Consistency checking in an infrastructure for large-scale generative programming

A. Rauschmayer, Alexander Knapp, Martin Wirsing

Consistency Checking in an Infrastructure for Large-Scale Generative Programming[*]

Axel Rauschmayer, Alexander Knapp, Martin Wirsing
Institut für Informatik
Ludwig-Maximilians-Universität München

E-mail: {rauschma,knapp,wirsing}@pst.ifi.lmu.de

Abstract

Ubiquitous computing increases the pressure on the software industry to produce ever more and error-free code. Two recipes from automated programming are available to meet this challenge: On the one hand, generative programming raises the level of abstraction in software development by describing problems in high-level domain-specific languages and making them executable. On the other hand, in situations where one needs to produce a family of similar programs, product line engineering supports code reuse by composing programs from a set of common assets (or features). AHEAD (Algebraic Hierarchical Equations for Application Design) is a framework for generative programming and product line engineering that achieves additional productivity gains by scaling feature composition up. Our contribution is GRAFT, *a calculus that gives a formal foundation to* AHEAD *and provides several mechanisms for making sure that feature combinations are legal and that features in themselves are consistent.*

1. Introduction

As manufacturers turn almost every household device into a computer, they increase the pressure on the software industry to produce ever more robust and bug-free code. Two approaches in automated programming are aiming to meet this challenge: First, *generative programming* [11] raises the level of abstraction by allowing the creation of *domain-specific languages* (DSLs, [21]). The idea is to express the solution to a problem in the terms of the problem domain which eases understanding, maintenance and the efficiency of programming. It is the responsibility of the generative programming infrastructure to provide a way to make a DSL executable (usually by generating code of a lower-level implementation language). Second, *product line engineering* [22] comes from the observation that in cases where we do not want to create a single program, but a *family* of related programs, work is often duplicated. An example of a product line is the software for a series of cell phones; very little changes with each model. Accordingly, product line engineering is about producing an instance of a program from a set of common assets (or *features*). AHEAD (*Algebraic Hierarchical Equations for Application Design*, [5, 6]) is a framework for generative programming and product line engineering that solves several problems in this area: As a typical software system comprises a mixture of artifacts (written in human language, programming languages, domain-specific languages etc.), AHEAD manages the synthesis of *all* of them. It does so by keeping the artifacts of a feature in a tree, produces members of a product line by tree composition and delegates artifact-specific generation tasks to specialized generators. The tree-based approach has the advantage of scaling easily and of adding expressiveness when laying out the contents of a feature.

The subject of this work is GRAFT, a calculus that provides a theoretical foundation for the current incarnation of AHEAD[1]. The clear and formal definition of what an AHEAD feature is allows us to perform a variety of consistency checks for a set of features that is to produce a program: Are the features structurally compatible; e. g., does a feature add something that has already been added before? Does this combination of features make sense according to internal semantic constraints? Are existential dependencies between resources fulfilled? In addition to just checking these conditions, GRAFT can also be used as a diagnos-

---

1. Note that at the very core, AHEAD is based on a theoretical model that is more generic than GRAFT (namely, nested functions and constants). But this does not currently show in practical applications which are completely describable in GRAFT and easier to analyze that way.

tic tool: An inferred interface of a feature gives a detailed account of what the feature needs and what it offers. Just like the checks, it describes structural, semantic and dependency information.

The next sections are structured as follows: In Sect. 2, we give a short introduction to AHEAD. Sect. 3 presents *design rules*, the current approach to consistency checking used in AHEAD. AHEAD data structures and design rules can be encoded in GRAFT, as shown in Sect. 4. With this foundation, GRAFT allows us to perform several analyses (Sect. 5). In Sect. 6 we sketch soundness proofs. We end by mentioning related work (Sect. 7) and by drawing conclusions about present and future work (Sect. 8).

2. AHEAD

A finished program can be viewed as being composed from a set of *features*, where a feature is all the code in the program that implements one functionality (user-visible or otherwise). Unfortunately, in object-oriented programming (OOP), features very often cannot be properly encapsulated, because the code is *cutting across* classes which are the common unit of encapsulation in OOP. Readers familiar with aspect-oriented programming (AOP, [13]) will realize that we have used AOP language [15] to express this problem. In fact, those parts of AHEAD that are concerned with its solution can be viewed as simplified AOP. Additionally, we need to be able to build a program from an arbitrary subset of all available features if we are to do product line engineering. Finally, we want other artifacts (e. g., documentation) to be composed automatically, alongside the code. AHEAD generalizes OOP inheritance to achieve these goals. We'll start right away with a concrete example and then explain what general principles are at work.

Fig. 1 shows an AHEAD *model*, a set of features (for now, view them as some kind of cross-cutting code module) from which we can compose programs. The *base feature* `base` provides three classes. The *feature increments* `html` and `latex` add to these classes; we have used the Java-1.5-style meta-data (as defined in [7]) annotation `@addto` to make it clear that `html` and `latex` add to existing classes.

Feature increments thus add code to a set of classes in a way that is reminiscent of inheritance. But instead of the static link between a superclass and a subclass, a feature has no fixed external dependencies. Features can thus be flexibly assembled into a product line. Using a notation for expressing composition, we can define members of the product line represented by this model: The model is the set of features {base, html, latex}. Feature composition is performed by the diagrammatic composition operator ';' that composes from left to right (as opposed to the traditional functional composition operator denoted as '∘' that composes from right to left). One program we can define is

```
public abstract class Text {
}
public class PlainText extends Text {
  private String _str;
  public PlainText(String myStr) {
    _str = myStr;
  } }
public class BoldText extends Text {
  private Text _text;
  public BoldText(Text myText) {
    _text = myText; } }
```

(a) Feature `base`

```
@addto public abstract class Text {
  public abstract String toHTML();
}
@addto public class PlainText extends Text {
  public String toHTML() { return _str; }
}
@addto public class BoldText extends Text {
  public String toHTML() {
    return "<b>"+_text.toHTML()+"</b>"; } }
```

(b) Feature `html`

```
@addto public abstract class Text {
  public abstract String toLatex();
}
@addto public class PlainText extends Text {
  public String toLatex() { return _str; }
}
@addto public class BoldText extends Text {
  public String toLatex() {
    return "\textbf{"+_text.toLatex()+"}";
  } }
```

(c) Feature `latex`

Figure 1: The source code of three AHEAD features. Feature `base` provides the three classes `Text`, `PlainText` and `BoldText` that define the skeleton of a nested rich text representation. This basic functionality is only about storing the information. Features `html` and `latex` add to the classes and implement an export of the internal data to HTML and LaTeX.

base;html, a text representation that can be converted to HTML. Other possibilities are base;html;latex and base (which is just a container that cannot be converted to anything).

The general data structure used in AHEAD is the tree: Features are represented as trees and feature composition is expressed as tree merging. Object-oriented programs lend themselves especially well for a tree representation, as exemplified by the features base and html (Fig. 2): Each level of namespace nesting (classes, methods) shows up as a level in the trees. Note that we do not explicitly represent the inheritance relation, it is only kept as part of the
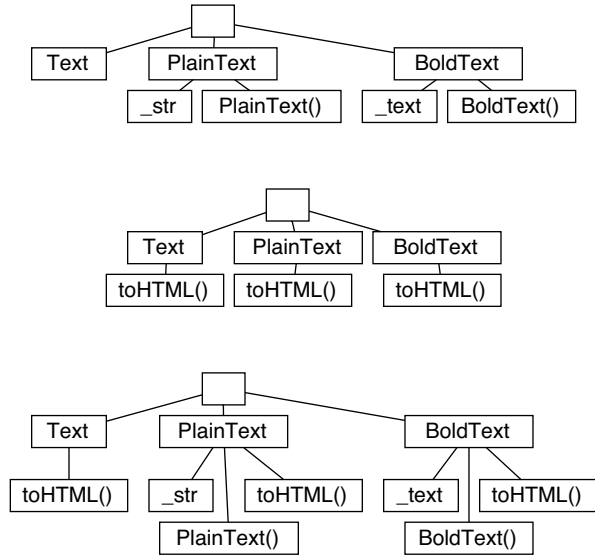
Figure 2: Features `base` (top) and `html` (middle) represented as trees: The root node is an anonymous container for the class nodes `Text`, `PlainText` and `BoldText` that in turn contain fields and methods. Composing the features leads to the combined tree `base`; `html` (bottom).

Figure 3: Documentation has been added as a node named "doc" to the features `base` (top) and `html` (bottom). Merging the features using the composition algorithm means that the documentation nodes are correctly being merged by a documentation-specific plug-in.

node data. Each node has a name and contains data (in this case, source code). A path of names precisely defines the location of a node in a tree. One merges the two trees into a combined tree `base`; `html` by applying the following simple algorithm to tree `base`:

- Iterate over all nodes h in `html`. p is the path of names to h.
  - If there is already a node b in `base` whose path is p, we need to merge the two nodes. We say that h *refines* b.
  - Otherwise, h is added as a new node at the location p.

We have kept the root nodes of features anonymous; otherwise the root node would have been an exceptional case in the algorithm. The algorithm does not specify how exactly nodes are to be merged but instead delegates refinement to generators (plug-ins, if you will) that are specific to the data that is stored in the node. That is, node merging is polymorphic. The following additional three cases are handled correctly by this approach and show how generic it is: First, method overriding in OOP is related to refinement. If two trees have a method node with the same name, method-specific merging does the right thing. Second, we can add further namespace nesting to the trees, a feature could for
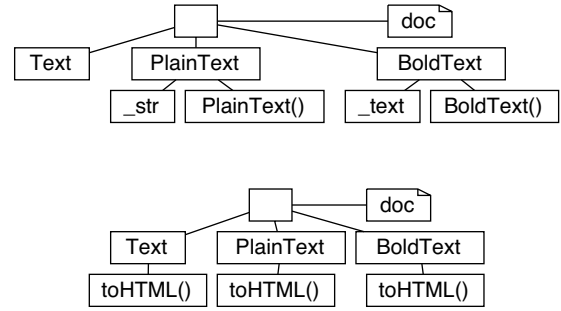
example consist of several packages. Third, the data we store in our nodes is not restricted to source code, it can be practically anything. We could, for example add documentation nodes to `base` and `html`, both call them `doc` and the composition algorithm would synthesize documentation in parallel to the source code (Fig. 3). Similarly, we can handle all kinds of DSLs. In practice, every AHEAD feature is defined by a directory tree in the file system. Leaves are usually files, but might also be saved in an aggregated form (for example, one class per source file). For further explanations and the deposition of other AHEAD ideas such as automatic optimization of feature selection and meta-models, we refer to [5].

3. Design Rule Checking

Order in AHEAD feature composition is significant, leading to many possible feature combinations, not all of which make sense: There are some that produce programs that are syntactically incorrect. In AHEAD, generators signal an error to prevent such cases. But there are also combinations that lead to syntactically correct yet semantically incorrect programs. A simple example is that we might consider a program incomplete if it is unable to print. But, if no other part depends on this feature, we'll be able to generate this program without printing. AHEAD uses *design rules* to express this kind of *semantic* condition. Every feature has a set of zero or more design rules associated with it that express its contribution to the program and the demands it makes on its surroundings. A contribution is specified by assigning a value to a so-called *property* (basically, a variable). For example, if a feature implements printing,

the design rule for expressing this contribution[2] is

$$\text{set havePrinting} = \text{true}$$

Checking for a contribution means checking if a property has a certain value. For example, the following design rule checks if printing is available:

$$\text{checknow havePrinting} = \text{true}$$

Note that the pattern that we have used for printing can be used for ensuring the existence of any kind of resource, be it a feature, a class, a method or something that is completely abstract. It is up to the programmer to give adequate names to properties and to set and check them correctly. There is one further complication: If a feature needs a contribution, that contribution can come from a feature that appears either before or after it in the feature sequence defining the program. Compare this to how a class can import behavior: it can either be inherited from a superclass or left to be implemented by a subclass, as an abstract method. Accordingly, design rule checking propagates properties and checks in two directions: From the first feature to the last and vice versa. This is similar to propagating inherited and synthesized properties up and down a parse tree in attribute grammars. As a final comment, AHEAD's universality allows it to neatly integrate design rules as just another kind of artifact: Like documentation in Fig. 3, a node with design rules is part of the feature that they describe. A special generator for this kind of node does the checking. For more details on design rules refer to [4, 3].

4. GRAFT: Trees and Design Rules

In order to be able to analyze an AHEAD feature or a feature composition, we need a formal definition of an AHEAD feature. We use a formal language called GRAFT to do so. We then also express design rules in GRAFT, making them part of the language (whereas in AHEAD, storing them in a node meant that they were external to the infrastructure). This leads to synergies when it comes to performing analyses, as shown in the next section.

4.1. Representing AHEAD trees in GRAFT

We initially use a notation for composition that distinguishes between base features and feature increments. We view base features as constants and increments as functions that we apply to them. This idea will help us to understand how GRAFT works. Later, a simple trick brings us back to the notation that we have used so far which does not

make that distinction. Applying first `html` and then `latex` to `base` is now written as `(base)html;latex`. We first show how GRAFT encodes increments as functions and then get to representing base features.

We have already seen that applying an increment f to a base b is like taking b as a starting point and then looping over every node in f: If the node is already present in b, we combine it with the existing node. Otherwise, this node is a new addition to b. We have thus encountered two operations that can be performed on a tree: refining and adding. But two more operations are typically used when manipulating trees: moving and deleting. In order to freely use and combine any of these four operations, GRAFT represents AHEAD increments as a sequence of commands, each one performing an addition, a delete, a move or a refinement. The node to operate on is given as a path of node names, the commands `add` and `refine` additionally get an argument containing the node (or rather, its data) that is to be added to the original tree. The last argument of every command is of course the tree that is to be modified. Let us look at the `add` command. A complete application is written as

$$\text{add } p \, [\texttt{code}] \dashv \{d\} \, t$$

This pattern is to be read as: At the location denoted by path p, add the node whose data is `code` to the tree t. We still have to explain one additional construct: the node we have just added *depends on* another node[3] d. For consistency checking, we want to express dependencies between nodes: If node x depends on node y, node y has to exist whenever node x exists. An example of a dependency is a method x that calls another method y. If the set of dependencies is empty, we completely omit that argument.

If we do not fill in the last argument t (a technique that is called *currying*), add becomes a function from trees to trees which corresponds to our intuition about feature increments being functions. To build sequences of commands, we just concatenate increments using the function composition operator ';'. The result is again an increment. Obviously, composing increments in GRAFT means composing blocks of code.

To represent base features in the same manner as increments, GRAFT views them as increments on empty trees. Accordingly, compositions always implicitly start with the empty tree and GRAFT programs never directly refer to trees. In Fig. 4 we use GRAFT command sequences to "implement" two features from the example AHEAD system defined in Fig. 1.

---

2   Here, we only look at boolean properties, while integer values can also be used in design rule checking. Support for these necessitates only a slight generalization of the mechanisms we'll present.

3   By abuse of notation, we write a singleton node $\{d\}$ where actually a set $\{d_1, \cdots, d_n\}$ of nodes can be used.

```
add Text [class Text];
add PlainText [class PlainText extends Text];
add PlainText._str [private String _str];
add PlainText.PlainText ⊢ {PlainText._str}
     [public PlainText(...) {...}];
add BoldText [class BoldText extends Text];
add BoldText._text [private Text _text];
add BoldText.BoldText ⊢ {BoldText._text}
     [public BoldText(...) {...}];
```

(a) Feature base

```
add Text.toHTML
     [public abstract String toHTML()];
add PlainText.toHTML ⊢ {PlainText._str}
     [public String toHTML() {...}];
add BoldText.toHTML ⊢ {BoldText._text, Text.toHTML}
     [public String toHTML() {...}];
```

(b) Feature html. That the *classes* are actually being refined is not made explicit., i. e., there are no commands that refine them.

Figure 4: Two features from our running example encoded in GRAFT. The tree arguments have been made implicit by currying.

4.2. Representing Design Rules in GRAFT

Integrating design rule checking in GRAFT means that we need to emulate two modes of design rule evaluation: first-to-last (property values are propagated from the first feature to the last and have corresponding checks) and last-to-first. We have actually already used GRAFT syntax for expressing design rules. Making these set and check statements part of a GRAFT program allows us to implement first-to-last evaluation, because that is the direction in which it is evaluated. Therefore, we have two new statements; for the time being, we only support equality checks:

set $p = v$	Assigns value v to property p
checknow $p = v$	Checks that p has value v

Whenever a feature uses a last-to-first check, it wants to make sure that a certain property (such as the existence of a resource) holds *after* everything has been composed, i. e, at the end of a GRAFT program. We can therefore reuse the existing (first-to-last) properties for last-to-first checks, if we have the ability to postpone a check until the end of the program. This also leads to a major simplification[4] of program

---

4 This simplification implies that GRAFT is not a superset of design rules any more, but in usual practical applications, this is not an is-

```
checknow baseIsThere = ⊥;
set baseIsThere = true;
checkfinal canExport = true
```

(a) Design rules for feature base: This feature should appear at most once (line 1 and 2) and demands that some kind of export be provided (line 3). Therefore, it is now illegal to use the feature base without having a feature that contributes export functionality.

```
checknow baseIsThere = true;
checknow htmlIsThere = ⊥;
set htmlIsThere = true;
set canExport = true
```

(b) Design rules of feature html: This feature depends on the existence of feature base (line 1), can appear at most once (line 2 and 3) and provides export functionality (line 4).

Figure 5: Design rules for the rich text system expressed in GRAFT.

evaluation, because we only need one pass. The following command performs the new kind of check:

checkfinal $p = v$	Checks that, at the end of the program, property p has value v

Accordingly, while this command can be introduced anywhere in the program, we wait until the end of the program and perform the check against the final value of the property p. Lastly, design rules provide the means to declare that a feature should only appear once. We support this assertion by a sequence of two commands: First, make sure that a property p has not yet been set. Second, set it to true. To perform this check for undefinedness with an equation, we allow the symbol '⊥' denoting "undefined" to appear on the right side:

checknow $p = ⊥$	Checks that property p is undefined

With these preparations, we can give examples of design rules expressed in GRAFT (Fig. 5). Sect. 5.1 gives a list of all GRAFT commands.

5. GRAFT: Checking

Let us quickly review what kinds of checks we would like to perform: We'd like to check for semantic consistency, for dependencies and for structural consistency. So far, we have only encountered semantic checks in the guise of design rules. We'll start by explaining how design rules

---

sue.

are evaluated and build on this foundation to enable the other two kinds of checks. This section concludes by outlining how interface inference is being done.

Design rule checking is centered around the idea of setting and checking properties. Therefore, GRAFT has a data structure called State for storing a set of *bindings*—(property name, value) pairs that assign values to properties. If a name does not appear at the left side of any of these bindings, we say that the value of the property is *undefined*. Thus, this set records the current state of the evaluation of a GRAFT program. Each of the *set* commands adds or changes a binding, each of the *check* commands queries State for a value. The special nature of final checks makes it necessary to keep them until the end of the program, when they can be evaluated. We provide a set called Final for that purpose.

We now use this foundation for reasoning about the presence or absence of nodes: If we make the path name p of a node a property, we can state the fact that the node is present by setting the property $p = $ true. Then, dependencies are much like final checks: If a node x depends on a node y, we must check that, at the end, node y is available. There are two special cases that we have got to take care of, though: If node x is deleted, we do not have to check for y any more. And, viewing the dependency as a link between x and y, we need to update it if either one of the two nodes is being moved. All this is accomplished by allowing *dependent checks* to be added to Final; checking for y is a check dependent on x. Operations like move need to update Final in order to keep the links current.

Structural consistency is about finding out whether an operation makes sense structurally. For example, before we can add a new node, we have to make sure that it is not already there. The wrong sequence of commands can therefore lead to a feature that is structurally inconsistent. We encode consistency (or executability) conditions as property checks and record the result of an operation by setting properties. We also include checks that make sure that we always produce well-formed trees.

Inferring the interface of a GRAFT program is practically done automatically, as a byproduct of consistency checking. Because we are keeping score of what nodes have been created, State is a summary of what a GRAFT program exports. Demands on subsequent features are all checks in Final that have not yet been fulfilled. Looking at first-to-last checks, we notice that each of these checks is actually a demand on a predecessor to provide something. If we cannot find an answer in State right away (be it positive or negative), that request is obviously still open. We record open requests in a set called Pre and that is the import interface of a program.

It is interesting to note that for none of these analyses, we needed the actual node data. That suggests to provide a skeletal GRAFT program, stripped of the source code, as public information about a feature. One can then check dependencies, design rules and structural consistency before actually performing the composition; the GRAFT skeleton acts as a type system. Unfortunately, due to the non-monotonous nature of GRAFT, the feature interface *on its own* is not sufficient for answering all questions about dependencies, even though it can be used for checking for all other conflicts: an interface does not say if a node a feature depends on can be provided by a predecessor or if it is deleted inside and thus can only be added by a successor. In the future, we might classify these two kinds of dependencies differently in the interface.

5.1. Formal Semantics of Checking

Property names. Property names p are a path of (node) names separated by the dot operator '.'. Large capital variable names such as X stand for single components. For example, $r.X$ is a path that starts with the sequence r of node names and has X as the last path component.

Constraint store. The result of analyzing a GRAFT program is the *constraint store*, a triple $(\text{Pre}, \text{State}, \text{Final})$ where

- Pre is the set of pre-constraints. These are conditions on properties that must be fulfilled prior to executing the program under analysis. A pre-constraint can be viewed as a guard which, if it holds, guarantees correct termination of the program. Pre is encoded as a set of equations, pairs written as $(p = b)$ where p is a property name and b is a boolean value (either true or false).

- State stores the current state, the values of the properties. If a property is set, it appears as a pair $(p = b) \in$ State. If its name does not appear on the left side of any pair in State, a property is said to be undefined. We sometimes use State as a total function that returns a property's value if it is set and the special symbol $\perp$, otherwise.

- Final is the set of post-constraints that are checked against the final "version" of State. Here we find conditions that must hold after the termination of a GRAFT program, such as "make sure that resource x has been added". Final generally contains *dependent checks*, an extension of the pair construct in Pre and State. These are encoded as a triple $(p = v \text{ depby } q)$ that means that the check $p = v$ depends on the presence of property q. If the check does not depend on any node, we write it as $(p = v \text{ depby } \perp)$.

Algorithm. The analysis comprises the following two steps: Note that each one of the steps mentioned above is a partial operation. Inability to perform any one of them can be seen

as a "compilation error", i. e., the consistency checks have failed.

Language layers. The Design of GRAFT is layered: The core of GRAFT are the design rule commands, including a few extensions to support the second layer, the commands used for tree manipulation.

Design Rules: Checking

Residuating check. This operation checks whether a binding in the state has a certain value. This operation can only be performed if the check does not contradict a binding in the state. If there is no corresponding binding then we keep the check as a Pre constraint (the check *residuates*).

$$(\mathrm{Pre}, \mathrm{State}, \mathrm{Final})\ \mathsf{check}\ p = b \equiv$$
$$\begin{cases} (\mathrm{Pre}, \mathrm{State}, \mathrm{Final}) & \text{if } \mathrm{State}(p) = b \\ (\mathrm{Pre} \cup \{(p = b)\}, \mathrm{State}, \mathrm{Final}) \\ \quad \text{if } \mathrm{State}(p) = \bot \end{cases}$$

Final check. The same check as above can be performed "finally", i. e., we have to add it to the final constraints. The condition means that Final must not become inconsistent by the addition. If a final check e does not depend on any node, we write it as $\mathsf{checkfinal}\ e$ and store it in Final as $(e\ \mathsf{depby}\ \bot)$ where $\bot$ signifies "no node".

$$(\mathrm{Pre}, \mathrm{State}, \mathrm{Final})\ \mathsf{checkfinal}\ p = b\ \mathsf{depby}\ q \equiv$$
$$(\mathrm{Pre}, \mathrm{State}, \mathrm{Final} \cup \{(p = b\ \mathsf{depby}\ q)\})$$
$$\text{if } (p = x\ \mathsf{depby}\ r) \notin \mathrm{Final} \text{ for all } r \text{ and } x \neq b$$

Immediate check. Then there is one last variety of check, the non-residuating "now" check. It allows one to find out whether a property has a value yet. In order to do that, we introduce the symbol $\bot$ to denote "undefined" and can check for undefinedness with the usual equality operator.

$$(\mathrm{Pre}, \mathrm{State}, \mathrm{Final})\ \mathsf{checknow}\ p = v \equiv$$
$$(\mathrm{Pre}, \mathrm{State}, \mathrm{Final})\quad \text{if } \mathrm{State}(p) = v$$

ϵ check. There is one special property, the so-called ϵ property that has the empty name. Any check on this property ("ϵ checks") always and immediately succeeds. See the definition of add below for an example of where an ϵ check can occur.

Design Rules: Destructive Commands

Change state. This operation changes the state in an imperative manner. No inconsistencies can arise, because we overwrite existing bindings.

$$(\mathrm{Pre}, \mathrm{State}, \mathrm{Final})\ \mathsf{set}\ p = b \equiv$$
$$(\mathrm{Pre}, \{(q = x) \in \mathrm{State} \mid q \neq p\} \cup \{(p = b)\}, \mathrm{Final})$$

Cleaning up Final. As final constraints sometimes depend on a node p (are "attached" to it), we have to provide an operation that removes these constraints when p is deleted.

$$(\mathrm{Pre}, \mathrm{State}, \mathrm{Final})\ \mathsf{cleanfinal}\ p \equiv$$
$$(\mathrm{Pre}, \mathrm{State}, \{(r = b\ \mathsf{depby}\ s) \in \mathrm{Final} \mid s \neq p\})$$

Renaming. The following operation provides the means to rename a binding. To maintain consistency, we also have to change references to it in the final constraints.

$$(\mathrm{Pre}, \mathrm{State}, \mathrm{Final})\ \mathsf{rename}\ p \to q \equiv (\mathrm{Pre},$$
$$\{(m(r) = b) \mid (r = b) \in \mathrm{State}\},$$
$$\{(m(r) = b\ \mathsf{depby}\ m(s)) \mid (r = b\ \mathsf{depby}\ s) \in \mathrm{Final}\})$$
$$\text{where } m(r) = \begin{cases} q & \text{if } r = p \\ r & \text{otherwise} \end{cases}$$

Tree Manipulation

Adding a node. Adding a node named X whose parent has the path r means that we check that the parent exists and that the node itself does not exist. Then we log that $r.X$ now exists and that a check for the existence of node[5] d depends on it. If the node to be added has no father, r is empty and ascertaining its existence is an ϵ check that always succeeds (see above).

$$\mathsf{add}\ r.X \dashv \{d\} \equiv\quad \mathsf{check}\ r = \mathsf{true}; \mathsf{check}\ r.X = \mathsf{false};$$
$$\mathsf{set}\ r.X = \mathsf{true};$$
$$\mathsf{checkfinal}\ d = \mathsf{true}\ \mathsf{depby}\ r.X$$

Refinement. Refinement is similar to adding, but has less side effects.

$$\mathsf{refine}\ p \dashv \{d\} \equiv\quad \mathsf{check}\ p = \mathsf{true};$$
$$\mathsf{checkfinal}\ d = \mathsf{true}\ \mathsf{depby}\ p$$

Deletion. delete performs the usual checks and changes. We also need to delete dependencies that have become obsolete by this operation.

$$\mathsf{delete}\ p \equiv\quad \mathsf{check}\ p = \mathsf{true}; \mathsf{set}\ p = \mathsf{false};$$
$$\mathsf{cleanfinal}\ p$$

Moving. This operation is much like first deleting and then adding, but renaming the bindings obviates the need for setting values in State and Final. move has an application that is motivated by Bracha's Jigsaw [8]: Refining a class is very similar to subclassing and move provides an alternative to `super()` calls from Java that is conceptually much cleaner: If one ever wants access to a method one overrides, one just moves the old method out of the way and calls it

---

5   By abuse of notation, we write a singleton node $\{d\}$ where actually a set $\{d_1, \cdots, d_n\}$ of nodes can be used.

from the new one. move makes sure that existing references (dependencies) to the old method are correctly updated.

$$
\begin{aligned}
\text{move } p \; r.X \equiv \quad & \text{check } p = \text{true}; \\
& \text{check } r = \text{true}; \text{check } r.X = \text{false}; \\
& \text{rename } p \to r.X
\end{aligned}
$$

Child Defaults

So far, we've made sure that add always produces well-formed trees. But we cannot make the same guarantee for delete, because we don't have yet the means to express the condition "if we delete a node, it must not have descendants". Child defaults are properties whose last path component is a wildcard. They allow us to make universally quantified assertions and checks about nodes. For example, before we can delete a node p, we have to make sure that all children are missing with check $p.* = \text{false}$ and potentially remember that as a constraint in Pre. Note that more specific assertions override the default (example: we first add a node p which guarantees that there are no children and then add one child $p.X$. Now the child default is still $p.* = \text{false}$, but in addition, we have the assertion $p.X = \text{true}$). We see that inside a GRAFT program, child defaults are used by add and delete in a complementary fashion: add asserts that the new node has no children, delete requires its argument to be a leaf.

Here is how we can extend the constraint store to support child defaults:

- Pre has a new kind of entry $(p.* = b)$.
- State also contains child defaults $(p.* = b)$. Additionally, we redefine the use of State as a function:

$$
\text{State}(p.X) = \begin{cases} b & \text{if } (p.X = b) \in \text{State} \\ c & \text{if } (p.X = b) \notin \text{State} \\ & \quad \wedge \, (p.* = c) \in \text{State} \\ \bot & \text{otherwise} \end{cases}
$$

- Final is unchanged, we only support child defaults for pre-checks.

Operations are extended as follows:

- New residuating check:

$$
\begin{aligned}
& (\text{Pre}, \text{State}, \text{Final}) \; \text{check } p.* = b \equiv \\
& \left\{ \begin{array}{l} (\text{Pre}, \text{State}, \text{Final}) \\ \quad \text{if for all } X.\text{State}(p.X) = b \\ (\text{Pre} \cup \{(p.* = b)\}, \text{State}, \text{Final}) \\ \quad \text{if } \forall X.\text{State}(p.X) \in \{b, \bot\} \\ \quad \wedge \, \exists Y.\text{State}(p.Y) = \bot \end{array} \right.
\end{aligned}
$$

- Change state: already works correctly for set $p = b$ in all instances: if the set command really provides new information, it either overrides an existing binding or complements a default with more specific information.

Otherwise, it leaves State untouched. A new variant can set defaults: set $p.* = b$ clears all settings for children of p.

- Rename: Rename must now change the names of the defaults, too. We omit the details.

5.2. Example Analysis

As an example, we infer the interface of feature `html` from Fig. 4(b). We sometimes abbreviate the class name `PlainText` as `PText` and the class name `BoldText` as `BText`. We do not show child defaults to make things easier to understand.

```
add Text.toHTML;
    Pre = {Text = true, Text.toHTML = false}
    State = {Text.toHTML = true}
    Final = ∅
add PlainText.toHTML ⊣ {PlainText._str}
    Pre = {PText = true, PText.toHTML = false,
        Text = true, Text.toHTML = false}
    State = {PText.toHTML = true,
        Text.toHTML = true}
    Final = {PText._str = true depby PText.toHTML}
add BoldText.toHTML ⊣ {BoldText._text}
    Pre = {BText = true, BText.toHTML = false,
        PText = true, PText.toHTML = false,
        Text = true, Text.toHTML = false}
    State = {BText.toHTML = true,
        PText.toHTML = true, Text.toHTML = true}
    Final = {BText._text = true depby BText.toHTML,
        PText._str = true depby PText.toHTML}
```

The last step is to reduce the set Final, but there are no reductions possible here; all of the dependencies are open. The last step therefore gave us an interface to feature `html`: Pre contains demands on predecessors in the composition chain (class `BoldText` should be available and not have a method `toHTML` etc.); State lists what feature `html` exports; Final contains nodes that still need to be added, either by a predecessor or by a successor.

An example for composition validation is to compose feature `html` with itself: `html; html`. This amounts to apply the above program a second time, this time to the last constraint store. Evaluation would correctly fail, for structural reasons: the first command, add Text.toHTML, requires that Text.toHTML = false which is in conflict with the value of this property in State.

6. Soundness

In this section, we briefly sketch what theoretical properties of GRAFT programs and checks can be proven and how. We first introduce a data structure for trees. It holds the tree

data plus the state and the final constraints that were created during its construction. Then we define how GRAFT operations change this data structure (omitted here). Finally, we need to define type relations in order to make meaningful assertions in our theorems:

- The type of a GRAFT program f is the constraint store that we have inferred for it: $f : (\mathrm{Pre}, \mathrm{State}, \mathrm{Final})$

- Given a tree t and a set Pre of pre-constraints, t has type Pre, written $t : \mathrm{Pre}$, if every constraint in Pre is fulfilled by the state data in t.

- If we view a State as a special case of pre-constraints, we can define a relation $t : \mathrm{State}$ just like in the previous item.

The following theorem asserts that if a tree t fulfills the pre-constraints of a program f, applying f to t *will* terminate and produce a tree whose type is the last state of f.

Theorem 1 (Subject Reduction) *Given a tree t and a* GRAFT *program f : $(\mathrm{Pre}, \mathrm{State}, \mathrm{Final})$. If t : Pre then* $(t)f : \mathrm{State}$.

Intuitively, the next theorem says that for any GRAFT program f : $(\mathrm{Pre}, \mathrm{State}, \mathrm{Final})$, Final correctly describes what dependencies that were declared in f have not yet been fulfilled. The meta-function deps extracts all dependencies that are declared in a program. Set difference $C \setminus \mathrm{State}$ between a set C of constraints and a state is defined as removing all constraints from C that are fulfilled by State. The subset relation $C \subseteq D$ between sets of constraints has the obvious definition and ignores the source of a constraint.

Theorem 2 (Dependencies) *If f : $(\mathrm{Pre}, \mathrm{State}, \mathrm{Final})$ then* $\mathrm{deps}(f) \setminus \mathrm{State} \subseteq \mathrm{Final}$.

7. Related Work

Out of the substantial body of work on software verification, composition validation, consistency checking etc., we can only pick a few representative examples; mostly work that has inspired GRAFT. For an extended version of this section, refer to [20].

A Framework for the Detection and Resolution of Aspect Interactions (DRAI). In [12], *aspect-oriented* programs [13] are translated to a formal notation and analyses are performed on the result. This is where the similarities with GRAFT end: DRAI models the dynamic execution of a program; analysis is only concerned with making sure that only one of the simultaneously active aspects is applicable at a time, which leads to deterministic execution. Conversely, GRAFT formalizes a static approach to composition that focuses on scalability and support for many artifact kinds (some of whom cannot be executed). Furthermore, GRAFT's properties and checks permit enforcement of semantic constraints and dependencies that often cannot be inferred from the syntax and complement conflict resolution.

A Calculus of Module Systems (CMS, [2]). Among module calculi, CMS is one of the most recent and general. CMS feels similar to GRAFT in many ways, but there are also marked differences: The focus of CMS is theoretic, it is able to encode the Abadi-Cardelli object calculus [1] and lambda calculus; GRAFT grew out of the desire to perform checking for AHEAD. The CMS way of composition is algebraic, GRAFT's ideas lean towards tree transformation and better suit the needs of generative programming. Finally, CMS is targeted at programming languages, while GRAFT needs to support many artifact kinds. CMS does not have an equivalent to design rules, either.

Evolution Contracts. Evolution contracts [17] are a very powerful formalism for supporting software evolution and detecting conflicts while doing so. The comprehensiveness of this approach is impressive [16], but its complexity can be daunting and evolution and analysis steps must often be defined and applied by hand. Contrarily, GRAFT keeps things simple, custom semantic constraints are easily implemented using design rules and the analysis is performed automatically. The focus of evolution contracts on software evolution is also slightly different from GRAFT's focus on generative programming.

Consistency Checking. The approach to consistency checking taken by Nentwich et al. [18] enforces constraints on a software system in a way that is orthogonal to our way of composition validation. It is concerned with the consistency of a system at a certain point in time, whereas we want to know if modifications are valid, before performing them. xlinkit [18] specifies consistency constraints as conditions on XML data. This leads to multi-artifact support that would be very useful in the setting of generative programming.

8. Conclusions and Future Research

We have presented GRAFT, a calculus that provides a theoretical foundation for the generator framework AHEAD. Our main contributions are: A clear and concise definition of the semantics of AHEAD; automated consistency checking; and interface inference for features. GRAFT analyses are independent of the language[6] that the artifact is expressed in; a fact that is crucial for a universal generator framework. Note that while most of GRAFT is quite AHEAD-specific, design rules are a universal approach of doing composition validation that

---

6   This includes formal languages not used for programming.

could just as well be applied to aspect-oriented programming.

Due to its non-monotonicity, formalizing the delete operator was an added challenge. While deletion is not very common in the usual composition paradigms, it is important to have that operator for two reasons: First, if refinement does not monotonically add functionality, but is instead a total (or partial) replacement of a node, it is not really refinement any more, but rather a delete and an add. Second, in order to gauge the impact of taking a node out of the system, you can delete it and then perform an analysis to find out if constraints have been invalidated.

Future research will enrich the GRAFT language with features such as lambda abstraction, the ability to reflect on the contents of a node, loops, conditionals etc. Dependencies are also just a small first step in describing various relationships between and properties of artifacts. In this field there are many ideas from other works that are applicable, among them: Behavioral interface specification [10], statechart diagrams [19], specialisation interfaces [14] and refinement calculus [9]. Typing is another issue: Is the current mechanism sufficient or would we want more than just constraints for typing a GRAFT program? Tree types similar to XML Schema would also probably be a useful addition. Easy and logical minor enhancements include support for property types other than Boolean and more checks than just equality (both things are already present in design rule checking). Lastly, we have implemented a previous version of GRAFT in Java and plan to update it to the status quo.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.

[2] D. Ancona and E. Zucca. A Calculus of Module Systems. *Journal of Functional Programming*, 12(2):91–132, March 2002.

[3] D. Batory et al. Hompage of the AHEAD Tool Suite. http://www.cs.utexas.edu/users/schwartz/ATS.html.

[4] D. Batory and B. J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Trans. Software Engineering*, 23(2):67–82, 1997.

[5] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. In *Proc. 25th IEEE Int. Conf. Software Engineering (ICSE)*. IEEE, 2003.

[6] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *ACM Transactions on Software Engineering (TSE)*, 30(6):355–371, June 2004.

[7] J. Bloch et al. A Metadata Facility for the Java Programming Language, 2002. Java Specification Request 175.

[8] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Department of Comp. Sci., Univ. of Utah, 1992.

[9] M. Büchi and E. Sekerinski. Formal Methods for Component Software: The Refinement Calculus Perspective. In *Proc. 2nd Wsh. Component-Oriented Programming (WCOP) held in conjunction with ECOOP, Jyväskylä*, June 1997.

[10] L. Burdy et al. An Overview of JML Tools and Applications. In T. Arts and W. Fokkink, editors, *8th Int. Wsh. Formal Methods for Industrial Critical Systems (FMICS)*, volume 80 of *Electronic Notes in Theoretical Computer Science*, pages 73–89. Elsevier, June 2003.

[11] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, 2000.

[12] R. Douence, P. Fradet, and M. Südholt. A Framework for the Detection and Resolution of Aspect Interactions. In D. S. Batory, C. Consel, and W. Taha, editors, *Proc. 1st Conf. Generative Programming and Component Engineering (GPCE)*, volume 2487 of *Lect. Notes Comp. Sci.*, pages 173–188, 2002.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. L. Knudsen, editor, *Proc. 15th Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 2072 of *Lect. Notes Comp. Sci.*, pages 327–353. Springer, 2001.

[14] J. Lamping. Typing the Specialisation Interface. In *Proc. 8th Conf. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 28 of *ACM SIGPLAN Notices*, pages 201–214. ACM Press, 1993.

[15] R. E. Lopez-Herrejon and D. Batory. Evaluating Feature Modularity: A Case Study, 2004. Submitted for publication.

[16] T. Mens. A Formal Foundation for Object-Oriented Software Evolution. In *Proc. Int. Conf. Software Maintenance (ICSM)*, pages 549–552. IEEE, 2001.

[17] T. Mens and T. D'Hondt. Automating Support for Software Evolution in UML. *Automated Software Engineering*, 7(1):39–59, 2000.

[18] C. Nentwich, W. Emmerich, and A. Finkelstein. Static Consistency Checking for Distributed Specifications. In *Proc. 16th Conf. Automated Software Engineering (ASE)*, pages 115–124. IEEE Computer Society, 2001.

[19] C. Prehofer. Plug-and-Play Composition of Features and Feature Interactions with Statechart Diagrams. In *Proc. 7th Int. Wsh. Feature Interactions in Telecommunications and Software Systems*, 2003.

[20] A. Rauschmayer, A. Knapp, and M. Wirsing. Type-Checking AHEAD. Technical Report 0406, Ludwig-Maximilians-Universität München, Institut für Informatik, 2003.

[21] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[22] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering*. Addison Wesley, 1999.