Structured Document Algebra in Action

Don Batory[1], Peter Höfner[2], Dominik Köppl[3],
Bernhard Möller[4], and Andreas Zelend[4]

[1] Dept. of Computer Science, University of Texas at Austin, USA
[2] NICTA and UNSW, Australia
[3] Department of Computer Science, TU Dortmund, Germany
[4] Institut für Informatik, Universität Augsburg, Germany

Abstract. A *Structured Document Algebra (SDA)* defines modules with variation points and how such modules compose. The basic operations are module addition and replacement. Repeated addition can create nested module structures. SDA also allows the decomposition of modules into smaller parts. In this paper we show how SDA modules can be used to deal algebraically with *Software Product Lines (SPLs)*. In particular, we treat some fundamental concepts of SPLs, such as refinement and refactoring. This leads to mathematically precise formalization of fundamental concepts used in SPLs, which can be used for improved *Feature-Oriented Software Development* (FOSD) tooling.

Keywords: software product lines, feature-oriented design, algebraic reasoning

It is our pleasure to dedicate this paper to Martin Wirsing on the occasion of his Formal Retirement. We contribute a study on a recently developed algebra for all kinds of structured and interconnected documents, but particularly the ones that describe product families or product lines in Feature Oriented Software Design — a topic on which Martin has been quite active for a while now. The frame of this are general formal methods and semantics. We pick up this latter theme to endow the algebra, which previously had a more syntactic flavour, with a semantic component, too. The particular approach we take uses the terminology of algebraic specification; this gives the fourth author the opportunity to fondly remember the days of the CIP project at TU Munich, in which Martin and he cooperated quite a lot on that topic. We hope that Martin will enjoy that bit of scientific nostalgia, too! Best wishes, Martin, for your Formal Retirement — enjoy! — but also for many further successful years, since we do hope that your Retirement is only Formal!

1 Introduction

A *Software Product Line (SPL)* is a family of related programs constructed from a common set of assets. Variations in programs are explained by *features*—increments in program functionality. The assets of an SPL are modules that implement features. These modules are the building blocks of SPL programs [2].

Today's SPL researchers are exploring two rather different forms of feature-based modularity: alternative-based variation (a.k.a. *classical modularity*) and projectional variation (a.k.a. *SYSGEN* or *virtual modularity*). Classical modularity is what you would expect: there are physical files that define a feature module and tools that compose modules to produce a desired program. Here, files refer to arbitrary documents, such as specifications, code composed of elementary program features, text fragments or manuals. In contrast,

virtual feature modularity is a preprocessor technology called *coloring*. The idea is simple: the code of the `Blue` feature is painted blue; code of the `Green` feature is painted green. Whenever `Blue` is not needed in a product, all blue-colored code is removed or is said to be *projected out*. The tools for virtual modularity are historically based on text preprocessors; more advanced tools color *abstract syntax trees (ASTs)* [5, 15, 18]. The current debate is which implementation technique is most appropriate for an SPL. Our position is that both implement the same abstractions—feature modules—in very different ways. *What is important is to understand the algebraic nature of these abstractions.*

The *Structured Document Algebra (SDA)*, partially first presented in [6], aims at providing a simple, yet effective, algebra of feature modularity (e.g., the modularization of text files, text fragments, ASTs, etc.).

It is completely independent of the underlying programming paradigm, such as Object-Oriented Programming or Functional Programming. In fact, it is even independent of programming, since it is abstract enough to cover also general structured documents, such as manuals or collections of web pages. It is meant as an aid for formal reasoning about the process of decomposing larger document pieces into smaller ones (and vice versa). It can be implemented on top of or inside any text editor, IDE or web page editor. Depending on the desired level of reasoning, it can be used purely syntactically or at a semantic level, as demonstrated in the Appendix. Phenomena modelled by the algebra include coherence, uniform transformations, deletion, overriding. In the special case of SPL documents, SDA additionally allows a description of their fine structure. This is in contrast to other algebraic approaches in this area (e.g., [1, 24, 25]), where modules are often treated as atomic units.

In the present paper we extend the basic repertoire of the version of SDA from [6] and show how it can be used to formally describe some standard techniques used in feature oriented software construction and SPLs.

The basis of SDA are structured, inter-linked documents or document *fragments*. A link is represented as a *Variation Point (VP)*, i.e., a labeled position in a fragment where contents can be inserted to yield a larger fragment. The association between VPs and their assigned fragments, if any, is provided by *modules*, i.e., partial functions from VPs to fragments.

SDA allows multiple "applied occurrences", i.e., replication of one and the same VP `v`; they all stand for (or share) the fragment assigned to `v` by some module. Conversely, different VPs may be associated with the same fragment (a module need not be an injective partial function).

In this paper we show how modules and fragments can be treated algebraically and discuss algebraic operations for module (de)composition and present an operation for overriding.

2 Structured Document Algebra

This section recapitulates a formal model of VPs, modules containing VPs, and compositions of such modules first presented in [6]. To keep SDA language-independent, we leave the exact nature of fragments (e.g., text or abstract syntax trees) unspecified and view it as a parameter of the algebra. For our examples we will use Java code fragments.

2.1 SDA Basics

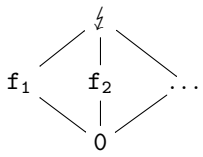
Variation Points and Fragments. We now formalize the notions mentioned above. Let $\mathtt{V}$ be a set of VPs, denoted by $\mathtt{v}_1, \mathtt{v}_2, \ldots$ at which fragments may be inserted and $\mathtt{F}(\mathtt{V})$ be a set of *fragments*, denoted by $\mathtt{f}_1, \mathtt{f}_2, \ldots$. Fragments may contain VPs from $\mathtt{V}$.

In incremental software design it is often advantageous to leave certain parts unspecified and to insert placeholders where (optionally) further features may be added. As an abstraction of such placeholders we use *default fragments*. In addition, it is convenient to introduce a special pseudo-fragment that represents an error, namely that there has been an attempt to assign two or more non-default and different fragments to the same VP. To this end we assume that the set $\mathtt{F}(\mathtt{V})$ includes two special elements, a default fragment $\mathtt{0}$ and an error $\lightning$.

The addition, or supremum, operator $+$ on fragments has the axioms

$$\mathtt{0} + \mathtt{x} = \mathtt{x}\ , \quad \lightning + \mathtt{x} = \lightning\ , \quad \mathtt{f_i} + \mathtt{f_j} = \lightning\ (\mathtt{i} \neq \mathtt{j})\ ,$$

where $\mathtt{x}, \mathtt{f_i}, \mathtt{f_j} \in \mathtt{F}(\mathtt{V})$ with $\mathtt{f_i}, \mathtt{f_j} \neq \mathtt{0}$. If we assume associativity, idempotence and commutativity of addition, this structure forms a flat lattice with least element $\mathtt{0}$ and greatest element $\lightning$.



Modules. A *module* is a partial function $\mathtt{m} : \mathtt{V} \rightsquigarrow \mathtt{F}(\mathtt{V})$. A VP $\mathtt{v}$ is *assigned* by $\mathtt{m}$ if $\mathtt{v} \in \mathtt{dom}(\mathtt{m})$, otherwise *unassigned* or *external*. Thus the domain $\mathtt{dom}(\mathtt{m})$ of a module is the set of VPs it "knows about" or administers. By using partial functions rather than relations, a VP can be filled with at most one fragment (*uniqueness*).

Example 2.1 Figure 1(a) is a sample file/module, structured by the assignment of fragments to its VPs. Its partial function is given in Figure 1(b). □

Ducasse et al. [12] also use a flat lattice of composable units, called *traits*. Our modules correspond to the *method dictionaries* there. However, these have to be total rather than partial maps, which makes distinguishing assigned and external VPs difficult. Had we taken the same decision, our algebraic laws would have become much more cumbersome.

A module $\mathtt{m}$ can be viewed in different ways:

- as a collection of fragments that instantiate the VPs of $\mathtt{dom}(\mathtt{m})$, i.e., a structured document;
- as filling certain VPs with contents (in term rewriting etc., it would be called a *substitution*); and
- as a generalized context-free grammar with $\mathtt{dom}(\mathtt{m})$ as the set of nonterminals and a production $\mathtt{v} \mapsto \mathtt{m}(\mathtt{v})$ for each $\mathtt{v} \in \mathtt{dom}(\mathtt{m})$.

The simplest module is the *empty module* $\mathbf{0}$, i.e., the empty partial map. Another very simple kind of module is provided by *constant modules*, i.e., modules which assign one and the same fragment—for instance $\mathtt{0}$—to a number of VPs. Let $\mathtt{W} \subseteq \mathtt{V}$ be a set of VPs and $\mathtt{f}$ a fragment. We set

$$[\mathtt{W} \mapsto \mathtt{f}](\mathtt{v}) =_{df} \begin{cases} \mathtt{f} & \text{if } \mathtt{v} \in \mathtt{W}, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

If $\mathtt{W} = \{\mathtt{w}\}$ is a singleton set, we abbreviate $[\{\mathtt{w}\} \mapsto \mathtt{f}]$ to $[\mathtt{w} \mapsto \mathtt{f}]$. Such *singleton modules* are the atomic building blocks from which all other modules can be constructed by the addition operation introduced next.

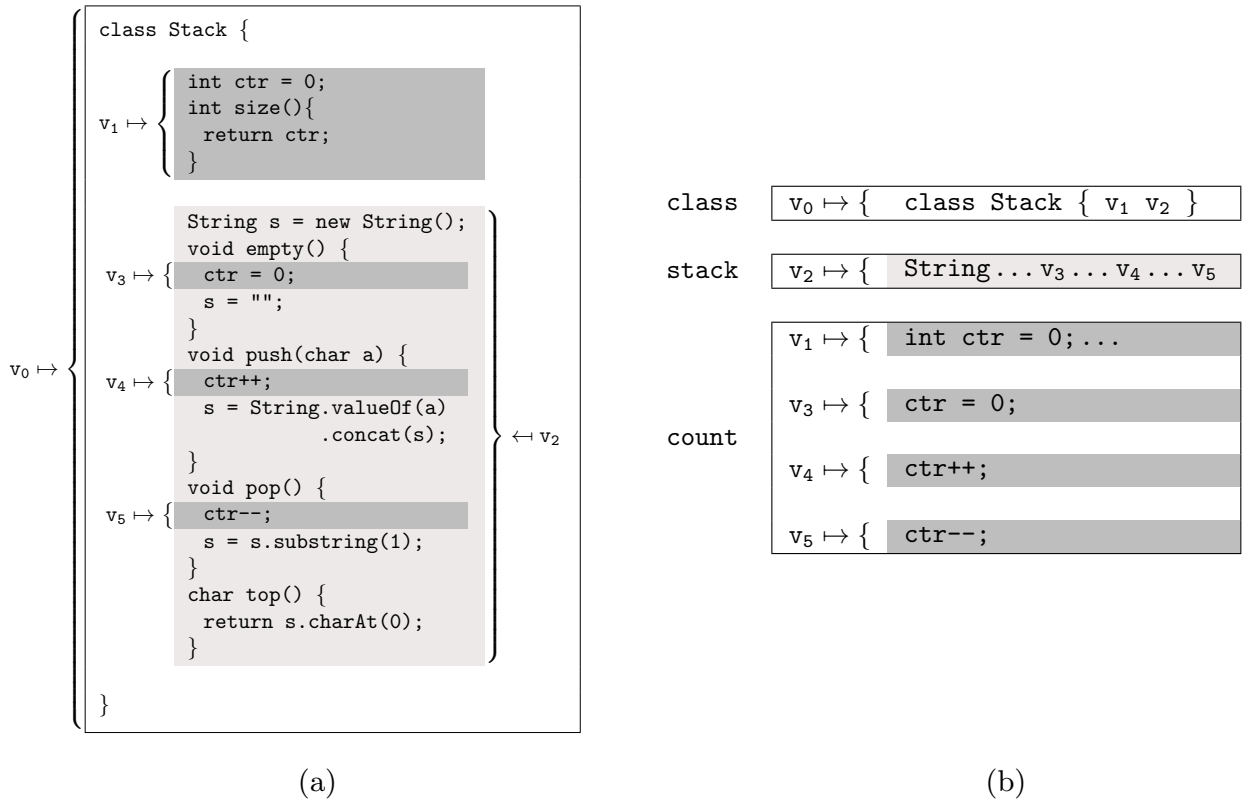


Fig. 1. Variation Points, Fragments and Modules.

Module Addition. We want to construct larger modules step by step by assigning more and more fragments to VPs. The central operation for this is module addition (+), which fuses two modules while maintaining uniqueness (and signaling an error upon a conflict). Basically, module addition can be viewed as union with flagged inconsistent VP/fragment associations. Desirable properties for + are commutativity and associativity. If two modules have no VPs in common, the partial functions characterizing them can easily be combined. For example, `class` + `stack` (Figure 1(b)) is the partial function

$$\texttt{class} + \texttt{stack} = \begin{cases} \texttt{v}_0 \mapsto \texttt{class Stack } \{\texttt{v}_1\ \texttt{v}_2\} \\ \texttt{v}_2 \mapsto \texttt{String ... v}_3 \texttt{ ... v}_4 \texttt{ ... v}_5 \end{cases}$$

Module addition can be defined as the lifting of + on fragments

$$(\mathtt{m}+\mathtt{n})(\mathtt{v}) =_{df} \begin{cases} \mathtt{m}(\mathtt{v}) & \text{if } \mathtt{v} \in \mathtt{dom}(\mathtt{m}) - \mathtt{dom}(\mathtt{n}) \\ \mathtt{n}(\mathtt{v}) & \text{if } \mathtt{v} \in \mathtt{dom}(\mathtt{n}) - \mathtt{dom}(\mathtt{m}) \\ \mathtt{m}(\mathtt{v}) + \mathtt{n}(\mathtt{v}) & \text{if } \mathtt{v} \in \mathtt{dom}(\mathtt{m}) \cap \mathtt{dom}(\mathtt{n}) \\ \text{undefined} & \text{if } \mathtt{v} \notin \mathtt{dom}(\mathtt{m}) \cup \mathtt{dom}(\mathtt{n}) \end{cases}$$

If in the third case $\mathtt{m}(\mathtt{v}) \neq \mathtt{n}(\mathtt{v})$ and $\mathtt{m}(\mathtt{v}), \mathtt{n}(\mathtt{v}) \neq \mathtt{0}$ then $(\mathtt{m}+\mathtt{n})(\mathtt{v}) = \lightning$, thus signaling an error. By the above laws for fragment addition, the set of modules forms a commutative and idempotent monoid under

$+$.[5] Therefore we can define a *submodule relation* as the natural order

$$\mathtt{m} \leq \mathtt{n} \Leftrightarrow_{df} \mathtt{m} + \mathtt{n} = \mathtt{n} \ . \tag{1}$$

Example 2.2 Figure 1(b) shows three modules: `class` and `stack` contain single fragments, assigned to $\mathtt{v}_0$ and $\mathtt{v}_2$ respectively; `count` contains fragments assigned to $\mathtt{v}_1$, $\mathtt{v}_3$, $\mathtt{v}_4$, and $\mathtt{v}_5$. The module addition `class` + `stack` + `count` corresponds to the code in Figure 1(a). □

Implementation. A simple example suggests several ways in which SDA modules can be implemented. Figure 2(a) shows how preprocessor macros might define three non-default fragments (labeled `BLUE`, `GREEN`, `RED`) and a default for an implicit variation point. Figure 2(b) shows how this might be rendered in a "coloring" tool (e.g., the one contained in the integrated development environment CIDE [18, 19, 2]) where the fragments assigned to the VPs are explicitly shown. (There is no need to actually "see" the names of VPs). However, Figure 2(b) would require significant engineering: a Java compiler would have to understand the preprocessor semantics of coloring (Figure 2(a)) so as not to alert programmers that the `GREEN` fragments and beyond are unreachable if `BLUE` is true. In a Java-like language this can be accomplished by defining feature variables as static Booleans and put a wrapper `if (feature){ fragment }` around each fragment. So if the feature variable is false, the compiler will effectively ignore the fragment as dead code. This is, for instance, offered by the CIDE compiler.

A more likely possibility—which is consistent with current text coloring tools—would be to "fool" the compiler by pretending that the code of Figure 2(c) is the definition of the `add` method, where a projection would produce a simpler method with only one assignment to the variable `result`.

```
int add(int x) {
#if BLUE
    return x+3;
#elif GREEN
    return x+5;
#elif RED
    return x+11;
#else
    return x+1
#endif
}
```

(a)

```
int add(int x) {

v1 ↦ { return x+3;
v1 ↦ { return x+5;
v1 ↦ { return x+11;
v1 ↦ { return x+1;

}
```

(b)

```
int add(int x) {

v1 ↦ { result = x+3;
v1 ↦ { result = x+5;
v1 ↦ { result = x+11;
v1 ↦ { result = x+1;
       return result;
}
```

(c)

Fig. 2. Module Implementations

[5] Modules and module addition can be recoded in terms of total functions, which makes it easier to see that the $+$ operation indeed is commutative, associative and idempotent, hence induces a lattice, too. Moreover, it has the empty module `0` as its neutral element and satisfies $\mathtt{dom}(\mathtt{m}+\mathtt{n}) = \mathtt{dom}(\mathtt{m}) \cup \mathtt{dom}(\mathtt{n})$.

These ideas are, in effect, standard fare for SPL development, except that the tool support needs to be beautified by coloring and VP recognition. Code fragments or *mini-modules* can indeed be expressed in terms of classical module systems; see [10] for examples. Coloring also presents the connection between modules and virtual modularity (cf. Sect. 1).

2.2 Structural Properties of Modules

Since fragments may contain VPs, cycles could occur when composing modules. A module, however, should be cycle-free. To handle this we use a dependence relation.

Cycle-Freeness. For a fragment $\mathtt{f} \in \mathtt{F(V)}$ let $\mathtt{VP(f)}$ be the set of VPs that occur in $\mathtt{f}$. We define a *direct dependence relation* $\mathtt{dep_m} \subseteq \mathtt{V} \times \mathtt{V}$ within a module $\mathtt{m}$ by

$$\mathtt{v}\ \mathtt{dep_m}\ \mathtt{w} \quad \Leftrightarrow_{df} \quad \mathtt{v} \in \mathtt{dom(m)} \ \wedge\ \mathtt{w} \in \mathtt{VP(m(v))}\ .$$

This means that VP $\mathtt{w}$ occurs in the fragment assigned to VP $\mathtt{v}$ by $\mathtt{m}$, so that $\mathtt{v}$ depends on $\mathtt{w}$. For example, in Figure 1(b), $\mathtt{v_2}\ \mathtt{dep_{stack}}\ \mathtt{v_3}$. A module $\mathtt{m}$ is *acyclic* if no VP depends directly or indirectly on itself, i.e., no VP $\mathtt{v}$ satisfies $\mathtt{v}\ \mathtt{dep_m^+}\ \mathtt{v}$, where $\mathtt{dep_m^+}$ is the transitive closure of $\mathtt{dep_m}$. Henceforth we only consider cycle-free modules.

The concepts of dependence and module addition do not interfere. Since the latter is just the "union" of VP/fragment associations, for acyclicity it does not matter whether dependences occur inside one module or between different modules.

Assembling Fragments. We now describe how to assemble a structured document into a single fragment (while "forgetting" the structure). To define this formally we use an auxiliary function $\mathtt{single\_fill(f,m)}$. It takes a fragment $\mathtt{f}$ and a module $\mathtt{m}$ and yields the fragment that results from $\mathtt{f}$ by replacing, in parallel, all occurrences of every $\mathtt{v} \in \mathtt{VP(f)}$ by the corresponding fragment $\mathtt{m(v)}$ (if any). The precise definition of $\mathtt{single\_fill}$ depends on the special type of fragments considered; as stated in the introduction we want to keep that parametric. For an acyclic module $\mathtt{m}$ and $\mathtt{v} \in \mathtt{dom(m)}$, the fragment $\mathtt{frag(v,m)}$ can be computed by iterating the $\mathtt{single\_fill}$ function. By the assumed acyclicity of $\mathtt{m}$ this always terminates. To cope with unassigned (= external) VPs we assume that every VP is also a fragment, i.e., that $\mathtt{V} \subseteq \mathtt{F(V)}$. Then the external VPs can simply be left unchanged by the assembly function. A corresponding program looks as follows:

```
fragment frag (vp v, module m){
  fragment f = v;
  while (VP(f) ∩ dom(m) ≠ ∅)
    f = single_fill(f,m);
  return f;}
```

Note that $\mathtt{frag(v,m)} = \mathtt{v}$ if $\mathtt{v} \notin \mathtt{dom(m)}$.

Module Equivalence. Two modules $\mathtt{m}, \mathtt{n}$ are *equivalent* if they generate the same fragments for all VPs:

$$\mathtt{m} \equiv \mathtt{n} \quad \Leftrightarrow_{df} \quad \forall \mathtt{v} \in \mathtt{VP} : \mathtt{frag(v,m)} = \mathtt{frag(v,n)}\ .$$

Lemma 2.3 *For modules* $\mathtt{m}, \mathtt{n}$ *we have* $\mathtt{m} \equiv \mathtt{n} \ \Rightarrow\ \mathtt{dom(m)} = \mathtt{dom(n)}$.

Restructuring. Module $\mathtt{m}$ *restructures into* module $\mathtt{n}$, in symbols $\mathtt{m} \sqsubseteq \mathtt{n}$, if

$$\mathtt{m} \sqsubseteq \mathtt{n} \Leftrightarrow_{df} \mathtt{dom(m)} \subseteq \mathtt{dom(n)} \wedge \forall\, \mathtt{v} \in \mathtt{dom(m)} : \mathtt{frag(v,m)} = \mathtt{frag(v,n)} \ .$$

This means that $\mathtt{n}$ offers a possibly more detailed representation of the fragments of $\mathtt{m}$ using auxiliary VPs in $\mathtt{dom(n)} - \mathtt{dom(m)}$. The empty module is the smallest one w.r.t. $\sqsubseteq$, since it does not offer any choice—in particular since $\mathtt{dom(0)} = \emptyset$.

It is easy to check that the relation $\sqsubseteq$ is reflexive and transitive, i.e., a preorder. It is, however, not antisymmetric as the following example shows.

Example 2.4 Consider the modules $\mathtt{m}$ and $\mathtt{n}$ defined as follows:

$$\begin{array}{lllll} \mathtt{m}: & \mathtt{v} \mapsto \mathtt{i{+}{+};\ w;} & \text{and} & \mathtt{w} \mapsto \mathtt{j{+}{+};} \\ \mathtt{n}: & \mathtt{v} \mapsto \mathtt{i{+}{+};\ j{+}{+};} & \text{and} & \mathtt{w} \mapsto \mathtt{j{+}{+};} \end{array}$$

These are equivalent, i.e., $\mathtt{m} \equiv \mathtt{n}$, in that they produce the same text. Hence, $\mathtt{m} \sqsubseteq \mathtt{n}$ and $\mathtt{n} \sqsubseteq \mathtt{m}$. If $\sqsubseteq$ were antisymmetric, this should imply $\mathtt{m} = \mathtt{n}$, but $\mathtt{m}$ and $\mathtt{n}$ are different as modules. □

However, we always have $\mathtt{m} \sqsubseteq \mathtt{n} \wedge \mathtt{n} \sqsubseteq \mathtt{m} \Leftrightarrow \mathtt{m} \equiv \mathtt{n}$.

We provide some basic properties of $\sqsubseteq$ that can be used to restructure larger modules in a modular fashion. To state them we define the set of all VPs occurring in a module $\mathtt{m}$ by $\mathtt{VP(m)} =_{df} \mathtt{dom(m)} \cup \bigcup_{\mathtt{v} \in \mathtt{dom(m)}} \mathtt{VP(m(v))}$. First we treat the case of decomposing the fragment belonging to a single VP.

Lemma 2.5 *Assume fragments* $\mathtt{f}, \mathtt{g}$ *and a module* $\mathtt{m}$ *with* $\mathtt{dom(m)} = \mathtt{VP(g)}$ *and* $\mathtt{f} = \mathtt{single\_fill(g,m)}$. *This means that the VPs in* $\mathtt{g}$ *are filled by* $\mathtt{m}$ *yielding* $\mathtt{f}$. *Moreover let* $\mathtt{v}$ *be a fresh VP, i.e.,* $\mathtt{v} \notin \mathtt{dom(m)} \cup \mathtt{VP(m)}$. *Then*

$$[\mathtt{v} \mapsto \mathtt{f}] \ \sqsubseteq \ ([\mathtt{v} \mapsto \mathtt{g}] + \mathtt{m}) \ .$$

The proof is immediate from the definitions and assumptions.

Lemma 2.6 *For modules* $\mathtt{m_i} \sqsubseteq \mathtt{n_i} (\mathtt{i} = 1, 2)$ *with* $\mathtt{VP}(\mathtt{n_1}) \cap \mathtt{VP}(\mathtt{n_2}) = \emptyset$ *we have*

$$\mathtt{m_1} + \mathtt{m_2} \ \sqsubseteq \ \mathtt{n_1} + \mathtt{n_2} \ .$$

Proof. (Sketch) We first state an auxiliary property. Assume modules $\mathtt{m}, \mathtt{n}$ such that $\mathtt{m} + \mathtt{n}$ is acyclic and $\mathtt{VP(m)} \cap \mathtt{VP(n)} = \emptyset$. Then $\forall\, \mathtt{v} \in \mathtt{dom(m)} : \mathtt{frag(v, m + n)} = \mathtt{frag(v, m)}$. In words, if $\mathtt{n}$ does not mention the VPs of $\mathtt{m}$ then it cannot influence the fragment represented by $\mathtt{m}$. This is shown by induction on the length of the longest $\mathtt{dep_m}$-path (or equivalently the smallest natural number $\mathtt{i}$ such that $\mathtt{dep_m}^{\mathtt{i}} = \emptyset$), which exists by the assumed acyclicity of $\mathtt{m} + \mathtt{n}$ and hence also of $\mathtt{m}$.

With that, the main claim follows easily from the definition of $\sqsubseteq$, because $\mathtt{VP}(\mathtt{n_1}) \cap \mathtt{VP}(\mathtt{n_2}) = \emptyset$ implies $\mathtt{VP}(\mathtt{m_i}) \cap \mathtt{VP}(\mathtt{n_j}) = \emptyset$ $(\mathtt{i} \neq \mathtt{j})$. □

3 Additional SDA Operators

3.1 Subtraction

In this section we present a way of defining an "inverse" to addition. The usefulness of this operation might not be straightforward. However, as we will show, it lays the foundation for more sophisticated operations such as overriding, which is a common technique in feature-oriented software development.

For modules $\mathtt{m}$ and $\mathtt{n}$ we define the *subtraction* $\mathtt{m} - \mathtt{n}$ via restriction $|$ as

$$\mathtt{m} - \mathtt{n} =_{df} \mathtt{m}\,|_{\mathtt{dom(m)}-\mathtt{dom(n)}} \,.^{6}$$

This spells out to

$$(\mathtt{m} - \mathtt{n})(\mathtt{v}) =_{df} \begin{cases} \mathtt{m}(\mathtt{v}) & \text{if } \mathtt{v} \in (\mathtt{dom(m)} - \mathtt{dom(n)}) \\ \text{undefined otherwise} \,. \end{cases}$$

Among others, subtraction satisfies, for arbitrary modules $\mathtt{m}, \mathtt{n}$ and $\mathtt{p}$, the following laws. (Proofs are straightforward.)

$$\begin{array}{rcl|l} \mathtt{dom(m-n)} & = & \mathtt{dom(m)} - \mathtt{dom(n)} & \mathtt{m-n} \leq \mathtt{m} \\ \mathtt{(m+n)-p} & = & \mathtt{(m-p)} + \mathtt{(n-p)} & \mathtt{m-0} = \mathtt{m} \\ \mathtt{m-(n+p)} & = & \mathtt{(m-n)-p} & \mathtt{m-m} = \mathbf{0} \\ \mathtt{m} \leq \mathtt{n} & \Rightarrow & \mathtt{m-n} = \mathbf{0} & \mathbf{0} - \mathtt{n} = \mathbf{0} \,. \end{array}$$

Note that the left law in the last line is only an implication, while the corresponding one for set theoretic difference is an equivalence. For the reverse direction we only have $\mathtt{m} - \mathtt{n} = \mathbf{0} \;\Rightarrow\; \mathtt{dom(m)} \subseteq \mathtt{dom(n)}$. Moreover, $\mathtt{m} - \mathtt{n} \sqsubseteq \mathtt{m}$ need not hold. Subtraction is isotone in its right argument and antitone in its left, i.e.,

$$\mathtt{m} \leq \mathtt{n} \;\Rightarrow\; \mathtt{m} - \mathtt{o} \leq \mathtt{n} - \mathtt{o} \,, \quad \text{and} \quad \mathtt{n} \leq \mathtt{o} \;\Rightarrow\; \mathtt{m} - \mathtt{o} \leq \mathtt{m} - \mathtt{n} \,.$$

3.2 Overriding

Ideally, modules that are composed have disjoint domains. And by using subtraction or deletion, modules can be customized. Still, object-oriented programmers are used to the notion of *overriding* or *replacing* definitions, an operation that can be defined in terms of subtraction and addition. The module $\mathtt{m} \mathbin{\rightarrowtriangle} \mathtt{n}$ which results from overriding $\mathtt{n}$ by $\mathtt{m}$ is defined as

$$\mathtt{m} \mathbin{\rightarrowtriangle} \mathtt{n} =_{df} \mathtt{m} + (\mathtt{n} - \mathtt{m}) \,.$$

This replaces all assignments in $\mathtt{n}$ for which $\mathtt{m}$ also provides a value. It may destroy acyclicity. $\mathbin{\rightarrowtriangle}$ is associative and idempotent with neutral element $\mathbf{0}$, but not commutative.

Example 3.1 A classic example of feature interaction and product lines is the fire-and-flood control [17]. Assume a building is equipped with two systems: a fire control and a flood control. The flood control turns off the water supply as soon as sensors in the building indicate a water level is too high. It is specified by

[6] Instead, one could define another subtraction operator $\ominus$ of type $(\mathtt{V} \rightsquigarrow \mathtt{F(V)}) \times \mathcal{P}(\mathtt{V}) \;\to\; (\mathtt{V} \rightsquigarrow \mathtt{F(V)})$ by $\mathtt{m} \ominus \mathtt{U} =_{df} \mathtt{m}\,|_{\mathtt{dom(m)}-\mathtt{U}}$ and then set $\mathtt{m} - \mathtt{n} =_{df} \mathtt{m} \ominus \mathtt{dom(n)}$.

the following module.

$$\texttt{Flood}: \begin{cases} \texttt{pv} \mapsto \texttt{void flood()\{ pt \}} \\ \texttt{pt} \mapsto \texttt{if(isWaterHigh()) waterOff(); else waterOn();} \end{cases}$$

Fire control works in an opposite manner: as soon as sensors detect a too high temperature, sprinklers are turned on.

$$\texttt{Fire}: \begin{cases} \texttt{pw} \mapsto \texttt{void fire()\{ i1 \}} \\ \texttt{i1} \mapsto \texttt{if(isTemperatureHigh()) spriOn(); else spriOff();} \end{cases}$$

Note that the sprinklers' functionality depends on a working water supply. Their composition/sum is $\texttt{FplusF} = \texttt{Fire} + \texttt{Flood}$:

$$\texttt{FplusF}: \begin{cases} \texttt{pv} \mapsto \texttt{void flood()\{ pt \}} \\ \texttt{pt} \mapsto \texttt{if(isWaterHigh()) waterOff(); else waterOn();} \\ \texttt{pw} \mapsto \texttt{void fire()\{ i1 \}} \\ \texttt{i1} \mapsto \texttt{if(isTemperatureHigh()) spriOn(); else spriOff();} \end{cases}$$

After a fire has been detected, it must be guaranteed that the water supply is not turned off; otherwise the building would burn down. This is not guaranteed with $\texttt{Fire} + \texttt{Flood}$; it depends on a race condition determined by the order of `fire`() and `flood`() execution.

The solution is to impose one more module, representing a feature interaction `FI`, denoted `Fire#Flood` in [5], whose alterations make `Flood` and `Fire` work correctly together. This is achieved by a shared VP `pv` and the use of overriding. `FI` replaces `Flood`'s fragment at `pv` with `FI`'s fragment:

$$\texttt{FI}: \begin{cases} \texttt{pv} \mapsto \texttt{void flood()\{ i2 \}} \\ \texttt{i2} \mapsto \texttt{if(isTemperatureHigh()) waterOn(); else \{ pt \}} \end{cases}$$

The occurrence of `pt` in the fragment of `i2` may be viewed as an `original` call to the respective method (e.g., [8]).

The composition of all three modules, $\texttt{FandF} = \texttt{FI} \rightarrowtriangle (\texttt{Fire} + \texttt{Flood})$, yields a correct implementation:

$$\texttt{FandF}: \begin{cases} \texttt{pw} \mapsto \texttt{void fire()\{ i1 \}} \\ \texttt{i1} \mapsto \texttt{if(isTemperatureHigh()) spriOn(); else spriOff();} \\ \texttt{pv} \mapsto \texttt{void flood()\{ i2 \}} \\ \texttt{i2} \mapsto \texttt{if(isTemperatureHigh()) waterOn(); else \{ pt \}} \\ \texttt{pt} \mapsto \texttt{if(isWaterHigh()) waterOff(); else waterOn();} \end{cases}$$

□

To state further laws, we call modules `m` and `n` *compatible*, in signs $\texttt{m} \downarrow \texttt{n}$, if their fragments coincide on their shared domains. Formally,

$$\texttt{m} \downarrow \texttt{n} \Leftrightarrow_{df} \forall\, \texttt{v} \in \texttt{dom(m)} \cap \texttt{dom(n)} :\ \texttt{m(v)} = \texttt{n(v)}\ .$$

All submodules of a module are pairwise compatible with each other. It follows that the properties below are equivalent:

$$\mathtt{m} \downarrow \mathtt{n} \;\Leftrightarrow\; \mathtt{m} \rightarrowtriangle \mathtt{n} = \mathtt{m} + \mathtt{n} \;\Leftrightarrow\; \mathtt{m} \rightarrowtriangle \mathtt{n} = \mathtt{n} \rightarrowtriangle \mathtt{m}$$

Module addition and overriding interact by the following laws, where the left implication $\Leftarrow$ means "provided".

$$\begin{array}{lll}
\mathtt{m} \rightarrowtriangle (\mathtt{n} + \mathtt{p}) = (\mathtt{m} \rightarrowtriangle \mathtt{n}) + (\mathtt{m} \rightarrowtriangle \mathtt{p}) & & \text{(left distributivity)} \\
(\mathtt{m} + \mathtt{n}) \rightarrowtriangle \mathtt{p} = \mathtt{m} \rightarrowtriangle (\mathtt{n} \rightarrowtriangle \mathtt{p}) & \Leftarrow\; \mathtt{m} \downarrow \mathtt{n} & \text{(sequentialisation)} \\
(\mathtt{m} + \mathtt{n}) \rightarrowtriangle \mathtt{p} = \mathtt{n} \rightarrowtriangle \mathtt{p} & \Leftarrow\; \mathtt{m} \downarrow \mathtt{n} \wedge \mathtt{m} \rightarrowtriangle \mathtt{n} = \mathtt{n} & \text{(absorption I)} \\
(\mathtt{m} + \mathtt{n}) \rightarrowtriangle \mathtt{p} = \mathtt{m} \rightarrowtriangle \mathtt{p} & \Leftarrow\; \mathtt{m} \downarrow \mathtt{n} \wedge \mathtt{n} \rightarrowtriangle \mathtt{p} = \mathtt{p} & \text{(absorption II)} \\
\mathtt{m} \rightarrowtriangle (\mathtt{n} + \mathtt{p}) = \mathtt{n} + (\mathtt{m} \rightarrowtriangle \mathtt{p}) & \Leftarrow\; \mathtt{dom}(\mathtt{m}) \cap \mathtt{dom}(\mathtt{n}) = \emptyset & \text{(localisation)}
\end{array}$$

The sequentialisation law means that a complex overriding may also be done by two successive simpler overridings. Absorption II, which is an immediate consequence of sequentialisation, allows simplifying a complex overriding by omitting the part that is "already there". Finally, localisation allows propagating an overriding to the submodule it really affects.

The previous laws for overriding dealt with immediate links from VPs to fragments. We now deal with preservation of transitive links under overriding. Let $\mathtt{dep}_\mathtt{m}^*$ be the reflexive, transitive closure of $\mathtt{dep}_\mathtt{m}$. For a module $\mathtt{m}$ and a VP $\mathtt{v} \in \mathtt{dom}(\mathtt{m})$ we define $\mathtt{deps}(\mathtt{v}, \mathtt{m}) =_{df} \{\mathtt{w} \mid \mathtt{v}\ \mathtt{dep}_\mathtt{m}^*\ \mathtt{w}\}$, i.e., the set of VPs on which $\mathtt{v}$ depends transitively, plus $\mathtt{v}$ itself. We now want to override or extend a module $\mathtt{n}$ with a module $\mathtt{m}$. If $\mathtt{n}$ does not alter the assignments of the VPs on which $\mathtt{v}$ transitively depends in $\mathtt{m}$ then the overriding/extension does not change the transitive dependence of $\mathtt{v}$ (cf. [23, 13]):

$$\left.\begin{array}{l}
\mathtt{deps}(\mathtt{v}, \mathtt{m} + \mathtt{n}) \;= \mathtt{deps}(\mathtt{v}, \mathtt{m}) \\
\mathtt{deps}(\mathtt{v}, \mathtt{m} \rightarrowtriangle \mathtt{n}) = \mathtt{deps}(\mathtt{v}, \mathtt{n})
\end{array}\right\} \;\Leftarrow\; \mathtt{deps}(\mathtt{v}, \mathtt{m}) \cap \mathtt{dom}(\mathtt{n}) = \emptyset\ .$$

3.3 Solving Module Equations

As discussed in [5], it is useful to be able to solve module equations. Subtraction and its relatives enable us to do so. Suppose that $\mathtt{m}$ is a submodule of $\mathtt{n}$, i.e., $\mathtt{m} \leq \mathtt{n}$ (see Equation (1)). Then the equation $\mathtt{m} + \mathtt{x} = \mathtt{n}$ has $\mathtt{x} = \mathtt{n} - \mathtt{m}$ as a solution. Moreover, this is the unique solution that is domain-disjoint from $\mathtt{m}$.[7]

Example 3.2 Consider a composition $\mathtt{comp} = \mathtt{a} + \mathtt{b} + \mathtt{c} + \mathtt{d}$ with pairwise domain-disjoint modules $\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}$. Then the equation $\mathtt{a} + \mathtt{x} + \mathtt{b} + \mathtt{d} = \mathtt{comp}$ has the unique solution $\mathtt{x} = \mathtt{c}$ domain-disjoint from $\mathtt{a} + \mathtt{b} + \mathtt{d}$. □

Note that the condition $\mathtt{dom}(\mathtt{m}) \subseteq \mathtt{dom}(\mathtt{n})$ is necessary for $\mathtt{m} + \mathtt{x} = \mathtt{n}$ to be solvable, because we need to have $\mathtt{dom}(\mathtt{m} + \mathtt{x}) = \mathtt{dom}(\mathtt{m}) \cup \mathtt{dom}(\mathtt{x}) = \mathtt{dom}(\mathtt{n})$, which implies $\mathtt{dom}(\mathtt{m}) \subseteq \mathtt{dom}(\mathtt{n})$. The above assumption $\mathtt{m} \leq \mathtt{n}$ implies that necessary condition. In fact, solvability of $\mathtt{m} + \mathtt{x} = \mathtt{n}$ conversely implies $\mathtt{m} \leq \mathtt{n}$, since

$$\mathtt{m} + \mathtt{n} = \mathtt{m} + (\mathtt{m} + \mathtt{x}) = (\mathtt{m} + \mathtt{m}) + \mathtt{x} = \mathtt{m} + \mathtt{x} = \mathtt{n}\ .$$

In short: $\mathtt{m} + \mathtt{x} = \mathtt{n}$ is solvable iff $\mathtt{m} \leq \mathtt{n}$.

[7] Another solution is $\mathtt{x} = \mathtt{n}$, since $\mathtt{m} \leq \mathtt{n}$ means $\mathtt{m} + \mathtt{n} = \mathtt{n}$. Such solutions are uninteresting.

Next, we have a brief look at equations involving overriding. Since

$$\mathtt{dom}(\mathtt{m} \mathbin{\rightarrowtriangle} \mathtt{n}) = \mathtt{dom}(\mathtt{n} \mathbin{\rightarrowtriangle} \mathtt{m}) = \mathtt{dom}(\mathtt{m}) \cup \mathtt{dom}(\mathtt{n}) \ ,$$

again $\mathtt{dom}(\mathtt{m}) \subseteq \mathtt{dom}(\mathtt{n})$ is necessary for $\mathtt{m} \mathbin{\rightarrowtriangle} \mathtt{x} = \mathtt{n}$ and $\mathtt{x} \mathbin{\rightarrowtriangle} \mathtt{m} = \mathtt{n}$ to be solvable. But by the definition of $\mathbin{\rightarrowtriangle}$, solvability of $\mathtt{m} \mathbin{\rightarrowtriangle} \mathtt{x} = \mathtt{n}$ even implies the stronger necessary condition $\mathtt{m} \leq \mathtt{n}$. In this case again $\mathtt{x} = \mathtt{n} - \mathtt{m}$ is the unique solution domain-disjoint from $\mathtt{m}$. A closer inspection shows that the same is the case for the equation $\mathtt{x} \mathbin{\rightarrowtriangle} \mathtt{m} = \mathtt{n}$. This means that it is sufficient to restrict interest to the solution of equations involving $+$.

3.4 Transformations

In this section we sketch another extension of SDA, intended to cope with some standard techniques in software refactoring (e.g., [4, 20]). Examples of such techniques are consistent renaming of methods or classes in a large software system. To stay at the same level of abstraction as before, we realize this by a mechanism for generally modifying the fragments in SDA modules.

By a *transformation* or *modification* or *refactoring* we mean a total function $\mathtt{T} : \mathtt{F}(\mathtt{V}) \to \mathtt{F}(\mathtt{V})$. By $\mathtt{T} \cdot \mathtt{m}$ we denote the *application* of $\mathtt{T}$ to a module $\mathtt{m}$. It yields a new module defined by

$$(\mathtt{T} \cdot \mathtt{m})(\mathtt{v}) =_{df} \begin{cases} \mathtt{T}(\mathtt{m}(\mathtt{v})) & \text{if } \mathtt{v} \in \mathtt{dom}(\mathtt{m}) \\ \text{undefined otherwise} \ . \end{cases}$$

We use the convention that $\cdot$ binds stronger than all other operators. The following properties hold:

$$\begin{array}{ll} (1)\ \mathtt{dom}(\mathtt{T} \cdot \mathtt{m}) = \mathtt{dom}(\mathtt{m}) \ , & (2)\ \mathtt{T} \cdot (\mathtt{m} + \mathtt{n}) = \mathtt{T} \cdot \mathtt{m} + \mathtt{T} \cdot \mathtt{n} \quad \Leftarrow \quad \mathtt{m} \downarrow \mathtt{n} \ , \\ (3)\ \mathtt{T} \cdot (\mathtt{m} - \mathtt{n}) = \mathtt{T} \cdot \mathtt{m} - \mathtt{T} \cdot \mathtt{n} \ , & (4)\ \mathtt{T} \cdot (\mathtt{m} \mathbin{\rightarrowtriangle} \mathtt{n}) = \mathtt{T} \cdot \mathtt{m} \mathbin{\rightarrowtriangle} \mathtt{T} \cdot \mathtt{n} \ . \end{array}$$

The proofs are straightforward calculations.

The analogue of Equation (2) is also used in the feature algebra of [1]; it entails that a transformation $\mathtt{T}$ is propagated and applied to all components of a composed module.

This applies, in particular, to transformations like method renaming; there $\mathtt{T}$ would be implemented using a global table with all the old-name/new-name associations which would be consistently applied in all submodules of the overall module under consideration. Note that, although $\mathtt{T}$ is supposed to be a total function on all fragments, it might well leave many of those unchanged, i.e., act as the identity on them.

Future work in this area will deal with further operators that reflect extended transformations concerning several modules, like moving methods from one module to another.

4 Using the Algebra

4.1 Projecting Out

Next, we deal with some aspects of modularity. In *classical modularity* there are physical files that define the feature modules and tools that compose them to produce a desired program. This is also known as *alternative-based variation*. Contrarily, *virtual modularity*, also known as *SYSGEN* (e.g. [14]) or *projectional*

variation is a preprocessor technology. A prominent representative of this technique is the *coloring* approach of [5, 15, 18]. The idea is simple: every document is painted in different colors, one color per feature. A color `C` that appears "inside" another color `D` indicates a feature interaction—`C` changes the source of `D` (denoted `C#D` in [5]). VPs are implicit in coloring. At every point in a document where coloring changes, an implicit VP is created. Figure 1 is not only an example for variation points, fragments and modules, but also for coloring: the code is colored white, light gray and dark gray. If a feature is not needed in a product, all code colored in the corresponding color (e.g., dark gray) is *projected out*, i.e., does not show up in the final product. Since one colors the entire code base of a product line, it is possible to compute the contents of SDA modules and their VPs.

We now show how this operation can be expressed in our algebra. Assume a module $\mathtt{m}$ and a subset $\mathtt{U} \subseteq \mathtt{dom}(\mathtt{m})$. Projection to $\mathtt{U}$ is supposed to hide everything that does not correspond to a VP in $\mathtt{U}$. This needs to be done in such a way that later the hidden parts can be uncovered again. Therefore, a corresponding operation should not remove the VPs outside $\mathtt{U}$ from $\mathtt{m}$. We preserve the hidden part $\mathtt{n} =_{df} \mathtt{m}|_{\mathtt{dom}(\mathtt{m})-\mathtt{U}}$ and temporarily work with the module $\mathtt{p} =_{df} [(\mathtt{dom}(\mathtt{m}) - \mathtt{U}) \mapsto \mathtt{0}] \rightarrow\!\triangleright\, \mathtt{m}$ which masks all VPs outside $\mathtt{U}$ with the default fragment. To restore the original module, i.e., to switch the masked parts back on again we use that $\mathtt{n} \rightarrow\!\triangleright\, \mathtt{p} = \mathtt{m}$.

The difference between restriction and projecting out is that in the former case all VPs outside $\mathtt{U}$ are removed and hence become external to the resulting module so that they would be considered as "fresh" there, whereas after projection they are still "known" and hence "protected" against unintentional change, so that they can be refilled later.

4.2 Introducing Wrappers

We can use overriding for adding a wrapper to a module. Let $\mathtt{m}$ be a module and $\mathtt{v} \in \mathtt{dom}(\mathtt{m})$ be the VP where we want to insert the wrapper. Choose a "fresh" VP $\mathtt{w} \notin \mathtt{dom}(\mathtt{m})$ and a non-trivial wrapper fragment $\mathtt{f}$ such that $\mathtt{VP}(\mathtt{f}) = \{\mathtt{w}\}$ and $\mathtt{w}$ occurs only once in $\mathtt{f}$. Then $\mathtt{w}$ marks the place in $\mathtt{f}$ where the original "contents" of $\mathtt{v}$ is to be put and thereby wrapped. The old contents $\mathtt{m}(\mathtt{v})$ of $\mathtt{v}$ is remembered by a link from $\mathtt{w}$ to it, so that it can be overridden by a link from $\mathtt{v}$ to the wrapper $\mathtt{f}$. Algebraically, the module with the wrapper is expressed by

$$\mathtt{m}' =_{df} ([\mathtt{v} \mapsto \mathtt{f}] \rightarrow\!\triangleright\, \mathtt{m}) + [\mathtt{w} \mapsto \mathtt{m}(\mathtt{v})] \ .$$

An illustration is given in Figure 3.

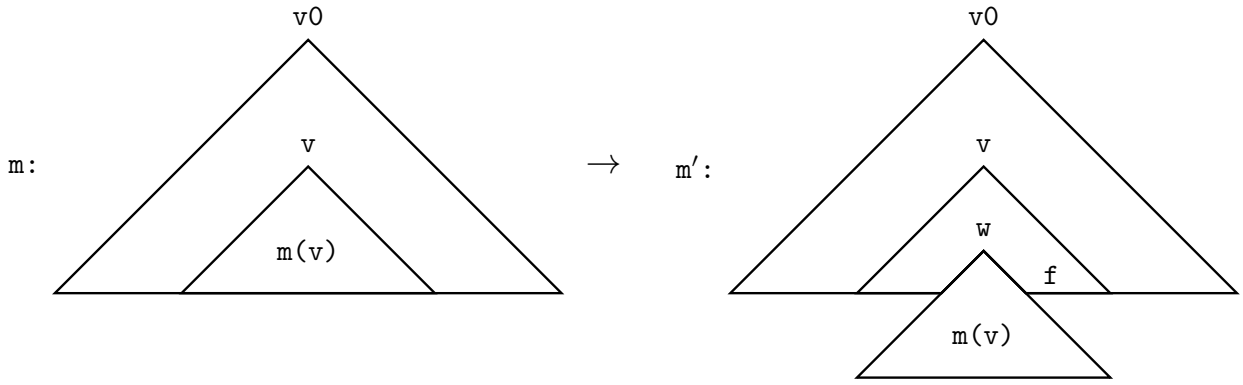


Fig. 3. Introducing a Wrapper (Splicing).

5 Small Case Study: Constructing Product Lines

There are often multiple ways how a finished product (i.e., a single fragment without further VPs) can result within a product line. This is, for example, the case when modules can be grouped into sets of features that are orthogonal to each other. Conversely, one may wish to refactor an existing monolithic system by decomposing it into directly re-usable or at least easily adaptable parts.

We illustrate this by taking up an example from [7]. Notationally we deviate a bit from that paper by not distinguishing "defining" and "applied" occurrences of VPs. All VP names start with `@` (for text processing reasons); in a module every VP is assigned the text indented to the position after the VP name. If no text appears to the right of a VP this means that the default is assigned to it.

We look at software for a basic calculator. It deals with arithmetic expressions composed by the operations addition and multiplication; expressions can be displayed and evaluated. First we present a program (or single module) `CALC`, given in [7] (see Figure 4). Formally, this is a module that has only one single root VP `@0` to which the complete program is assigned, without any VPs in its text. However, to prepare the restructuring to come, we already indicate a number of further VPs, to be read as comments at this level. For operator signs `op`, such as `+` or `*`, the abbreviation `?op?` stands for the string concatenation `+ "op" +`.

CALC

```
@0 abstract class Exp {
     @1 String print();
     @2 int eval();
   }

   class Int extends Exp {
     int v;
     Int(int a) { v=a; }
      @3 String print() { return v; }
      @4 int eval() { return v; }
   }

class Plus extends Exp {
  Exp l,r;
  Plus(Exp L, Exp R) {l=L; r=R;}
  @5 String print()
     { return l.print() ?+? r.print(); }
  @6 int eval()
     { return l.eval() + r.eval(); }
}

class Times extends Exp {
  Exp l,r;
  Times(Exp L, Exp R) {l=L; r=R;}
  @7 String print()
     { return l.print() ?*? r.print();}
  @8 int eval()
     { return l.eval() * r.eval(); }
}
```

Fig. 4. The module `CALC`

CALC contains sub-packets, which may have interest of their own. For example, a user may only want to deal with arithmetic expressions for addition. This can be achieved by grouping the parts of CALC into Base, Plus and Times as given in Figure 5 and then forming adequate sums of some of these. For instance, the product line TPL combines all three submodules: $\texttt{TPL} =_{df} \texttt{Base} + \texttt{Plus} + \texttt{Times}$. Using the restructuring preorder from Sect. 2.2, we have the relation $\texttt{CALC} \sqsubseteq \texttt{TPL}$ between the original code base CALC and TPL.

TPL:

Base:

```
@0 abstract class Exp {
     String print();
     int eval();
   }

   class Int extends Exp {
     int v;
     Int(int a) { v=a; }
     String print()
       { return v; }
     int eval()
       { return v; }
   }

   @+
   @*
```

Plus:

```
@+ class Plus extends Exp {
     Exp l,r;
     Plus(Exp L, Exp R)
       {l=L; r=R;}
     String print()
       { return l.print() ?+?
                r.print(); }

     int eval()
       { return l.eval() +
                r.eval(); }
   }
```

Times:

```
@* class Times extends Exp {
     Exp l,r;
     Times(Exp L, Exp R)
      {l=L; r=R;}
     String print()
       { return l.print() ?*?
                r.print();}
     int eval()
       { return l.eval() *
                r.eval(); }
   }
```

Fig. 5. The module TPL

A second restructuring reflects that some users may only wish to evaluate expressions but not to print them. This can be accommodated by decomposing the original code base CALC into a Core module and two optional modules Print and Eval, whose definitions are given in Figure 6. A new product line $\texttt{CPE} =_{df} \texttt{Core} + \texttt{Print} + \texttt{Eval}$ combines these three submodules.

The original product line CALC relates to the new one by $\texttt{CALC} \sqsubseteq \texttt{CPE}$.

Finally we can form a common restructuring of CPE and TPL. This SPL is presented in Figure 7 and corresponds to the EPL matrix of [7]. We use two sets of feature names, $\texttt{F} =_{df} \{\texttt{core}, \texttt{print}, \texttt{eval}\}$ and $\texttt{G} =_{df} \{\texttt{base}, \texttt{plus}, \texttt{times}\}$. We use lower-case names here, since these are not module names but will serve as indices for a matrix EPLmat, where every entry $\texttt{EPLmat}_{\texttt{ij}}$ ($\texttt{i} \in \texttt{F}, \texttt{j} \in \texttt{G}$) is a submodule. Then setting $\texttt{EPL} =_{df} \sum_{\texttt{i} \in \texttt{F}} \sum_{\texttt{j} \in \texttt{G}} \texttt{EPLmat}_{\texttt{ij}}$ we obtain our finest restructuring of the original program CALC. As a consequence we have

$$\texttt{CPE} \sqsubseteq \texttt{EPL} \wedge \texttt{TPL} \sqsubseteq \texttt{EPL} \ .$$

If we sum the columns of EPLmat we obtain the constituent modules of OPL, while the row sums give the constituent modules of TPL. Hence the small modules in EPLmat can be considered as the elementary features in this product line. Of course, by transitivity of $\sqsubseteq$, the original code base is an element of this product line too, i.e., $\texttt{CALC} \sqsubseteq \texttt{EPL}$; but EPL offers many more products.

6 Related Work

Ideas similar to those of SDA can be found in [9], where elements of UML models could be tagged with feature predicates. Given a set of selected features, an element is removed from a model if its predicate is false.

CPE:

Core:	Print:	Eval:

@0 abstract class Exp {
  @1
  @2
}
 	
class Int extends Exp {
  int v;
  Int(int a) { v=a; }
  @3
  @4
}
 	
class Plus extends Exp {
  Exp l,r;
  Plus(Exp L, Exp R)
  {l=L; r=R;}
  @5
  @6
}
 	
class Times extends Exp {
  Exp l,r;
  Times(Exp L, Exp R)
  {l=L; r=R;}
  @7
  @8
}
	@1 String print();
	
	@3 String print()
	{ return v; }
	
	@5 String print()
	{ return l.print()
	?+? r.print(); }
	
	@7 String print()
	{ return l.print()
	?*? r.print(); }
		@2 int eval();
		
		@4 int eval()
		{ return v; }
		
		@6 int eval()
		{ return l.eval()
		+ r.eval(); }
		
		@8 int eval()
		{ return l.eval()
		* r.eval(); }

Fig. 6. The module CPE

Derivatives [21] were the first identified building blocks of feature modules. Unfortunately, the mathematics of derivatives was incomplete, as composition of derivatives was not associative. This made it impossible to algebraically calculate the results of feature splitting (replacing T with R × S if T is split into features R and S) and feature merging (replacing R × S with T). CIDE [18] showed a simple and elegant way to visualize features and their interactions, resulting in the coloring algebra, which does support splitting and merging.

Other algebras for feature-based composition, such as [1, 22], focus on the internal structure of modules. The algebra presented in [1] is (to our knowledge) the first that dealt with feature replication. It uses the algebraic law of *distant idempotence* (a form of idempotence where adjacency of identical features is not required). Feature composition is not commutative and there is no operation of subtraction on feature modules (called feature structure trees there). An algebra that also captures replicas is presented in [16]. However, this algebra only works at a semantic level and cannot cope with real source code, as SDA does.

The *Choice Calculus (CC)* [25] offers an interesting and alternative approach to our work. Among the goals of CC are to integrate classical and virtual modularity, but to do so in the context of a formal programming language. Large-scale fragments can be placed in modules of their own, while small-scale fragments (suitable for annotations) can be embedded into other modules. As in coloring and `ifdef` preprocessing, variation points are implicit. The key difference between our work and CC is that the issues of classical and virtual modularity are not limited to a fixed set of programming languages.

Delta Oriented Programming (DOP) [24] is another interesting language-based approach within our field. Delta modules are qualified to be composed into a product when the corresponding `where` clause is satisfied. Such a clause is a propositional formula over features, namely the conjunction of the feature formulas that arise in coloring. Disjunction allows a single module to be reused in different contexts (rather than requiring a module to be replicated for each context). Negation seems to offer a more general way for defining alternatives. Understanding this connection will be the subject of future work. Delta modules also have `after` clauses,

EPLmat

	core	print	eval

base	@0 abstract class Exp {
	  @1
	  @2
	}
	
	class Int extends Exp {
	  int v;
	  Int(int a) { v=a; }
	  @3
	  @4
	}
	@*
	@+
		@1 String print();
		  eval print core
		
		@3 String print()
		  { return v; }
			@2 int eval();
			
			@4 int eval()
			  { return v; }

plus	@+ class Plus extends Exp {
	  Exp l,r;
	  Plus(Exp L, Exp R)
	  {l=L; r=R;}
	  @5
	  @6
	}
		@5 String print()
		  { return l.print()
		  ?+? r.print(); }
			@6 int eval()
			  { return l.eval()
			  + r.eval(); }

times	@* class Times extends Exp {
	  Exp l,r;
	  Times(Exp L, Exp R)
	  {l=L; r=R;}
	  @7
	  @8
	}
		@7 String print()
		  { return l.print()
		  ?*? r.print(); }
			@8 int eval()
			  { return l.eval()
			  * r.eval(); }

Fig. 7. The matrix `EPLmat`

which specify a partial order in which to compose them. We express the composition order of modules explicitly with our overriding operator $\rightarrow\!\triangleright$.

A project close in spirit and ideas is Kästner's *CIDE* [18, 19, 2]. It started out with a tool for software product line development (esp. analyzing and decomposing legacy code), following the paradigm of virtual separation of concerns. For this it offers the possibility of distinguishing fragments within the original code by different background colors. CIDE is also a compiler (and an IDE), closely related to conditional compilation with preprocessors. Additionally it analyzes the deep structure of the code and hence guarantees syntactic correctness as well as type correctness of all generated products. Hence CIDE already provides, at the tool level, some of the functionality that our algebra treats at the abstract level.

7 Conclusions and Outlook

We have presented a structured document algebra not only in concept, but also at work on some essential phenomena of large-scale software construction. We hope to have convinced the reader that SDA is both concise and precise, and comes with a number of useful laws. These can be used to construct and reason about structured modules in an algebraic fashion.

The main aim is to provide a formal basis for governing variability in large interconnected collections of documents, in particular ones that define SPLs. SDA can be used to implement tools which relieve developers from managing variability manually. In particular, it provides a basis for precise reasoning about complex

and error-prone operations such as subtraction, overriding, and various refactorings. Currently, work on a pilot implementation is under way.

The algebra has interesting connections to other structures, such as the pointer algebras presented in [13, 23] and to the concept of the demonic join or compatible union [3, 11].

While so far SDA is presented along a concrete mathematical model, an abstraction to more general concepts like monoids, semirings and modules in the linear algebra sense is under way.

Finally, so far SDA has a very syntactic flavor, since the nature of fragments is left open. However, it is possible to work with fragments that have a semantic character, such as functions from valuations of VPs in some semantic model to a semantic value. We have sketched this in an Appendix; the elaboration of these ideas will be the subject of future investigations.

Acknowledgments We are grateful to the anonymous referees for helpful comments and suggestions. We gratefully acknowledge support for this work by NSF grants CCF 0724979 and OCI-1148125, as well as by DFG grant MO 690/7-2. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

1. Apel, S., Lengauer, C., Möller, B., Kästner, C.: An algebraic foundation for automatic feature-based program synthesis. SCP 75(11), 1022–1047 (2010)
2. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines. Springer (2013)
3. Backhouse, R., van der Woude, J.: Demonic operators and monotype factors. Mathematical Structures in Computer Science 3(4), 417–433 (1993)
4. Batory, D.: Program refactorings, program synthesis, and model-driven design. In: Krishnamurthi, S., Odersky, M. (eds.) Compiler Construction. LNCS, vol. 4420, pp. 156–171. Springer (2007)
5. Batory, D., Höfner, P., Kim, J.: Feature interactions, products, and composition. In: Generative Programming and Component Engineering (GPCE'11). pp. 13–22. ACM (2011)
6. Batory, D., Höfner, P., Möller, B., Zelend, A.: Features, modularity, and variation points. In: Feature-Oriented Software Development. pp. 9–16. ACM (2013)
7. Batory, D., Shepherd, C.: Product lines of product lines, Department of Computer Science, University of Texas at Austin, submitted (2013)
8. Bergel, A., Ducasse, S., Nierstrasz, O.: Classbox/J: Controlling the scope of change in Java. SIGPLAN Not. 40(10), 177–189 (2005)
9. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: Glück, R., Lowry, M. (eds.) Generative Programming and Component Engineering (GPCE'05). LNCS, vol. 3676, pp. 422–437. Springer (2005)
10. Delaware, B., Cook, W.R., Batory, D.S.: Product lines of theorems. In: Lopes, C.V., Fisher, K. (eds.) Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011. vol. 10, pp. 595–608. ACM (2011)
11. Desharnais, J., Belkhiter, N., Sghaier, S.B.M., Tchier, F., Jaoua, A., Mili, A., Zaguia, N.: Embedding a demonic semilattice in a relational algebra. Theor. Comput. Sci. 149(2), 333–360 (1995)
12. Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, P.: Traits: A mechanism for fine-grained reuse. ACM Trans. Program. Lang. Syst. 28(2), 331–388 (2006)

13. Ehm, T.: The Kleene Algebra of Nested Pointer Structures: Theory and Applications. Ph.D. thesis, Fakultät für Angewandte Informatik, Universität Augsburg (2003)
14. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison Wesley (2004)
15. Heidenreich, F.: Towards systematic ensuring well-formedness of software product lines. In: Feature-Oriented Software Development. ACM (2009)
16. Höfner, P., Khedri, R., Möller, B.: Feature algebra. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) Formal Methods (FM'06). LNCS, vol. 4085, pp. 300–315. Springer (2006)
17. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-oriented domain analysis (foda) feasibility study. Tech. Rep. CMU/SEI-90-TR-021, Carnegie Mellon Software Engineering Institute, Carnegie Mellon University (1990)
18. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in software product lines. In: Schäfer, W., Dwyer, M., Gruhn, V. (eds.) Conference on Software Engineering (ICSE 2008). ACM (2008)
19. Kästner, C.: Virtual separation of concerns: toward preprocessors 2.0. Ph.D. thesis, Otto von Guericke University Magdeburg (2010)
20. Kuhlemann, M., Batory, D., Apel, S.: Refactoring feature modules. In: Edwards, S., Kulczycki, G. (eds.) Formal Foundations of Reuse and Domain Engineering. LNCS, vol. 5791, pp. 106–115. Springer (2009)
21. Liu, J., Batory, D.S., Lengauer, C.: Feature oriented refactoring of legacy applications. In: Osterweil, L.J., Rombach, H.D., Soffa, M.L. (eds.) 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006. pp. 112–121. ACM (2006)
22. Lopez-Herrejon, R., Batory, D., Lengauer, C.: A disciplined approach to aspect composition. In: Partial Evaluation and Semantics-based Program Manipulation (PEPM 06). ACM (2006)
23. Möller, B.: Towards pointer algebra. SCP 21(1), 57–90 (1993)
24. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Bosch, J., Lee, J. (eds.) Software Product Lines: Going Beyond. LNCS, vol. 6287, pp. 77–91. Springer (2010)
25. Walkingshaw, E., Erwig, M.: A calculus for modeling and implementing variation. In: Generative Programming and Component Engineering. pp. 132–140. ACM (2012)
26. Wirsing, M.: Algebraic specification. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B). pp. 675–788 (1990)

Appendix: Sketch of a Semantic Model of SDA

For historical reasons, in particular in honour of Martin Wirsing, we have chosen an approach based on the notions of algebraic specification [26]. Other variants are conceivable, too.

Basic Definitions. Assume a signature $\Sigma = (\mathtt{s}, \mathtt{F})$ with a sort $\mathtt{s}$ and a set $\mathtt{F}$ of operators $\mathtt{f} : \mathtt{s}^{\mathtt{n}} \to \mathtt{s}$. (For simplicity we restrict ourselves to the one-sorted case; the generalisation to many sorts is straightforward.) The number $\mathtt{n}$ is called the *arity* of $\mathtt{f}$. An operator of arity $\mathtt{0}$ is called a *constant operator*. We also assume a set $\mathtt{V}$ of *variation points* such that $\mathtt{V}$ is disjoint from the set of constant operators of Σ (in logic the elements of $\mathtt{V}$ would be called *variables*). The set $\mathtt{T}(\Sigma, \mathtt{V})$ of *terms* over Σ and $\mathtt{V}$ is defined as usual. If $\mathtt{VP}(\mathtt{t}) = \emptyset$ then $\mathtt{t}$ is called *closed* or a *ground term*. For describing SDA modules we assume a signature with two special constant symbols $\mathtt{0}, \lightning$.

A *Σ-algebra* is a pair $\mathtt{A} = (\mathtt{s}^{\mathtt{A}}, \mathtt{F}^{\mathtt{A}})$ where $\mathtt{s}^{\mathtt{A}}$ is a nonempty *carrier set* and $\mathtt{F}^{\mathtt{A}} = \{\mathtt{f}^{\mathtt{A}} \mid \mathtt{f} \in \mathtt{F}\}$ is a set of *operations* $\mathtt{f}^{\mathtt{A}} : (\mathtt{s}^{\mathtt{A}})^{\mathtt{n}} \to \mathtt{s}^{\mathtt{A}}$ associated with the $\mathtt{f} : \mathtt{s}^{\mathtt{n}} \to \mathtt{s} \in \mathtt{F}$. The set $\mathtt{T}(\Sigma, \mathtt{V})$ can be made into a Σ-

algebra in the usual way. A *valuation* of $\mathtt{V}$ in a Σ-algebra $\mathtt{A}$ (also called an *environment*) is a partial function $\mathtt{e} : \mathtt{V} \rightsquigarrow \mathtt{s}^{\mathtt{A}}$. The set of all environments is denoted by $\mathcal{E}^{\mathtt{A}}$.

Syntactic Modules. A *syntactic fragment* simply is an element of $\mathtt{T}(\Sigma, \mathtt{V})$. By this, all VPs are fragments, too. A *syntactic module* is an environment $\mathtt{m} : \mathtt{V} \rightsquigarrow \mathtt{T}(\Sigma, \mathtt{V})$ from $\mathcal{E}^{\mathtt{T}(\Sigma,\mathtt{V})}$. The function $\mathtt{single\_fill}(\mathtt{t}, \mathtt{m})$ is defined inductively as syntactic substitution:

1. If $\mathtt{t} = \mathtt{v} \in \mathtt{V}$ is a VP then

$$\mathtt{single\_fill}(\mathtt{t}, \mathtt{m}) =_{df} \begin{cases} \mathtt{e}(\mathtt{v}) \text{ if } \mathtt{v} \in \mathtt{dom}(\mathtt{e}) \ , \\ \mathtt{v} \quad \text{otherwise} \ . \end{cases}$$

2. If $\mathtt{t} = \mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ then

$$\mathtt{single\_fill}(\mathtt{t}, \mathtt{m}) =_{df} \mathtt{f}(\mathtt{single\_fill}(\mathtt{t}_1, \mathtt{m}), \ldots, \mathtt{single\_fill}(\mathtt{t}_n, \mathtt{m})) \ .$$

   For a constant operator $\mathtt{f}$ hence $\mathtt{single\_fill}(\mathtt{t}, \mathtt{m}) = \mathtt{f}$.

Term Evaluation. For a Σ-algebra $\mathtt{A}$ we define inductively the *evaluation* $[\![\_]\!] : \mathtt{T}(\Sigma, \mathtt{V}) \to (\mathcal{E}^{\mathtt{A}} \rightsquigarrow \mathtt{s}^{\mathtt{A}})$ that assigns to every term a value using an environment, if possible.

– For VP $\mathtt{v}$ we set

$$[\![\mathtt{v}]\!](\mathtt{e}) =_{df} \begin{cases} \mathtt{e}(\mathtt{v}) & \text{if } \mathtt{v} \in \mathtt{dom}(\mathtt{e}) \ , \\ \text{undefined} & \text{otherwise} \ . \end{cases}$$

– For other terms we set

$$[\![\mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_n)]\!](\mathtt{e}) =_{df} \begin{cases} \mathtt{f}^{\mathtt{A}}(\mathtt{u}_1, \ldots, \mathtt{u}_n) & \text{if all } [\![\mathtt{t}_i]\!](\mathtt{e}) \text{ are defined and } \mathtt{u}_i = [\![\mathtt{t}_i]\!](\mathtt{e}) \ , \\ \text{undefined} & \text{otherwise} \ . \end{cases}$$

Hence for a constant symbol $\mathtt{f}$ we have $[\![\mathtt{f}]\!](\mathtt{e}) = \mathtt{f}^{\mathtt{A}}$ for all $\mathtt{e}$.

This definition entails what is called the *Coincidence Lemma* in logic; the proof is a straightforward induction.

Lemma 7.1 *If two environments* $\mathtt{e}, \mathtt{e}'$ *agree on the VPs of a term* $\mathtt{t}$*, i.e., if* $\mathtt{e}|_{\mathtt{VP}(\mathtt{t})} = \mathtt{e}'|_{\mathtt{VP}(\mathtt{t})}$*, then* $[\![\mathtt{t}]\!](\mathtt{e}) = [\![\mathtt{t}]\!](\mathtt{e}')$.

Semantic Modules. The idea is now to make the mapping $[\![\mathtt{t}]\!] : \mathcal{E}^{\mathtt{A}} \rightsquigarrow \mathtt{s}^{\mathtt{A}}$ somehow into a *semantic fragment* corresponding to the syntactic object $\mathtt{t}$ and to define a semantic module as a partial function from VPs to semantic fragments. However, this is not entirely straightforward, since we need to have VP information in some way to still be able to talk about cycle-freeness of modules. The solution proposed here is to enrich a semantic fragment by a set of VPs it "administers". A *semantic fragment* over a set $\mathtt{V}$ of VPs and a Σ-algebra $\mathtt{A}$ is a pair $(\mathtt{W}, \mathtt{g})$ where $\mathtt{W} \subseteq \mathtt{V}$ and $\mathtt{g} : \mathcal{E}^{\mathtt{A}} \rightsquigarrow \mathtt{s}^{\mathtt{A}}$ satisfies the *coincidence property* on $\mathtt{W}$:

$$\forall\, \mathtt{e}, \mathtt{e}' : \mathtt{e}|_{\mathtt{W}} = \mathtt{e}'|_{\mathtt{W}} \Rightarrow \mathtt{g}(\mathtt{e}) = \mathtt{g}(\mathtt{e}') \ .$$

We set $\mathtt{VP}(\mathtt{W},\mathtt{g}) =_{df} \mathtt{W}$ and $\mathtt{ev}(\mathtt{W},\mathtt{g}) =_{df} \mathtt{g}$. The dependence relation for semantic modules uses this definition of VP.

Every VP v can be made into the fragment $(\{\mathtt{v}\}, \lambda \mathtt{e} \,.\, 0^{\mathtt{A}})$.

A *semantic module* is a partial function from V to the set of semantic fragments. We can translate every syntactic module m into a semantic one called $\hat{\mathtt{m}}$ by setting $\hat{\mathtt{m}}(\mathtt{v}) =_{df} (\mathtt{VP}(\mathtt{m}(\mathtt{v})), [\![\mathtt{m}(\mathtt{v})]\!])$. If m is cycle-free then so is $\hat{\mathtt{m}}$.

The Single Fill Function for Semantic Modules. To make SDA work for semantic modules we have to define the `single_fill` function suitably:

$$\mathtt{single\_fill}((\mathtt{W},\mathtt{g}),\mathtt{m}) =_{df} ((\mathtt{W} - \mathtt{dom}(\mathtt{m})) \cup \mathtt{Z}, \lambda \mathtt{e} \,.\, \mathtt{g}(\mathtt{e}')) \ ,$$

where $\mathtt{Z} =_{df} \bigcup_{\mathtt{v} \in \mathtt{dom}(\mathtt{m})} \mathtt{VP}(\mathtt{m}(\mathtt{v}))$ and

$$\mathtt{e}'(\mathtt{v}) =_{df} \begin{cases} \mathtt{m}(\mathtt{v})(\mathtt{e}) & \text{if } \mathtt{v} \in \mathtt{dom}(\mathtt{m}) \ , \\ \mathtt{e}(\mathtt{v}) & \text{otherwise} \ . \end{cases}$$