Capturing Timeline Variability with Transparent Configuration Environments

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

Merijn de Jonge
CWI, P.O. Box 94079,
1090 GB Amsterdam, The Netherlands
M.de.Jonge@cwi.nl

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

Abstract

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 deployment life-cycle. Some features can be bound only at specific moments in the life-cycle, while some can be bound at* several *distinct moments (*timeline variability*). This leads to inconsistent configuration interfaces; variability decisions are generally made through different interfaces depending on the moment in the life-cycle. In this paper we propose to formalize variability into a feature model that takes timeline issues into account and to derive from such feature models configuration interfaces that abstract over the life-cycle.*

1. Introduction

Managing the variability in software systems is rapidly becoming an important factor in software development. Instead of developing and deploying a "fixed" one-of-kind system, it is now common to develop a family of systems whose members differ with respect to functionality or technical facilities offered [4]. As a simple example, consider a software development environment that is delivered in a light, professional, and enterprise version, each providing increasing amounts of functionality. As another source for variability, modern systems need to run on different computing platforms and provide a user interface in different natural languages and possibly interaction styles. Finally, systems typically offer extensive means for configuration and customization during installation, startup, and run-time. Again, this extends the space of actual systems of the family.

An important reason for explicitly introducing variability into a system is to obtain reuse of software. Building a separate system for each variant means that the overall development effort and time will increase, and that time to market will be seriously affected. In addition, having multiple systems with significant overlap among them seriously affects the programming and management effort needed in maintenance.

Variability is studied, among others, in the field of system families or software product lines [2]. A common approach found there is to explicitly identify features that are common to a family of products, or specific for some of its members, and to organize them in a feature model. A feature then represents a variation point in the system for which multiple choices or variants can be made available. For each variation point, a particular variant may have to be selected to actually use the system.

A feature model specifies the variability in a system on a conceptual level. It is typically used as a basis for the design and implementation of family-common and product-specific assets. However, since feature models are not first-class citizens in development environments, the link from variability on a conceptual level to the actual implementation typically has to be maintained manually and frequently is not available at all [14].

Timeline variability To create an implementation, the feature model itself is not enough, because the timing characteristics of variation points [5] play a prominent role in the design process. For each variation point, we have to ask ourselves when (in the development, deployment and usage timeline) it should be possible to extend or reduce the set of variants and when the variation point should be bound to a particular choice. Typical examples of such decision moments are compilation-time, distribution-time, build-time, installation-time, start-time, and run-time. To identify such moments, it is necessary to consider the needs of various parties or stakeholders that might be involved in the decision process — e.g., designer, coder, product manager, system administrator, end-user.

From such an analysis it may follow that it should be possible to make a decision at *several* moments on the timeline. For instance, if we consider an operating system it may be desirable to statically link a driver into the kernel, but also to have the opportunity to load it at start-up time. This *timeline variability* is an extra dimension to variability that is often ignored. In current practice, the issues of the timeline, decision moments, stakeholders and timeline variability are not considered, nor modeled explicitly. As a result, variability is often not orthogonal to the timeline and the variability of a system appears to have been designed in an ad-hoc fashion. Some features can be configured at install-time, others at start-up time, and still others at run-time. Moreover, the *binding time* [5] of variability is usually fixed and cannot be altered, e.g., binding a run-time variation point at installation-time.

A typical scenario is a system that supports bundling of a selection of packages from a package repository. Configuration of the packages with file system layout information is done after package selection and distribution. However, it might be desirable to configure packages *before* bundling, for example, for mass deployment on machines with a standardized file system layout. Such alternative orders for making variability decisions typically require a completely different set-up.

Configuration mechanisms The realization of variability decisions can be achieved using a wide variety of *configuration mechanisms*. For example, conditional compilation allows us to choose a particular variant during compilation by identifying whether or not a piece of code should be compiled. Likewise, we can transform existing code to introduce particular behavior. Source tree composition on the other hand, allows us to include or exclude particular (source) components [7]. For more late-time variability we can use object-oriented techniques like inheritance and abstract-coupling combined with a factory object and a parameter file that defines the correct variant to use. Alternatively, we can use run-time discovery and binding of (distributed) objects in platforms such as Corba. In practice, there is a significant distance between variability on a conceptual level (features, variation points) and the configuration mechanisms actually used.

Configuration interface A software system with variability provides a *configuration interface*, through which variability decisions are made. Ideally this interface provides a view to the system that corresponds to variability at the conceptual level (i.e., the feature model). However, the underlying mechanisms are usually reflected in the interface to such an extent that the high-level view is obscured by low-level mechanisms. Since variability decisions are realized through many different configuration mechanisms, which are closely tied to moments in the timeline, variability appears to be treated in an ad-hoc fashion. A particular mechanism is chosen arbitrarily, or for technical reasons, rather than for support of an appropriate configuration interface. As a result configuration is not transparent, but determined by the time of configuration.

Implementation of timeline variability Another issue of current practice and mechanisms is that changing the timing-characteristics of a variation point involves a lot of work, typically because the implementation of the variation point is scattered across many different artifacts (source code files, build files, et cetera). For example, allowing "pre-binding" of run-time variation points during installation (e.g. making a partially parameterized version of an X Window System server configuration) may involve a lot of work in different parts of the system. Finally, the consequences of a particular choice, e.g., in terms of performance, resource overhead, or maintainability, are often unclear or not considered. As a consequence, potential techniques to optimize the software at a particular binding time may not be fully used.

Contribution In this paper we demonstrate the problem of timeline variability and the configuration issues related to variability in general. The central idea is to provide a general formal model of variability that can cope with timeline aspects. Such a model can be annotated with *actions* to perform configuration transitions depending on the configuration state of the system (i.e., the "moment" on the timeline). From such a model we can generically derive configuration interfaces that close the current gap between variability at the conceptual and implementation levels.

Outline In section 2 we provide some concrete examples of timeline variability and their impact on the developers and users of a system. In section 3 we describe a general model of feature models. We describe in section 4 how configuration interfaces can be obtained generically from the feature models by annotating the models with actions. We discuss related work in section 5. Concluding remarks and directions for future work are given in section 6.

2. Motivating Examples

In this section we show some examples of variability in real systems. In particular we are interested in the impact of variability on configuration of the system, and in the presence and implementation of *timeline variability*—the phenomenon that certain features may be selected at *several* different moments on the timeline.

The Linux kernel The Linux kernel provides the basis for several variants of the GNU/Linux operating system. 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 loaded 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 realized through the module system is for the most part straightforward. For example, operations on 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*. That is, they must be made known to the system, and this presents difficulties. Slightly simplified, every module exports an initialization function f which must be called during kernel initialization, in the case of statically linked modules, or at module load time, in the case of dynamically loaded modules.

For dynamically loaded modules, obtaining the address of f is a matter of looking it up in the module's symbol table at load time. For statically linked modules, the problem is harder, since the C language 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. This problem is solved by emitting these addresses in a specially designated *section* of the executable image, which can then be iterated over at runtime. The point here is not to show the details of the implementation of timeline variability in the Linux, but rather to show that it is non-obvious and quite different at each point on the timeline. In this case, we achieve timeline variability of module activation at build time and runtime, through a combination of preprocessor, compiler, and linker magic.

Another issue is how variability appears to the user. A problem with systems that allow configurability at different moments on the timeline is that the configuration interface tends to be different at each conceptual moment. For example in the case of the Linux kernel, modules are added at build time through an interactive tool that allows variation points to be bound through a textual or graphical user interface based on a feature model of the kernel. On the other hand, at runtime the interface is more primitive: adding a module happens through commands such as `modprobe` which simply takes the name of a module to be loaded.

The Apache web server 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 at build time, 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 (that is, 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. Note that this solution is again, in a sense, outside of the C language; we need to *generate* C code (a process external to the language proper) in order to deal with these open variation points. Registering a module at startup time happens by loading the module and lookup up a fixed name in its symbol table.

Configuration is quite different at build time and startup time. At build time, modules are selected by specifying the list of desired modules to an Autoconf configuration script (which constructs the build files). If modules are added later, at startup time, they must be added to a configuration file (`httpd.conf`).

Issues The aforementioned examples demonstrate the two main issues in implementing timeline variability. First, we tend have to *a different configuration interface per configuration moment*, even when some features can be bound at several moments during the life-cycle (thus presenting an inconsistent interface to the user).

Second, *implementations techniques are* ad hoc. This is almost necessarily so, 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 tends to require some hackery. Consider, for example, a binary variation point that is bound at runtime, implemented in C. This might be implemented as follows:

```
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 to allow for this feature to be bound both at build time and runtime, we would need, e.g.,:

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

which is inelegant: not only do we need two different implementation mechanisms, but binding the feature will tend to happen through different configuration interfaces.

3. Feature models and time

The main problem in timeline variability 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 order to generalize system configuration, we propose that a generic configuration interface is parameterized with a formalized feature model.

In approaches such as FODA [8] or FDL [15] feature models are described as graph-like structures, where the edges between features denote certain relationships (such as alternatives, exclusion, and so on). 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 [12] language was designed to drive the configuration process of the Linux kernel (and other systems) 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. That is, we should not place constraints on configurations but on transitions between configurations.

Formally, a feature model for a system with a statically fixed set[1] 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 an assignment of states to variation points, 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 is the transitive closure of $\{c_0\}$ under the T relation.

[1]It is possible for the set of variation points to be dynamic, e.g., loadable modules may add their own variability. For simplicity we do not take this possibility into account here.

Note that static feature models such as FODA, FDL, and CML 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 an usability issue; the model is as described above.

It may be argued that implementation restrictions should not appear in the feature model. 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. An example would be a system administrator who restricts some end-user configurability. The specification language for the feature model should allow these constraints to be specified separately.

Example An example may be useful. We shall encode a very small subset of the variant space of the Apache web server using the formalism given above.

What is the set of variation points P and the associated sets of states for each variation point? An example of a simple variation point in this model is `debug` to enable or disable emission of debug information (with states `on` and `off`, respectively). This variation point can only be bound at build time. More relevant to timeline variability is Apache's module support. For example, we have variation points such as `mod_cgi` (also with states `on` and `off`) to enable or disable support for CGI scripts, respectively. Recalling the discussing of modules in Apache in section 2, such features can always be bound at build time, but they can only be changed at startup time when support for dynamic loading of modules is enabled. This is also a variation point, of course, which we denote as `mod_dso` (for *dynamic shared objects*).

In order for our approach to work we need to encode in the valid-transition relation T that the state of `mod_cgi` can be changed up to build time, but up to startup time only if `mod_dso` is set to `on`. Hence, we need to be able to distinguish the point on the timeline that we are at. To do this we introduce a *pseudo variation point* `time` with states `initial`, `built`, and `running`, denoting the deployment points at which the source has been obtained, the system has been built, and the system has been started.

Note that there is nothing particularly special about `time`, except that it does not denote a real (i.e., conceptual) variation point; i.e., this is quite general: any aspect of the deployment state (such as an installation path) can be stored in the configuration.

Using `time` we can fill in T, which describes the set of valid transitions. Hence, we have to deal with *two* configurations: the configuration c_1 we are coming from, and the configuration c_2 that we are going to. For any c_1 and c_2, $(c_1, c_2) \in T$ if and only if the following hold:

(1) $c_1.\texttt{time} \leq c_2.\texttt{time}$
(2) $c_1.\texttt{time} \geq \texttt{built} \rightarrow c_1.\texttt{debug} = c_2.\texttt{debug}$
(3) $c_1.\texttt{time} \geq \texttt{built} \rightarrow c_1.\texttt{mod\_dso} = c_2.\texttt{mod\_dso}$
(4) $c_1.\texttt{time} \geq \texttt{built} \wedge c_1.\texttt{mod\_dso} = \texttt{off}$
$\rightarrow c1.\texttt{mod\_cgi} = c_2.\texttt{mod\_cgi}$
(5) $c_1.\texttt{time} \geq \texttt{running}$
$\rightarrow c_1.\texttt{mod\_cgi} = c_2.\texttt{mod\_cgi}$

(The ordering on `time` is `initial` $\leq$ `built` $\leq$ `running`). Condition (1) encodes that time is monotonically non-decreasing. Conditions (2) and (3) specify that the `debug` and `mod_dso` variation point can never change after the system has been built. On the other hand, condition (4) says that `mod_cgi` cannot change if, additionally, support for dynamic loading is disabled. Hence, `mod_cgi` *can* change after build time if DSO support is enabled. Finally, condition (5) restricts this a bit: CGI support cannot be changed after the system has been started.

4. From models to configuration interfaces

The construction of a formal feature model as discussed in the previous section is valuable in itself because it enables analysis of both conceptual *and* implementation-defined variability in a system. The real strength of a formal model, however, is that it allows the automatic generation of configuration interfaces. The intent is that using the feature model we can drive a generic configuration tool called *TraCE* (for *Transparent Configuration Environment*). The idea is outlined in figure 1. At each point in time we maintain the configuration state corresponding to the state of the system. Starting with an initial system (e.g., the source code distribution of Apache), the user can make modification to the configuration through the TraCE user interface, which presents a visualization of the feature model. Such modifications are actualized upon the system by TraCE. For example, in the Apache example, the transition from initial

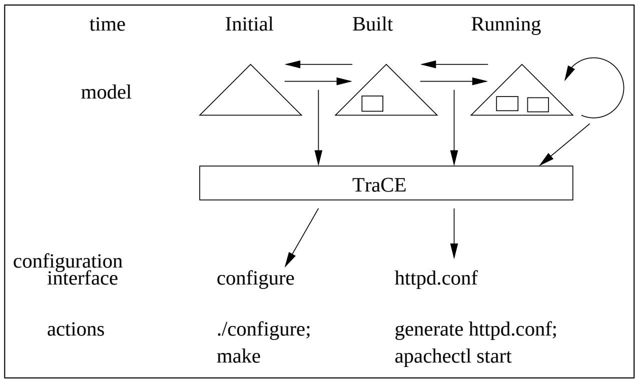

Figure 1. Sketch of the TraCE system operating on a feature model for Apache.

to built stage is performed by configuring the source with the right parameters (depending on the selected variation points) and building it.

More precisely, given a *current* configuration $c \in C$, the user can modify c by changing the states of variation points, yielding a *target* configuration $c' \in C$. We associate with each valid transition $t \in T$ some imperative action that should be performed to *realize* the configuration transition. Hence, if $(c, c') \in T$, the configuration c' can be realized by executing the associated action. Note that actions are associated with transitions, and so configurations may be realized in different ways depending on the configuration we are coming from. This is necessary for supporting timeline variability, since the binding of variation points can proceed through different implementation points depending on the time, or on the state of other variation points.

A problem here is that while (c, c') may not be a valid transition, there may be a sequence of transitions $(c, c_1), (c_1, c_2), \ldots, (c_n, c') \in T$ that realizes the desired transition. Finding a path in the transition space is computationally prohibitive. We can side-step this problem by requiring that the user always specifies transitions that are in T. This is not unreasonable if the developer of the feature model ensures that T is (more or less) transitive, that is, $(c_1, c_2) \in T \wedge (c_2, c_3) \in T \rightarrow (c_1, c_3) \in T$.

5. Related work

Variability is an emerging area of research. The first attempts to handle variability in a disciplined way are the feature-modeling formalisms originally developed in [8]. These models are directed at domain analysis, however, and are not directly used for implementation. Rather, such models suggest where in the system the implementor should construct variation points to deal with anticipated or unanticipated variants. In [16] another feature modeling is addressed, which uses feature logic to reason about collections of components and their properties. Basic support for timeline variability is addressed in [11]. They use partial evaluation techniques of components parameters to choose between compile-time and run-time variability. Variability mechanisms are described in [5]. They introduce the notion of variability binding time (i.e., the moment in time where a variability point is bound) but binding time is not explicitly modeled nor transparently handled. Feature binding cannot be rolled back in product instances to change parts of its functionality. Variation management in software product lines is discussed in [10]. They discuss variation during the life-time of a product line rather than during the deployment time of a product instance.

Several techniques have been developed to realize variability at compile-time, such as Frame Technology [6], Mixin layers [13], and aspect-oriented programming [9]. None of these techniques explicitly model variability. GenVoca is another compile-time variability mechanism [1]. Feature modeling in combination with GenVoca is briefly addressed in [3] but does not take timeline variability into account.

6. Conclusion

We have discussed some of the issues in timeline variability. We suggest that the problem of inconsistent configuration interfaces can be solved through formal feature models that encode timeline aspects and that these can be used to generically drive the configuration process.

We are currently implementing a prototype of TraCE. The are several important issues that must be addressed. First, we need a language (or interface) that allows the efficient formulation of feature models, as well as the association of actions to transitions. Second, there are user interface issues. For instance, how do we present the feature space to the user? In formalisms such as CML2 or FDL the presentation structure is more-or-less obvious (due to the use of an essentially tree-like model structure). In TraCE we need to automatically derive an appropriate presentation structure from the feature model.

The other main problem in timeline variability—implementation techniques—deserves study; e.g., language mechanisms and programming techniques that allow easier binding at several moments must be investigated.

References

[1] D. Batory, C. Johnson, B. MacDonald, and D. von Heeder. Achieving extensibility through product-lines and domain-specific languages: a case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):191–214, 2002.

[2] J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

[3] K. Czarnecki and U. W. Eisenecker. Components and generative programming. In O. Nierstrasz and M. Lemoine, editors, *ESEC/FSE '99*, volume 1687 of *LNCS*, pages 2–19. Springer-Verlag / ACM Press, 1999.

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

[5] J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In R. Kazman, P. Kruchten, C. Verhoef, and H. van Vliet, editors, *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, pages 45–54. IEEE, 2001.

[6] S. Jarzabek and R. Seviora. Engineering components for ease of customization and evolution. *IEE Proceedings – Software*, 147(6):237–248, Dec. 2000. A special issue on Component-based Software Engineering.

[7] M. de Jonge. Source tree composition. In *Proceedings: Seventh International Conference on Software Reuse*, volume 2319 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[8] 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.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[10] C. Krueger. Variation management for software production lines. In G. J. Chastek, editor, *Proceedings of the Second Software Product Line Conference (SPLC2)*, volume 2379 of *Lecture Notes in Computer Science*, August 2002.

[11] R. van Ommering. Configuration management in component based product populations. In *Tenth International Workshop on Software Configuration Management (SCM-10)*, 2001.

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

[13] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255, 2002.

[14] A. van Deursen, M. de Jonge, and T. Kuipers. Feature-based product line instantiation using source-level packages. In G. J. Chastek, editor, *Proceedings of the Second Software Product Line Conference (SPLC2)*, volume 2379 of *Lecture Notes in Computer Science*, August 2002.

[15] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. Submitted.

[16] A. Zeller and G. Snelting. Unified versioning through feature logic. *ACM Transactions on Software Engineering and Methodology*, 6(4):398–441, Oct. 1997.