TraCE: Transparent Configuration Environments

Eelco Dolstra, Gert Florijn, Merijn de Jonge, and Eelco Visser

1 The Problem: Variability

1.1 The phenomenon

Almost all software systems exhibit **variability**. For example:

- Building an operating system kernel with or without multiprocessor support.
- Building a "light" or "professional" version of a word processor.
- Enabling or disabling support for some piece of hardware in an operating system.

That is, systems have **variation points** that can be bound to several **variants** to implement variant behaviour. Binding can take place on several **moments** on the implementation and deployment **timeline**:

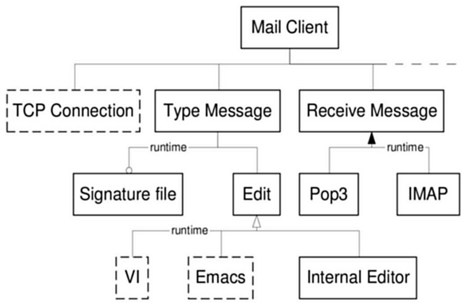
- At **design time** (e.g., we can choose (not) to implement some feature).
- At **build time** (e.g., through the C preprocessor).
- At **packaging time** (e.g., by selectively including components in an RPM package).
- At **installation time** (e.g., the Microsoft Office installer allows components to be (de)selected).
- At **startup time** (e.g., the Apache webserver can load extensions when it's started).
- At **runtime** (e.g., through dynamic loading of plug-ins, or plain `if-then-else` constructs).

1.2 The issues

The **implementation mechanisms** are **ad hoc**: if-then-else, conditional compilation, partial evaluation, dynamic linking, selective file inclusion, and so on. Worse, the corresponding **configuration mechanisms** are similarly ad hoc.
This is particularly annoying for those features that are **timeline variable**, that is, that can be bound at **several** different moments on the timeline. An example is the Linux module system; modules can be enabled both at build time and at runtime.

2 The Solution: Feature Models

In order to provide a **consistent configuration process** we need a formalisation of the variability present in the system.
Classical **feature models** don't cut it:



(Source: Svahnberg et al.)

The problem is that these describe a static set of **valid configurations**. To model timeline aspects we need to take into account where we came from and where we are going to.

2.1 Model

That is, we need to describe **valid transitions between configurations**. Formally:

- 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 $P \to \cup_{p \in P} S_p$.
- An initial configuration $c_0 \in C$.
- A relation $T \subseteq C \times C$ expressing valid configuration transitions; i.e., it constrains configurations. (The set of valid configurations is the transitive closure of $\{c_0\}$ under the T relation.)
- A mapping from $t \in T$ to imperative actions that **realise** the transition. This allows generic configuration tools to be constructed.

2.2 Example: Apache

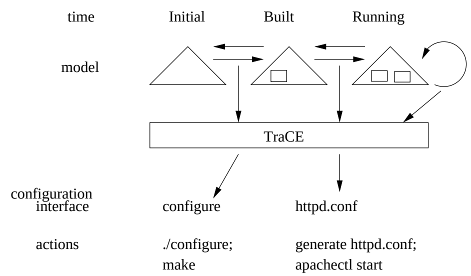
(1) $c_1.\mathtt{time} \leq c_2.\mathtt{time}$
(2) $c_1.\mathtt{time} \geq \mathtt{built} \to c_1.\mathtt{debug} = c_2.\mathtt{debug}$
(3) $c_1.\mathtt{time} \geq \mathtt{built} \to c_1.\mathtt{mod\_dso} = c_2.\mathtt{mod\_dso}$
(4) $c_1.\mathtt{time} \geq \mathtt{built} \land c_1.\mathtt{mod\_dso} =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

3 The TraCE Project

TraCE (an NWO Jacquard project) aims to provide the following:

- **Transparent configuration environments** that allow generic and consistent configuration of software systems at all moments on the deployment timeline.
- This requires **models of variability**, such as given above; we will develop a Variability Modeling Language (VML) to formalise these.
- **Component composition** mechanisms that take variability in components into account.
- Generalise variability **implementation techniques** such as partial evaluation.
- Variability **discovery** and refactoring (for "legacy" systems).

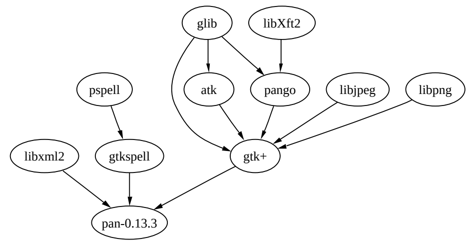
3.1 Architecture



4 Nix

Nix is a package manager, deployment system, and component glue in one. It's an initial prototype for TraCE that does not yet deal with timeline variable features, but does take timeline aspects into account. There are many problems with deployment, such as:

- Dependency hell:



- Versioning, consistency, and upgrade problems.
- Source vs. binary deployment.
- Reproducibility: if a system works in one environment, will it work in another?

Solution: require that packages are built in **isolation**, and can only use other packages at build- or runtime by explicitly declaring them as dependencies in a **descriptor**. Solve versioning issues by using cryptographic **hashes** of sources and descriptors.

```
Descr(
  [ Bind("atk", Pkg("99f2..."))
  , Bind("build", File("2a4e...")), ...
  , Bind("spellCheck", Bool(True))
  , Bind("src",File("7beb..."))
  , Bind("system",Str("...-linux"))])
```

The hash of a package therefore completely and uniquely identifies the package and its entire dependency tree.

Such descriptors are not user-friendly, but they can be generated automatically from high-level formalisms such as **Fix**. This will allow the user to, e.g., graphically instantiate a system by selecting variants for all variation points.