Int J Software Informatics, Volume 5, Issue 1–2 (2011), Part II, pp. 231–244 E-mail: ijsi@iscas.ac.cn
International Journal of Software and Informatics, ISSN 1673-7288 http://www.ijsi.org

Tel: +86-10-62661040

Feature-Oriented System Design and Engineering

Christian Lengauer and Sven Apel

(University of Passau, Germany)

Abstract This is a personal appreciation and snapshot view of Manfred Broy's contributions to the research area of feature-oriented system design and engineering. We sketch the algebraic approach to the area and relate Broy's work to it. To give it a concrete context, we compare it with our own work on feature orientation. There are a number of correspondences and some differences: Broy works at a higher level of abstraction, the specification level, we at a level closer to the software structure, the programming level. We put more emphasis on the concept of program similarity than Broy does.

Key words: component; feature; program algebra; service; software product line; system specification

Lengauer C, Apel S. Feature-Oriented system design and engineering. *Int J Software Informatics*, **Vol.5, No.1-2 (2011), Part II: 231–244. http://www.ijsi.org/1673-7288/5/i82.htm**

1 System Design and Engineering

Early in computing, in the 1950s and 1960s, computer programs were monolithic, individual entities. As their size and functionality grew, the need for a modular structure arose and concepts such as abstract data types, objects, and processes were invented. Different components could be developed by different systems engineers and, if their interfaces were specified cleanly, the collection of all components would work together correctly as a system. The paradigm of object orientation, also coined early on (in the mid-Sixties) with the invention of the programming language Simula, facilitated the reuse of previously constructed specifications in the extension of system functionality.

The concept of a component was initially quite implementation-bound. As, in the Nineties, systems became yet more complex, modularity was required at a higher level of abstraction, closer to the application domain. There are several names for this notion – prevalently it has been called a service or a feature. There is no common agreement on what precisely a service or a feature is and in how the two differ. We will speak here of features, although Broy speaks mainly of services. For the purposes of this paper, we mean roughly the same thing.

This project is funded by the Deutsche Forschungsgemeinschaft (DFG).
Corresponding author: Christian Lengauer, Email: lengauer@fim.uni-passau.de
Received 2010-09-09; Accepted 2011-01-03; Final revised version 2011-02-09.

2 Feature Orientation

2.1 Features

A feature is a unit of functionality as observed by the user or stakeholder. In an automobile, features could be, for instance, the wipers or brakes or the navigation system (i.e., their respective functionality), in a database system the memory management or the transaction management. In a phone system, they could be different ways of handling a call when another call is being conducted (e.g., forwarding, waiting, or cancelling), in a video recorder the option of preprogrammed recording, etc. The important issue –and one major complication of feature-oriented system design– is that what appears as a unit to the user does not necessarily translate to a component in the implementation – often not even to a collection of components. The general situation is as follows.

A system is viewed as a collection of features (and nothing else!). The addition of a new feature to the system may cause:

- the addition of new components to the implementation or
- the extension or modification of components that have already been introduced by previously added features.

In the latter case, one says that the feature is crosscutting the modular structure of the implementation. This crosscutting nature poses a serious challenge to reliable feature-oriented system design.

Another serious challenge arises from the influences that different features can have on each other. It may be that a feature (e.g., of a Web browser) can only exist in the presence of some other feature (e.g., safe data transmission requires an encryption protocol) or that two features exclude each other (e.g., only one of a choice of available rendering engines can be present). Such conditions can be specified by a feature model. It may also be that the presence of one feature alters the behaviour of another, possibly, inadvertently. This phenomenon is called a feature interaction. If the feature interaction is desired, we speak of a feature cooperation, if not, of feature interference. As an example of feature interference, in a phone system, there are usually two alternative ways of handling an incoming call when the line is busy: either enqueue the call in a wait queue or forward the call to another number. Selecting both features simultaneously may impair the system's state and integrity.

2.2 Feature composition

In feature-oriented system design, a system is developed by composing features; composition is the central –if not only– operation needed. It must address all the challenges mentioned previously: it must respect the feature model and handle crosscutting and feature interaction. We use the bullet ($\bullet$) to denote feature composition.

One way of modelling feature composition formally is as a mapping that takes a system and a feature to be added and returns the system with the feature added:

$$\bullet : S \times F \longrightarrow S$$

A system with any number of features, f_1 to f_n, is built by starting with the empty system, denoted $\emptyset$, and adding features one at a time, i.e., by applying feature composition repeatedly. Here, we choose to develop sequences of compositions from left to right[1)]:

$$(\cdots((\emptyset \bullet f_1) \bullet f_2) \bullet \cdots f_n)$$

2.3 Feature algebra

The formal approach to system design and engineering has been at the heart of Broy's work from the start. His quest has been not only to drive this approach forward academically but also to convince industry and commerce that the use of formal models has significant benefits right now and to help cushion and support the use of formal models with design tools.

Feature-oriented system design can only succeed with a formal approach. In large systems, the challenges of crosscutting and feature interaction cannot be seriously addressed informally.

Both Broy and we take an algebraic approach. An algebra comprises sets of data, the operations on the data, and laws which the operations must satisfy. In feature-oriented system design, features and the systems they make up are the data, composition is the central operation. Composition has certain algebraic properties. Additional properties, not imposed by the algebra, may be imposed by the feature model; e.g., it may forbid the composition of certain features.

One can adapt the feature algebra as needed. For example, when one views a system, i.e., a composition of features, again as a feature, the composition operator must map two source features to the composed target feature:

$$\bullet : F \times F \longrightarrow F$$

and can be made associative:

$$(f_1 \bullet f_2) \bullet f_3 \quad = \quad f_1 \bullet (f_2 \bullet f_3)$$

That is, the order in which features are being added can be varied. Note that, in general, feature composition will not be commutative:

$$f_1 \bullet f_2 \quad \neq \quad f_2 \bullet f_1$$

That is, the positioning of the features in the chain of compositions will not be variable. But, in special cases, it can be.

An algebra is a wonderful mathematical device for the precise study of varying properties of features and feature composition and for the comparison of their effects on system design. In addition, there is the potential of automated support.

2.4 Automated support

A formal model has two major benefits: it provides a precise reference and it can be implemented. The implementation can be used to derive instances of the model automatically.

---

1) Elsewhere, we go from right to left[1].

In system design, automated support has become a major requirement. One has been realizing increasingly that support tools not based on formal models are difficult to implement, maintain, and trust.

The use of a feature algebra has three benefits for automated system generation:

- The generator can handle the issues of crosscutting and feature interaction reliably.
- By varying the algebra, a variety of generators can be produced easily.
- Based on domain knowledge and the algebraic laws, the generator can exploit feature-algebraic equations to optimize the corresponding system design.

2.5 Product lines

With feature orientation, the similarity of systems can be exploited conveniently. The set of all systems that can be composed from a given set of features is called a product line; the systems are its members or products. The more features two systems have in common, the more similar they are. One can derive one system from another by removing certain features and adding others[2].

The concept of a product line is particularly useful when one would like to offer a system in a, possibly large, number of variations. A product line consists of a set of features and a feature model. The feature model describes the valid combinations of features that define products. A by now classic example is the production line of a car assembly, on which hardly ever two identical products are assembled. Another example familiar to most of us is an operating system, which varies according to the platform on which it is being installed and, possibly, according to the needs of the applications and customers.

2.6 Feature interaction

One main issue of quality assurance in feature orientation is the problem of feature interaction[3]. Typical questions are:

- *Feature referencing*: Does the presence of a feature require the presence of some other feature?
- *Feature exclusion*: Are two features alternatives in the sense that at most one of them can be present?
- *Feature interference*: Does the presence of one feature alter the behaviour of some other feature in an undesirable way?
- *Feature cooperation*: Do two features that work well in isolation require additional functionality to work correctly together?

These questions must be addressed whether one wants to produce just one system or several similar ones, whether one is interested in product lines or not.

The four issues just mentioned are of primary interest to the stakeholder. Two other issues are of interest to the feature engineer, the one who develops a feature set or product line:

- *Feature refinement*: Can the specification of features be developed by adding detail step by step? Is there an abstraction hierarchy to reveal or hide details of features?

- *Feature modularity*: Can features be specified in isolation? If two features interfere, can features be added which repair the interference?

Complete feature modularity entails that a system or product line can be specified by only a set of features and nothing else. This is ultimate feature orientation.

3 Broy's Approach

Broy and his group have been developing a set of formal methods, called FOCUS[4]. FOCUS can be used for the specification and refinement of the components of a system and their interactions. The various components communicate via data streams, which they exchange along interconnecting channels. The behaviour of a component is specified as a stream-processing function, the entire system as a composition of stream-processing functions. An independent, in-depth appreciation of the goals and benefits of FOCUS can be found in another paper in this special issue[5].

Broy models a service as a partial stream-processing function. That is, it may be undefined on some inputs. The concept of a subservice is used to decompose a larger service into parts. One calls this a hierarchical decomposition. S' is a subservice of S if it exhibits a subset of the behaviours of S. S' cannot do more than S; typically, it does less. This is in contrast to the subtype concept in object orientation, where a subtype S' provides additional functionality compared to its supertype S.

The decomposition can violate the property of self-containedness of a subservice. That is, it may be that a subservice requires input or produces output that is not captured by its interface. One might consider this input or output interference. A subservice that exhibits no interference in its input and output is called faithful.

To deal with unfaithful subservices, Broy introduces the concept of a mode[6]. A mode is an additional interface that captures the nasty inputs and outputs to make the subservice non-interfering. When composing two subservices, one first captures their interference in modes and then composes the resulting, self-contained subservices with a benign operator that expects non-interfering operands.

The introduction of modes is a way to increase feature modularity. The imposition of modes can make a feature modular, i.e., fully specified by the behaviour via its interface, which otherwise would not be.

One interest of Broy has been the specification of automotive products[7,8]. Of course, FOCUS can be applied not only to automotive systems but also to other systems with similar requirements, such as telecommunication systems, enterprise management systems, and what have you.

The most powerful tool for the support of FOCUS is called AUTOFOCUS[9]. A specification can be entered conveniently via a graph editor. A stream-processing function is represented by a box, a stream-carrying channel by an arrow. Semantics is added by defining –also GUI-supported– an automaton in a tabular form. The automaton can be displayed graphically. Other forms of more partial semantic specifications are also provided, e.g., sets of communication sequences.

The formal specification in FOCUS facilitates a check of desirable properties of the system's behaviour such as the absence of deadlocks or the strong causality of all components, i.e., the fact that any component's output is influenced only by the input which the component received strictly before the output is produced. The use of AUTOFOCUS makes this check easy for finite-state systems: via automatic model checking. For other applications, there is a connection to the proof assistant Isabelle[10].

One common way of adding structure to a system is by refinement, and Broy has addressed this concept in depth[4]. A refinement adds requirements to a specification. The purpose of a refinement is to reduce non-determinism and partiality on the way to an implementation. The ideal endpoint of a refinement chain is a deterministic, efficient implementation that exhibits the expected behaviour on all possible inputs. AUTOFOCUS offers a mechanism to check the validity of a refinement.

If fully and correctly specified, the behaviour of the system can be simulated by its automaton in AUTOFOCUS. In some cases (e.g., software systems), the automaton may serve as the actual implementation, in others (e.g., automobiles), it may serve as a reference and as a platform for experiments.

4 Our Approach

Like Broy, we take a language-independent approach. However, while he refers to a fixed specification model, stream-processing functions, we allow a variety of models. In fact, one of our central goals is to make our approach model-extensible, i.e., to enable the addition of new models and languages as the need arises. A language which qualifies for addition must be what we call feature-ready, i.e., it must provide a sufficient amount of structure to facilitate the definition and manipulation of features. One central requirement is that each component be identifiable by a unique name. But neither object orientation, nor an imperative semantics, nor even an executable semantics are required.

To reiterate: Broy takes the view of a feature engineer whose goal is to specify an individual system interactively and use software tools to check its correctness and simulate its behaviour. Our goal is to exploit the similarity of the members of a software product line and be able to generate arbitrary individual members automatically. One might observe that the former is a preqrequisite of the latter, and this is correct. Our goal will only be attained on well delineated software product lines, whose members have many similarities and whose variabilities can be captured precisely and reasonably simply. Broy's approach applies to a wider range of problems. However, we believe that some of the examples favoured by him, such as telecommunication systems or automobiles, qualify to be made into product lines.

Broy concentrates on the design of a system as a composition of services. His main design concepts are the refinement and hierarchical decomposition of specifications[4]. We concentrate on the implementation and its use. We start at the point at which the development of the structure of a set of features has come to a successful end, i.e., we assume a set of features that has been specified textually, either in a specification language or in a programming language. Our model is a syntax tree representing the text describing a feature, a so-called feature structure tree.

Our tool suite FEATUREHOUSE[11] can handle, for instance, the languages Java,

C#, C, Haskell, JavaCC, Python, and Alloy. The example of Alloy[12] shows that FeatureHouse can also handle specification languages, not only programming languages[3]. We have also looked at the integration of UML into FeatureHouse[13].

The process of adding a language is light-weight. It may take an afternoon if the model is already present (e.g., for adding the functional language F# to our present collection, which contains already the functional language Haskell). Essentially one has to make the language's grammar available to FeatureHouse. The grammar of a language contains almost all information on how software artifacts written in that language can be composed. The only missing information is which syntactic elements of the language represent the modular structure (e.g., packages, namespaces, classes, interfaces, and methods) and which represent the terms of the language (e.g., method bodies and field initializations). The developer provides this information in the form of annotations added to the language's grammar. Based on an annotated grammar, FeatureHouse is able to compose artifacts of the corresponding language. FeatureHouse has been used to compose systems in a number of application domains including database systems, compilers, file systems, and network clients[2)].

Our interest in the exploration of the similarity of systems makes us want to navigate between different members of a product line and ensure that all members are correct in the strongest sense possible. To this end, we have made a first step by being able to guarantee the type safety of all members of a product line in a Java-like language, without checking the members individually[14]. In the future, we plan to extend existing work on language-independent type systems for feature-oriented programs to feature-oriented product lines[15].

We have also been working on tools to assess the quality of features. The static checker which we have been developing, called FeatureTweezer[16], addresses the first two questions raised in Sect. 2.6. It processes dependences and incompatibilities between structural elements of any language to create a feature model, which reflects the feature references and mutual exclusions of features. The generated model contains precisely all combinations of features that a compiler will accept. A feature model supplied by the user (which captures the intended variability and domain knowledge) can be compared to the generated one, indicating both combinations that break feature references, but are allowed in the given model, and combinations not allowed by the given model that may be desirable to increase product line variability.

Furthermore, we have developed a feature-oriented specification language, called FeatureAlloy, that enables us to detect feature interactions at a semantic level[3], which cannot be detected by our static and structure-driven checker FeatureTweezer. That is, by specifying pre- and postconditions, we can detect situations in which features violate the expectations of other features, which leads often to undesired feature interactions.

To address the third and fourth issue listed in Sect. 2.6, there is the concept of a derivative[17]. If two features interfere, their derivative is an additional feature that captures the functionality necessary to make them interact correctly. Our aim is to identify the need for derivatives and generate them automatically, but we need a behavioural specification to do this. In some cases, specifying the behaviour of the two interfering features suffices, in other cases, the behaviour of the derivative must

2) `http://www.fosd.de/fh/`

be specified as well. We are still at the beginning of our exploration of this issue. We expect the automatic generation of derivatives to be practical only in cases of tightly knit software product lines that require only simple derivatives.

5 An Example: A Switchable Software Store

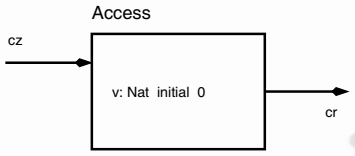
Let us take a very small example offered by Broy[6] to illustrate and compare the two approaches. Consider a simple software store which has only two features:

- Feature Access provides the functionality of the store: reading and updating natural numbers.
- Feature Switch has the functionality of turning the store on or off. In off mode, input to the store is ignored.

5.1 The software store as stream-processing functions

Broy specifies the two features as stream-processing functions. Let us do so first in isolation. We depict the features and also give their semantics as transition tables. A column of a transition table lists the values on a fixed input or output channel or the initial values, or the final values, of a fixed attribute. (For final values, the attribute's name is primed). A dash as table entry denotes the lack of value on a channel, a question mark denotes a wild card (any value).

- Feature Access (Fig.1): A local variable v –Broy calls it an attribute– holds a natural number. Input channel cz carries control signals, read and set(n), which tell to read out the buffered value or update it with parameter value n. Output channel cr carries a mixture of naturals that are read out and the control signal done that acknowledges an update.

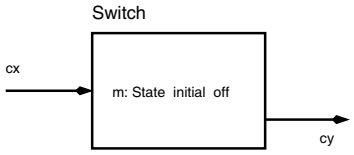


Access

v	cz	v′	cr
j	read	j	j
j	set(n)	n	done
j	–	j	–

Figure 1. Feature Access in isolation

- Feature Switch (Fig.2): Its attribute m holds a control signal telling the position of the switch: on or off. Input channel cx carries a stream of the control signal flip; every occurrence of the signal tells the component to flip the state m. If there is no signal, the value s of attribute m remains. Output channel cy carries a stream of switch positions (on or off).

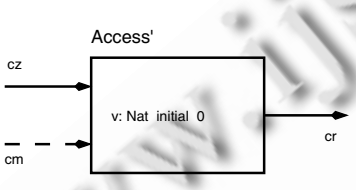


Switch

m	cx	m′	cy
off	flip	on	on
on	flip	off	off
s	–	s	–

Figure 2. Feature Switch in isolation

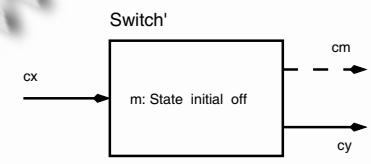
To make the store switchable, one must compose the two features and give feature Access a way to react to the output of feature Switch. Broy does so by introducing a new channel cm, which carries switch positions. He refines features Access to Access' by adding cm as input channel, and feature Switch to Switch' by adding cm as output channel. Being a mode channel, cm is depicted by a dashed arrow; see Fig.3 and Fig.4.



Access′

v	cm	cz	v′	cr
j	on	read	j	j
j	on	set(n)	n	done
j	on	–	j	–
j	off	?	j	–

Figure 3. Feature Access' with mode channel



Switch′

m	cx	m′	cm	cy
off	flip	on	on	on
on	flip	off	off	off
s	–	s	s	–

Figure 4. Feature Switch' with mode channel

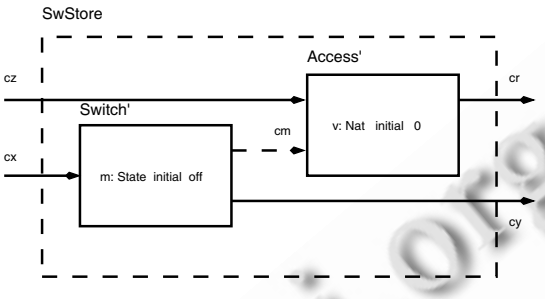
The extended features are combined by using cm as a common internal channel, not visible to the outside. The semantics of the combined features is derived by putting the two transition tables together with a non-standard join-like operation explained in detail by Broy. The result is depicted in Fig.5.

Switch′ • Access′

m	cx	m′	cm	cy	v	cm	cz	v′	cr
off	–	off	off	–	j	off	?	j	–
off	flip	on	on	on	j	on	–	j	–
off	flip	on	on	on	j	on	read	j	j
off	flip	on	on	on	j	on	set(n)	n	done
on	–	on	on	–	j	on	read	j	j
on	–	on	on	–	j	on	set(n)	n	done
on	flip	off	off	off	j	off	?	j	–
on	–	on	on	–	j	on	–	j	–

Figure 5. Combination of the transition tables of features Switch' and Access'

If we omit the columns for internal channels, in this case for cm, and sort the remaining columns by input (to the left) and output (to the right), we arrive at the description of the software store; see Fig.6.



SwStore

v	m	cz	cx	v'	m'	cr	cy
j	off	?	–	j	off	–	–
j	off	–	flip	j	on	–	on
j	off	read	flip	j	on	j	on
j	off	set(n)	flip	n	on	done	on
j	on	read	–	j	on	j	–
j	on	set(n)	–	n	on	done	–
j	on	?	flip	j	off	–	off
j	on	–	–	j	on	–	–

Figure 6. SwStore as the composition of features Switch' and Access'

5.2 The software store as feature structure trees

Next, we model the features of the software store as feature structure trees in the syntax of Java. Our objective is to mimick Broy's specification, but some differences arise due to the model. Also, working at the level of Java code, we show some implementational detail that Broy leaves unaddressed. In the following, consider Fig.7 top to bottom. The left side shows the Java code, the right side the corresponding feature structure trees.

Our first feature, Access, introduces the class SwStore, which models the software store. Feature Access gives it the capability of reading and updating a buffer variable v. Thus, we adopt Broy's attribute but, in the Java programming model, control signals are unnatural. We exert control by method call. The control signals read and set(n) become methods of class SwStore. The control signal done is represented by the return from method set. While Broy communicates the value of the attribute on channel cr, we let our method read return it.

Our next feature, Switch, extends class SwStore by adding a new enumeration type State, a field m for storing the switch's state, and a method that flips the state. Broy communicates the flip of the switch by outputting a control signal on channel cy which tells the switch's new position (Fig.2); as long as there is no flip, cy carries no signal. We have no need for such a communication and do not model it.

In isolation, feature Access is unaware of feature Switch, just as in Broy's approach. To make it aware, we must add a derivative feature that provides the connection. Look at the bottom of Fig.7. The derivative, named SwitchAccess, extends class SwStore by overriding the two methods read and set. Keyword original calls the overridden method and is like Java's super.

In Broy's model, a failed read generally goes unnoticed (unless the switch has just been flipped, in which case channel cy carries an off signal). We cannot let a failed read go unnoticed because method read must return a value. We return an error code of –1, which the caller may or may not pay attention to.

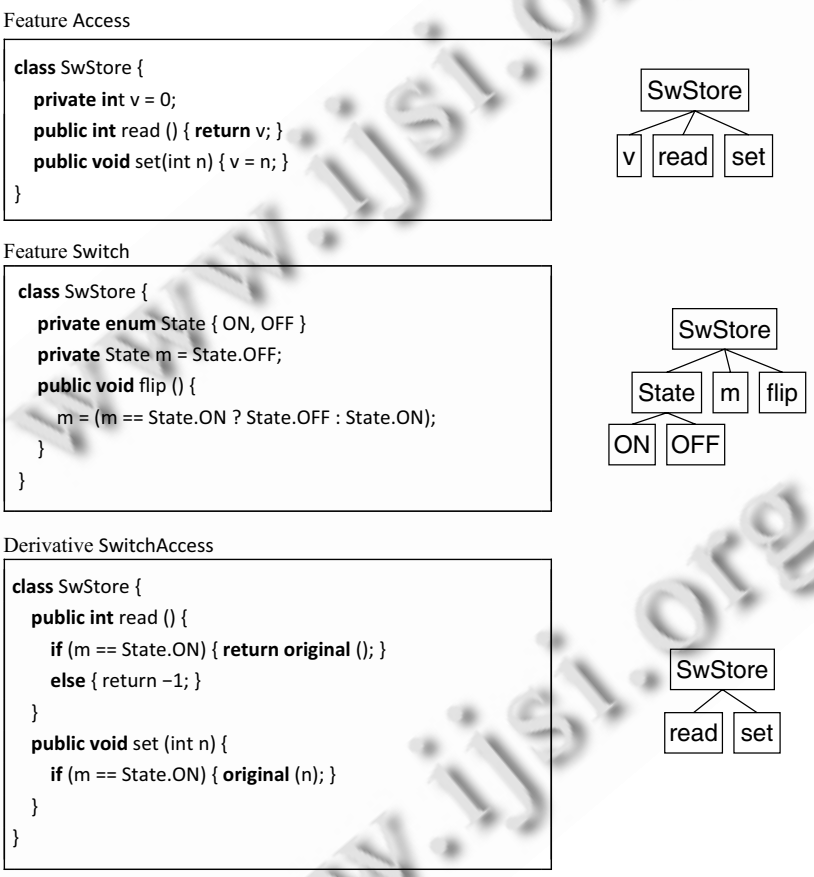


Figure 7. Feature structure trees and Java code of features Access, Switch, and their derivative

We compose two features by superimposing their feature structure trees. In the resulting tree, nodes that appear in both operand trees are identified and nodes that appear in just one of the two are adopted as are[18]. Conceptually, superimposition is a form of graph amalgamation that is applied to the roots of two trees and that proceeds recursively toward their leaves[19]. The switchable store is derived by superimposing the trees of the three features Access, Switch and SwitchAccess. Consider Fig.8.

Access•Switch•SwitchAccess

```
1  class SwStore {
2     private int v = 0;
3     public int read () {
4        if (m == State.ON) { return v; }
5        else { return −1; }
6     }
7     public void set (int n) {
8        if (m == State.ON) { v = n; }
9     }
10    private enum State { ON, OFF }
11    private State m = State.OFF;
12    public void flip () {
13       m = (m == State.ON ? State.OFF : State.ON);
14    }
15 }
```

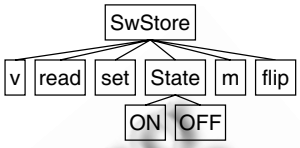


Figure 8. The superimposition of the three feature structure trees of the software store

6 A Comparison

There are correspondences between Broy's and our concept of a feature and of feature composition, but there are also major conceptual differences.

6.1 Features

Broy specifies a feature by stating syntactic and semantic conditions of its interface. He does not require an implementation; if he provides one, it is a finite automaton. We specify a feature by text in a programming or specification language. For executable languages, this corresponds to an implementation, which need not be a finite automaton.

6.2 Composition

Broy's composition operation is hierarchical. Two stream-processing functions, each with its interface, become one function with the two interfaces combined. Semantically, their transition tables are merged. The composition operator is in harmony with the modular structure of the system: it is the decomposition which imposes this structure. Crosscutting occurs, e.g., in the form of added mode channels which extend and connect several features.

Our composition operation is superimposing. It merges the identical structures of two feature structure trees and adds the structures unique to each of the two trees to the result tree. This affects the modular structure of the system, i.e., is crosscutting. Superimposition allows us to decompose a system along multiple dimensions, not only along a single dominant decomposition as in the case of Broy hierarchical decomposition into services and subservices.

6.3 Interaction

Controlling the interaction between components is a multi-faceted and still insufficiently well understood problem. In Broy's approach, interaction is controlled via the specification of channel communication. Unexpected communication may be

captured via additional mode channels.

Our counterpart to Broy's mode is the derivative. If something else but the two features is needed to make the composition work correctly, we condense it in one or more derivatives, i.e., additional features that take part in the superimposition. At present, we have to do this manually – just like Broy has to add his modes manually.

6.4 Algebra

The algebraic properties of composition are important to both Broy[6] and us[1]. However, we put a different emphasis on this aspect because we are more interested in exploiting the similarity of systems. Both Broy and our composition operator are associative but not commutative. But, other than Broy, we leave the option of giving up the associativity of feature composition when involving more expressive composition mechanisms such as advice weaving. Broy works at levels of abstraction at which the associativity of composition is not in question.

7 Conclusions

Broy and we address similar problems but offer different solutions. The main difference lies in the nature of the model and the composition operator. A language of stream-processing functions could be integrated into FeatureHouse. However, if nothing else is done but to teach FeatureHouse the grammar, it will apply superimposing composition. If one requires hierarchical composition, one should use AutoFOCUS rather than FeatureHouse. Ultimately, an integration of the two approaches would combine their strengths.

In any feature-oriented approach, including Broy's and ours, the main future challenge lies in the treatment of feature interaction. How is it to be specified? In what situations can it be detected automatically? Can it be repaired automatically? Does the (de)composition operator foster or hinder feature interaction? Researchers are only beginning to chart the many facets and challenges of this phenomenon.

Acknowledgements

We are grateful to Bernhard Möller, Wolfgang Scholz and Tobias Schüle for readings and discussions. We thank the anonymous referee for constructive comments. Ingolf Krüger and Bernhard Rumpe gave us helpful comments on Focus at the Dagstuhl Seminar 11021 on Feature-Oriented Software Development. We also had an informative exchange on AutoFOCUS with Bernhard Schätz.

The first author expresses heart-felt thanks to Manfred Broy (and also to Martin Wirsing) for 25 years of cheerful and supportive friendship.

References

[1] Apel S, Lengauer C, Möller B, Kästner C. An algebraic foundation for automatic feature-based program synthesis. Science of Computer Programming, 2010, 75(11): 1022–1047.

[2] Apel S, Kästner C. An overview of feature-oriented software development. J. Object Technology, 2009, 8(5): 49–84.

[3] Apel S, Scholz W, Lengauer C, Kästner C. Detecting dependences and interactions in feature-oriented design. Proc. IEEE Int. Symp. on Software Reliability Engineering (ISSRE). IEEE Computer Society, 2010. 161–170.

[4] Broy M, Stølen K. Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces, and Refinement. Monographs in Computer Science, Springer-Verlag, 2001.
[5] Ringert JO, Rumpe B. A little synopsis on streams, stream processing functions, and state-based stream processing. Int. J. Software Informatics, 2011. In this special issue.
[6] Broy M. Multifunctional software systems: Structured modeling and specification of functional requirements. Science of Computer Programming, 2010, 75(12): 1193–1214.
[7] Broy M, Krüger IH, Meisinger M, eds. Model-Driven Development of Reliable Automotive Services. LNCS 4992, Springer-Verlag, 2008.
[8] Botaschanjan J, Broy M, Gruler A, Harhurin A, Knapp S, Kof L, Paul W, Spichkova M. On the correctness of upper layers of automotive systems. Formal Aspects of Computing, 2008, 20(6): 637–662.
[9] Broy M, Fox J, Hölzl F, Koss D, Kuhrmann M, Meisinger M, Penzenstadler B, Rittmann S, Schätz B, Spichkova M, Wild D. Modeling CoCoME with Focus/AutoFocus. In Rausch A, Reussner R, Mirandola R, Plasil F, eds. The Common Component Modeling Example. Comparing Software Component Models. LNCS 5153, Springer-Verlag. 2007. 177–206.
[10] Spichkova M. Specification and Seamless Verification of Embedded Real-Time Systems: FOCUS on Isabelle. VDM Verlag Dr. Müller, 2008.
[11] Apel S, Kästner C, Lengauer C. FeatureHouse: language independent, automated software composition. Proc. of the ACM/IEEE Int. Conf. on Software Engineering (ICSE), IEEE Computer Society. 2009. 221–231.
[12] Jackson D. Alloy: A lightweight object modelling notation. ACM Trans. on Software Engineering and Methodology (TOSEM), 2002, 11(2): 259–290.
[13] Apel S, Janda F, Trujillo S, Kästner C. Model superimposition in software product lines. Proc. of the Int. Conf. on Model Transformation (ICMT). LNCS 5563, Springer-Verlag, 2009. 4–19.
[14] Apel S, Kästner C, Größlinger A, Lengauer C. Type safety for feature-oriented product lines. Automated Software Engineering, 2010, 17(3): 251–300.
[15] Apel S, Hutchins D. A calculus for uniform feature composition. ACM Trans. on Programming Languages and Systems (TOPLAS), 2010, 32(5): Article 19.
[16] Apel S, Scholz W, Lengauer C, Kästner C. Language-independent reference checking in software product lines. Second Int. Workshop on Feature-Oriented Software Development (FOSD). ACM Press, 2010. 65–71.
[17] Liu J, Batory D, Lengauer C. Feature oriented refactoring of legacy applications. Proc. of the ACM/IEEE Int. Conf. on Software Engineering (ICSE). ACM Press, 2006. 112–121.
[18] Apel S, Lengauer C. Superimposition: A language-independent approach to software composition. Proc. of the ETAPS Int. Symp. on Software Composition (SC). LNCS 4954, Springer-Verlag. 2008. 20–35.
[19] Böcker S, Bryant D, Dress A, Steel M. Algorithmic aspects of tree amalgamation. Journal of Algorithms, 2000, 37(2): 522–537.