IWI preprint 2003-7-01, Groningen, The Netherlands, 2003. Reseach Institute of Computer Science and Mathematics, University of Groningen.

ISSN 0924-3275

Address:
Eelco Dolstra
Utrecht University, P.O. Box 80089,
3508 TB Utrecht, The Netherlands
eelco@cs.uu.nl

Gert Florijn
SERC, P.O. Box 424,
3500 AK Utrecht, The Netherlands
florijn@serc.nl

Eelco Visser
Utrecht University, P.O. Box 80089,
3508 TB Utrecht, The Netherlands
visser@cs.uu.nl

Timeline Variability: The Variability of Binding Time of Variation Points

Eelco Dolstra
Utrecht University, P.O. Box 80089,
3508 TB Utrecht, The Netherlands
eelco@cs.uu.nl

Gert Florijn
SERC, P.O. Box 424,
3500 AK Utrecht, The Netherlands
florijn@serc.nl

Eelco Visser
Utrecht University, P.O. Box 80089,
3508 TB Utrecht, The Netherlands
visser@cs.uu.nl

1. Introduction

Timeline variability is the ability of a software system to have variation points bound at different moments of the system's life-cycle.

Virtually every non-trivial software system exhibits *variability*: the property that the set of *features*— characteristics of the system that are relevant to some stakeholder— can be changed at certain points in the system's life-cycle. The parts of the system that implement the ability to make such changes are called *variation points*. Selecting some variant supported by a variation point is called *binding* the variant. Every variation point has at least one associated *binding time*: the moment in the system's life-cycle at which the variation point can be bound. A more detailed exposition of this terminology can be found in, e.g., [6, 2].

For example, the decision to build an operating system kernel with multiprocessor support, or to build a "light" or "professional" version of a word processor, might be implemented at build time. On the other hand, the decision to include support for some brand of hard drive in an operating system, or to use some particular language for spell checking in a word processor, might be made at runtime.

Generally, one would like variation points to be as flexible as possible with regard to binding time. That is, ideally one wants to have the ability to bind a variation point at build time, installation time, runtime, and so on. This leads to the notion of *timeline variability*: that certain features can be bound at *several* stages of the life-cycle. We do not formalise the term *timeline* here. Intuitively, we use if to refer to the set of distinguished moments during the build and deployment process where a user can potentially select variants. For example, the Linux operating system kernel allows functionality, e.g., device drivers, to be included either at build time or at runtime. However, changing features at runtime proceeds through entirely different interfaces than changing them at build time. Similarly, the Apache httpd webserver allows server extensions to be included at build time or at load time, but through different configuration mechanisms. Microsoft Office 2000 allows components to be installed either at install time proper or on demand, at runtime.

The concept of timeline variability—that is, *variability of binding time*—should not be confused with the binding time of variation points. In this paper we illustrate timeline variability through two case studies, Apache and the Linux kernel, and show that the two main technical issues in timeline variability are *inconsistent configuration interfaces* and *ad hoc implementation mechanisms*. We also provide some directions for future research.

2. Examples

In this section we show some examples of timeline variability in real systems. As we shall see, implementation of such variability is problematic. Consider, for example, a binary variation point that is bound at runtime, implemented in C. This is not hard to implement:

```
if (feature) f() else g();
```

Moving this variation point to build time is not hard either using conditional compilation:

```
#if FEATURE
  f()
#else
  g()
#endif
```

But suppose we wish to allow for this feature to be bound both at build time and runtime. A possible implementation would be:

```
#if FEATURE_BOUND_AT_BUILD_TIME
#if FEATURE
 f()
#else
  g()
#endif
#else
  if (feature) f() else g()
#endif
```

which is not very elegant. For more complex variation points, the situation becomes even worse.

2.1. The Linux kernel

The Linux kernel provides the basis for several variants of the GNU/Linux operating system. The kernel's job is to virtualise the hardware (e.g., provide multitasking and virtual memory) and abstract from it (e.g., provide a unifying interface to different types of storage devices or file systems).

The Linux kernel was originally implemented as a traditional monolithic kernel. In this situation all device drivers are statically linked into the kernel image file. Conditional defines and makefile manipulation are used to selectively include or exclude drivers and other features.

The disadvantage of this approach is that it closes a large number of variation points at build time. Hence, the kernel was retro-fitted with a *module* system. A set of source files constituting a module can be compiled into an object file and linked statically into the kernel image, or compiled into an object file that is stored separately and may be dynamically linked into a running kernel. Modules may refer to symbols exported by other modules. A tool exists to automatically determine the resulting dependencies to ensure that modules are loaded in the right order.

The implementation of the variation points realised through the module system is for the most part straightforward. For example, operations on block or character device files are implemented through dispatch through a function pointer; this is a feature of standard C. However, these function pointers must at some point be *registered* (i.e., be made known to the system), and this cannot be done in standard C. In particular, every module exports an initialisation function which must be called during kernel initialisation, in the case of statically linked modules, or at module load time, in the case of dynamically loaded modules. The C language, however, does not provide a mechanism to iterate over a set of function *names* that are not statically known. For example, we have no way of calling every function called init_module() that is linked into the executable image.

The Linux kernel solves this problem through the technique of emitting certain data in specially designated *sections* of the executable image. An invocation of the macro __initcall(f) arranges for the address of f to be placed in the special section .initcall.init;

```
typedef int (*initcall_t)(void);

#define __initcall(f) \
  static initcall_t __initcall_##f \
    __attribute__((unused,__section__\
      (".initcall.init"))) \
    = f
```

A module can declare some initializer f by invoking the macro module_init(f). For statically linked modules, module_init expands to an invocation of __initcall, and so the address of f is emitted in the .initcall.init section. We can then iterate through all initialisers as follows:

```
initcall_t *call = &__initcall_start;
do {
  (*call)();
  call++;
} while (call < &__initcall_end);
```

The symbols __initcall_start and __initcall_end are emitted at the start and end of the .initcall.init section by the linker script that guides the linker.

For dynamically loaded modules, on the other hand, module_init(f) emits a symbol init_module as an alias for f. The module loader will simply look this symbol up and call it.

Hence, we achieve timeline variability of module activation extending to build time and runtime, through a combination of preprocessor, compiler, and linker magic.

Cross-cutting features One problem facing the scheme implementing the module system is that it is closely tied to the structure of source modules; it is therefore difficult to modularise features that are not localisable into one or a few distinct source modules, i.e., cross-cutting features. An example is whether the kernel is built for uniprocessing or for symmetric multiprocessing (SMP). In an SMP configuration, many kernel data structures have to be guarded carefully against concurrent access; this affects a large amount of code. Quantitatively, we can get an indication of the degree to which a feature cross-cuts a system by counting the ifdefs conditionalised on the feature variable. In this case, we see that #ifdef CONFIG_SMP occurs more that 540 times in 250 source files of version 2.4.10 of the kernel. Because they impact so many source components, cross-cutting features are not very well suited for dynamic

loading. Additionally, variation points such as SMP support affect the definition of data structures, which makes it practically impossible to bind them at any time later than build time.

Analysis A problem of the Linux kernel is its monolithic distribution. If a feature is required that is not part of the distribution, either the kernel must be patched (e.g., the JFS file system) or the code must be compiled separately, outside of the kernel source tree (e.g., the ALSA sound system). Note that the latter solution makes static linking into the kernel impossible, the build mechanism is totally different, and it creates more work for users. Dynamic source tree composition [3] can alleviate this problem.

Note that the timeline variability of the module system does not directly extend to startup time, i.e., the loading of the kernel, since the kernel may not have the ability to load kernel modules at boot time. For example, the modules supporting the storage medium and file system on which the modules are stored must be statically linked into the kernel to prevent a chicken-and-egg problem. In essence, the timeline variation point has been closed with respect to startup time by the problem domain. However, an *initial ramdisk* (which is part of the kernel's image) may be used to store the required modules, thus extending the timeline variability to startup time.

2.2. Apache

The Apache httpd server is a freely available web server. In order to support various kinds of dynamic content generation, authentication, etc., the server provides a module system. Modules can be linked statically, or dynamically, at startup time. Dynamically loaded modules can be compiled inside or outside the Apache source tree.

Apache faces the same problem as the Linux kernel: how to register a variable set of modules (i.e., how to make statically included modules known to the core system)? The solution used by the Apache developers is to have the configuration script generate a C source file containing a list of pointers to the module definition structures:

```
module *ap_preloaded_modules[] = {
  &core_module,
  &access_module,
  &auth_module,
  ...
};
```

Note that this solution is again, in a sense, outside of the C language; we need to *generate* C code (i.e., externally) in order to deal with these open variation points.

Analysis Note that neither Apache nor the Linux kernel take advantage of static linking beyond the fact that it may be a necessity, e.g., dynamic linking may not be available on some platforms on which Apache is configured, provides simplified runtime characteristics, or, in the case of the Linux kernel, may be perceived as a security feature (the absence of dynamic loading of kernel modules makes it a little bit more difficult to subvert the kernel). Compile time knowledge of the module configuration does not lead to more efficient code, since this requires cross-module optimisation; many C compilers are not capable of this.

2.3. Issues

So what are the issues in timeline variability? First, though some features can be bound at several moments during the life-cycle, the configuration interfaces tend to be different for each moment. For example, in the case of the Linux kernel, a module may be included at build time through the use of an interactive configuration tool that shows variants, dependencies between features, and so on. On the other hand, including a module at runtime happens by running the modprobe command; an entirely different interface. Likewise, Apache modules can be added at build time through a Autoconf configure script, or at startup time by editing a configuration file.

Second, the techniques used to implement timeline variability are *ad hoc* necessarily because the underlying languages do not offer the required support. Providing a variation point *either* at build time *or* at runtime is not hard, but providing it at both requires quite a bit of "magic".

3. Future Work

We have seen that timeline variability causes difficulties at two different levels, namely, in the *implementation* and in the *configuration* of the system.

Implementation The main implementation issue is that variation points are not first-class citizens in conventional programming languages and development environments, that is, they are not represented explicitly and cannot be manipulated directly. Rather, the implementation of a variation point happens through some mechanism that is specific to the binding time, e.g., conditional compilation or dynamic loading of shared libraries. This means that moving a variation point to a different binding time, or supporting binding at multiple binding times, requires explicit and often non-trivial modification to the system.

A partial solution to this problem is the use of *staged compilation*. For example, partial evaluation may be used to move an apparent runtime variation point to build time.

The converse—moving from build time to runtime—is generally harder. For example, it is not obvious how to deal with conditional data structure definitions.

Configuration The main problem here is that every stage in the life-cycle tends to present a different configuration interface to the user. This is particularly annoying for variation points that have several binding times. In the *Transparent Configuration Environments* (*TraCE*) project we aim at generalising system configuration interfaces. TraCE consists of a generic configuration interface parameterised with a formalised feature model.

In approaches such as FODA [4] feature models are described as graph-like structures, where the edges between features denote certain relationships such as alternatives and exclusion. The model therefore describes a set of *valid* configurations that satisfy all constraints on the feature space. Apart from being used during analysis and design, such models can also be used to drive the configuration process directly. For example, the CML2 [5] language was designed to drive the configuration process of the Linux kernel on the basis of a formal feature model of the system.

However, these models provides a *static* view of the configuration space: a configuration is either valid or it is not; no timeline aspects are taken into account. In order to model timeline aspects, it is necessary to take into account that some feature selections, i.e., bindings of variation points, are valid only on certain points on the configuration timeline. Therefore, the feature model presented in this section does not place constraints on configurations, but rather on transitions between configurations.

Formally, a feature model for a system with a statically fixed set of variation points has the following elements:

- A set of named variation points P and, for each variation point $p \in P$, the set of named states S_p.
- A *configuration* C is a mapping from variation points to states, that is, a function $P \rightarrow \cup_{p \in P} S_p$.
- An initial configuration $c_0 \in C$.
- A relation $T \subseteq C \times C$ expressing valid configuration transitions; i.e., it constrains configurations. As noted above, it is not sufficient merely to describe valid configurations, since not every valid configuration can be transformed into any other valid configuration. However, the set of valid configurations follows by computing the transitive closure of the set $\{c_0\}$ under the T relation.

Note that static feature models such as FODA [4], FDL [**?**], and CML [5] can be transcoded into this model; they are just different ways of expressing the valid-transition relation T. Indeed, the main problem in making this approach useful is to find a suitable way to specify T. Note that this is just a usability issue; the model is as described above.

It may be argued that implementation restrictions should not appear in the feature model (e.g. in [1], p. 117). However, they are required to generate configuration systems. In addition, we can identify several types of constraints. First, there are constraints that are inherent to the problem domain; these arise from the domain analysis. Second, some constraints result from implementation restrictions. This may well be the largest set in typical systems. Finally, some constraints are not forced by the domain or implementation, but rather are added by some stakeholder. (for example, a system administrator restricting some end-user configurability). The specification language for the feature model should allow these constraints to be specified separately.

4. Conclusion

Timeline variability makes the configuration of software systems more flexible by leaving open the decision about the binding time of a feature. However, the implementation of timeline variability is often ad hoc and presented through inconsistent configuration interfaces. Better support for timeline variability requires features models to describe the variability of a system *including* its timeline variability, and *transparent configuration environment* which provide an abstract interface to the details of configuration mechanisms required

References

[1] K. Czarnecki and U. W. Eisenecker. *Generative Programming — Methods, Tools, and Applications*. Addison-Wesley, June 2000.

[2] L. Geyer and M. Becker. On the influence of variabilities on the application-engineering process of a product family. In G. J. Chastek, editor, *Proceedings of the Second Software Product Line Conference (SPLC2)*, volume 2379 of *Lecture Notes in Computer Science*, August 2002.

[3] M. de Jonge. Source tree composition. In *Proceedings: Seventh International Conference on Software Reuse*, volume 2319 of *LNCS*, pages 17–32. Springer-Verlag, Apr. 2002.

[4] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[5] E. S. Raymond. The CML2 language: Python implementation of a constraint-based interactive configurator. In *9th International Python Conference*, March 2001.

[6] J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Proceeedings of WICSA 2001*, August 2001.