A Survey on Software Feature Decompositions basing on Product Line Models

Jianfeng Chen
Department of Computer Science
North Carolina State University
Raleigh, NC
jchen37@ncsu.edu

ABSTRACT

In this paper, we review the development of feature decomposition techniques basing on the eight papers from 2009 to 2013. We introduce the basic terms in software product lines and feature checking models; as well as how the algorithms develop in recent years. We also do a comparison between some of them and raise some suggestions in this area.

Keywords

Feature model; Software Product Lines; Software Decomposition; Paper Review

1. INTRODUCTION

This paper is the survey for the development of feature decomposition techniques basing on the eight papers from 2009 to 2013. We introduce the basic terms in software product lines and feature checking models; as well as how the algorithms develop in recent years. We also do a comparison between some of them and raise some suggestions in this area.

To do this survey, we first select one highly cited paper in ASE11 conference, and then found some of this referencing paper as the history; as well as some of its cited paper as the process in the later years. Finally, we had seven papers as the material for our survey.

This paper organizes as follows: section 2 introduces many important keywords in this research area, many of them are mentioned in more than one paper; section 3 introduces the development process and research hot-spot from 2008 to 2013 in the area of software decomposition; section 4 introduces several typical models in the related area, including their motivations, advantages and shortcomings; section 5 introduces the results from the models in section 6, from which we can see the technical improvement from the past 10 years; section 7 is the conclusions and the future; section 6 is the threats to validity in our survey.

2. KEYWORDS

DOI: 10.475/123_4

In software development, a *feature* is a unit of functionality that satisfies a requirement, represents a design decision, and provides a potential configuration option[1]. It is the stakeholder-visible behavior or characteristic of a program. For example, in a web portal system, NTTP, FTP or HTTPS are the features in the view of protocols.

A *software product line* is a family of software products that share a common set of features and differ in others[2]. Typically, from a set of features, many different software systems (a.k.a. variants) can be generated that share common features and differ in other features. The complete set of variants is also called a software product line[3].

Product Line Engineering is a development paradigm that explicitly addresses reuse by differentiating between two kinds of development processes[4]: In domain engineering, the domain artifacts of the product line are defined and developed. In application engineering, customer- and/or market-specific products are derived from the domain artifacts by binding the variability defined in the domain artifacts according to customer and/or market specific needs. The overall quality of the product line and its derived products mainly depends on the quality of the domain artifacts. In contrast to the development artifacts created in single systems engineering, the domain artifacts created in product line engineering are reused in several products derived from the product line. Thus, a high quality of the domain artifacts is desirable. A defect in a domain artifact typically affects several products of the product line and is thus costly to remove.

A *feature interaction* is a situation[5][6] in which the composition of multiple features leads to error or other situations. For example, a telephone line offers two features: call forwarding and call waiting. If one call comes during another call is on, the system has to decision which feature/function should be applied–it can forward the call to secondary person or let the first coming call wait.

Feature Structure Tree (FST) model[7] [8]are designed to represent any kind of artifact with a hierarchical structure. An FST represents the essential modular structure of a software artifact and abstracts from language-specific details. For example, an artifact written in Java contains packages, classes, methods, etc., which are represented by nodes in the FST. An XML document (e.g., XHTML) may contain elements that represent the underlying document structure, e.g., headers, sections, paragraphs. A makefile or build script consists of definitions and rules that may be nested. See the figure 1.

```java
package com.sleepycat;
public class Database {
  private DbState state;
  private List triggerList;
  protected void notifyTriggers(Locker locker, DatabaseEntry priKey,
    DatabaseEntry oldData, DatabaseEntry newData) throws DatabaseException {
      for (int i=0; i<triggerList.size(); i+=1) {
        DatabaseTrigger trigger = (DatabaseTrigger)triggerList.get(i);
        trigger.databaseUpdated(this, locker, priKey, oldData, newData);
      }
  } // over 650 further lines of code...
}
```

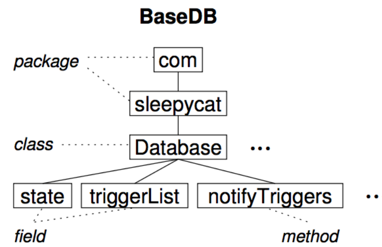


Figure 1: An example of feature structure tree model

3. RELATED WORK

There is a large body of research on automated detection of feature interactions and automated softweare decompositions [9] [10]. Many approaches aim at detecting feature interactions at the specification level. For example, [11][12] use a pair-wise measurement approach based on linear temporal logic to detect feature interaction. They specify the behavior of a product line in Promela (a modeling language). Using a model checker, they generate for each pair-wise combination a model checking run to verify whether the defined properties are still valid. Other approaches use state charts to model and detect feature interactions[13]. [14] feature specifications are translated to a reach-ability graph. The authors use state transitions to detect whether a certain state is not exclusively reachable in isolation (i.e. a feature interaction occurs). There are approaches that provide means to detect semantic feature interactions, i.e., feature interactions that change the functional behavior of a program. Some use model checking techniques to find semantic feature interactions[15][16]. Apel's work uses model-checking techniques to verify whether semantic constraints still hold in a particular feature combination [17][18]. Other approaches aim at investigating the code base to detect structural feature interactions. For example, Liu et al.[19] propose to model feature interactions explicitly using algebraic theory. In contrast to these approaches, they focus on performance feature interactions in a black-box fashion.

Software composition[20] is related to the broad field of software merging. Software merging attempts to merge different versions of a software system not only at the module level but at all levels of granularity by using syntactic, semantic, and evolutionary information [21]. Especially for the implementation of artifact-specific composition rules, superimposition can benefit from these developments. In a parallel line of research, we have implemented a product line tool, called CIDE, that allows a developer to decompose a legacy software system into a product line, to type-check all products of a product line, and to visualize and resolve feature interactions[22]. CIDE pursues also a generative approach of integrating new languages [23] based on the same grammar format we use in FEATUREHOUSE but on different attributes; initially, FEATUREBNF has been developed for CIDE. It uses the entire parse tree, thus, it does not require a mapping to terminals and nonterminals of an FST. The coordinated development of FEATUREHOUSE and CIDE allows us to use grammars in both projects. CIDE has been used to refactor Berkeley DB, one of our case studies.

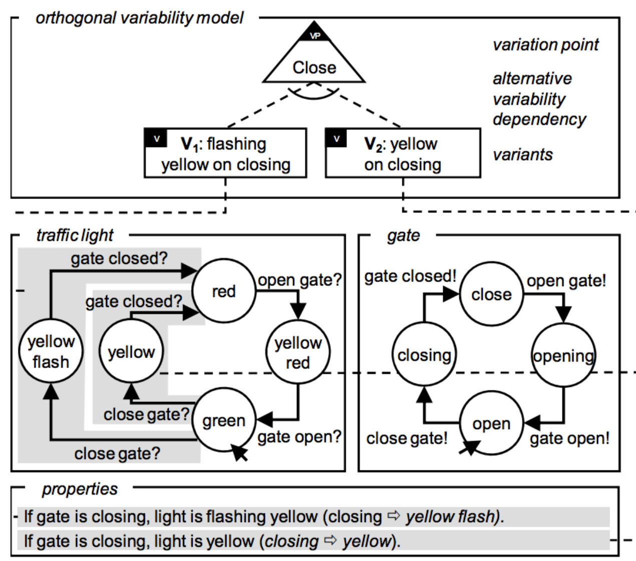


Figure 2: Simplified Example of Domain Artifacts

4. TYPICAL MODELS

In this section, we will introduce several typical models in the area of software decomposition.

4.1 CTL Model Checking[4]

4.1.1 Motivation

The approach presented in CTL model checking supports model-checking of properties specified in CTL and thus, compared with the previous work which just focused on invariants, supports the model-checking of much richer property specifications (Fig.2).

4.1.2 Algorithm

The central idea of the CTL approach is to include the variability information specified in the variability model (as Boolean variables) into the model checking algorithms. During the exploration of the state space, the algorithms consider the variability model to ensure that the current path explored in the state space is valid with respect to the variability model.

The adaptation is threefold:

1) Adaptation of state labeling: In variable IO - automata, the fulfillment of a property may rely on variable transitions. Therefore, the state labeling may include the variant selection which is necessary to fulfill the property.

2) Adaptation of algorithms: We adapt the algorithms for model checking of EX, EU, and EG. This is sufficient since all other expression can be reduced to a combination of the EX, EU and EG operators. It is not necessary to adapt the procedures for handling expressions of the form f_1 and $f_1 f_2$ because the results of the computations only depend on single states.

3) Checking the completeness of witnesses: The existing single system algorithms rely on witnesses to show that a property is fulfilled for a given system. This approach is not sufficient for variable IO - automata, since a variable I/O-automaton represents a set of systems and thus a witness must exist for every possible system.

4.2 A feature algebra[24]

4.2.1 Motivation

The feature algebra can abstract from the details of different programming languages and environments used in FOSD(*Feature-Oriented Software Development*). Second, alternative design decisions in the algebra reflect variants and alternatives in concrete programming language mechanisms; for example, certain kinds of feature composition may be allowed or disallowed. Third, the algebra is useful for describing, beside composition, also other operations on features formally and independently of the language, e.g., type checking and interaction analysis. Fourth, the algebraic description can be taken as an architectural view of a software system. External tools can use the algebra as a basis for optimizing feature expressions.

4.2.2 Implement example

Implementation and FST of the feature `BASE`–see Fig3

FST superimposition (`ADD • BASE = ADDBASE`)–see Fig 4

Superimposing Java methods in `COUNT • BASE = COUNT-BASE`– see Fig 5

Superimposing Java methods by inlining–see Fig 6

4.3 Detecting Dependences and Interactions in Feature-Oriented Design

4.3.1 Overview

In [17], they propose a novel design paradigm, called feature-oriented design, that is tailored to the needs of FOSD. The idea is to take advantage of the clean mapping of features and their implementations and to concentrate on designing the structure and behavior of features as well as their dependences and interactions. [17] base our proposal of feature-oriented design on the lightweight but expressive modeling language Alloy [25]. [17] favor Alloy over other modeling languages, such as the unified modeling language (UML), because of its support of automatic reasoning. Alloy's automatic reasoning facilities are useful for detecting semantic dependences and interactions between features, which cause major problems in complex software systems[10] and are still

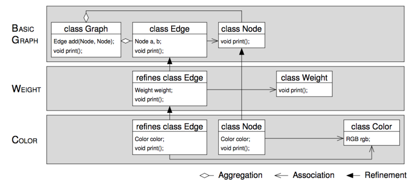


Figure 7: Collaboration-based design of a simple graph structure

a challenge for research and industry

4.3.2 ALLOY

Alloy is a lightweight, textual modeling language for software design. It is based on relations and logic, but has an object-oriented look and feel. This may be one reason for its acceptance in academia and industry. Its simplicity and sound mathematical foundation allow tools such as the Alloy Analyzer[1] to reason about Alloy models automatically (e.g., to decide whether there are legal instances of a model or whether certain properties hold in a model). See fig 78 as an example.

4.3.3 FeatureAlloy

FeatureAlloy extends Alloy by three ingredients useful for FOSD: (1) collaboration-based design, (2) stepwise refinement, and (3) feature composition. In a nutshell, FeatureAlloy follows the philosophy of contemporary FOSD languages and tools. It represents each feature as a containment hierarchy, which encapsulates a collaboration of model elements (signatures, facts, etc.) that belong to a feature. Furthermore, FeatureAlloy supports the refinement of existing Alloy modules, signatures, facts, and so on by subsequent features without the need to modify existing model elements. Note that, like in the seminal work on feature interactions in telecommunication systems, a feature is not necessarily decoratively complete and a developer may have to combine it with a base program or with other features. That is, features are often increments in program functionality. Finally, features are composed based on an external and declarative user'featured transitions systemss specification. A generator superimposes the model elements of the features involved and produces the final model of the system.

Furthermore, they add a feature **UNIQUEVALUES**, which assigns unique values to the nodes of a graph. In Figure 9, they show a corresponding refinement that refines signature Node by adding a new field val and that adds a fact defining that node values are unique. Notice the similarity of module and signature refinement. If there is already a field with the same name, the new definition of the signature refinement overrides the existing definition.

4.4 Featured Transitions Systems and Feature LTL[11]

4.4.1 Motivational Statements

Building variability-intensive systems has inherent advantages. But the complexity created by the variability leads to the problems. In the current state of SPLE, most analysis are thus carried out when building a product; only a few

[1] http://alloy.mit.edu/

```java
package util;
class Calc {
  int e0=0, e1=0, e2=0;
  void enter(int v) { e2=e1; e1=e0; e0=v; }
  void clear() { e0=e1=e2=0; }
  String top() { return String.valueOf(e0); }
}
```

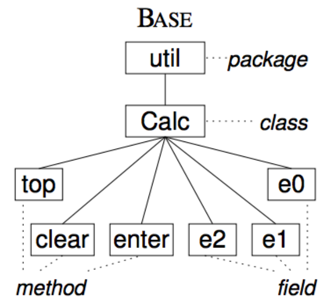


Figure 3: Implementation and FST of the feature `BASE`

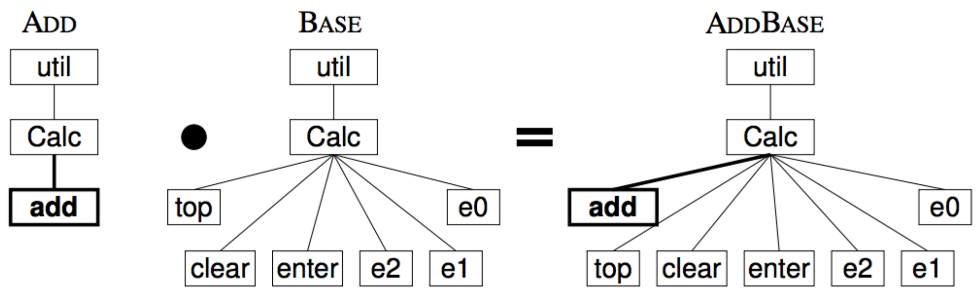


Figure 4: An example of FST superimposition (`ADD • BASE = ADDBASE`)

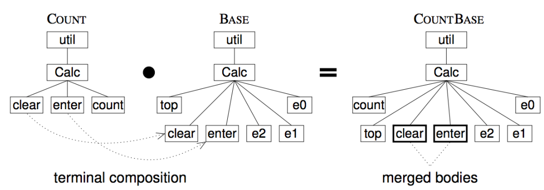


Figure 5: Superimposing Java methods in `COUNT • BASE = COUNTBASE`

```java
package util;
class Calc {
  int count=0;
  void enter(int val) { original(val); count++; }
  void clear() { original(); if(count > 0) count--; }
}
```

•

```java
package util;
class Calc {
  int e0=0, e1=0, e2=0;
  void enter(int val) { e2=e1; e1=e0; e0=val; }
  void clear() { e0=e1=e2=0; }
  String top() { return String.valueOf(e0); }
}
```

=

```java
package util;
class Calc {
  int e0=0, e1=0, e2=0;
  void enter(int val) { e2=e1; e1=e0; e0=val; count++; }
  void clear() { e0=e1=e2=0; if(count > 0) count--; }
  String top() { return String.valueOf(e0); }
  int count=0;
}
```

Figure 6: Superimposing Java methods by inlining

Feature BASICGRAPH

```
module Graph
// a singleton graph contains multiple nodes
one sig Graph {
  nodes: set Node
} {
  Node in nodes
}
// each node has multiple incoming and outgoing edges
sig Node {
  inEdges: set Edge, outEdges: set Edge, edges: set Edge
} {
  edges = inEdges + outEdges
}
// each edge has a source and destination node
sig Edge {
  src: one Node, dest: one Node
}
// defines proper connections between nodes and edges
fact prevNext {
  all n: Node, e: Edge |
    (n in e.src <=> e in n.outEdges) && (n in e.dest <=> e in n.inEdges)
}
// determines the number of reachable nodes (incl. the given node)
fun reachableNodes [n: Node] : Int {
  #(n.^(edges.(src + dest)))
}
// property that a graph has no double edges
pred noDoubleEdges {
  all e, e': Edge | e != e' => e.(src + dest) != e'.(src + dest)
}
// creates an instance without double edges
run noDoubleEdges for 5
// holds if the graph has no double edges
assert hasNoDoubleEdges {
  !noDoubleEdges
}
// checks whether the graph has no double edges
check hasNoDoubleEdges for 5
```

Figure 8: An Alloy model of a simple graph

Feature UNIQUEVALUES

```
refines module Graph
// adds a value to each node
refines sig Node {
  val: one Int
}
// defines that node values are unique
fact uniqueValues {
  all disj n, n': Node | n.val != n'.val
}
```

Figure 9: A refinement that assigns unqiue values to nodes.

are conducted during domain engineering, that is, build the assets from which products are derived.

In this model, they study model checking algorithms for featured transition systems (FTSs) [15], a concise mathematical model for representing the behavior of a large number of products. The proposed FTS algorithms are semisymbolic, combining an automata-based model checking algorithm [26] with a symbolic encoding of sets of products. Their main contribution is an in-depth treatment of these algorithms. We study their properties, optimizations, and a symbolic encoding for sets of products; their provide correctness proofs and an analysis of their computational complexity. Among the minor contributions, we elaborate on concepts such as expressiveness of FTSs, parallel composition, deadlock checking, vacuity detection, and introduce a logic called feature LTL (fLTL).

4.4.2 *Important terms in the fLTL*

Fig 10, 11, 12 are the definition for LTL property, feature

Definition 11. *An LTL property ϕ is an expression*

$$\phi ::= 1 \mid a\ (\in AP) \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \bigcirc \phi \mid \phi_1 U \phi_2.$$

A property is interpreted over infinite executions. Satisfaction for an execution π is defined as follows:

$$\begin{array}{lll}
\pi \models 1 & & \\
\pi \models a & \iff & a \in L(head(\pi)) \\
\pi \models \phi_1 \wedge \phi_2 & \iff & \pi \models \phi_1 \text{ and } \pi \models \phi_2 \\
\pi \models \neg\phi & \iff & \pi \not\models \phi \\
\pi \models \bigcirc\phi & \iff & \pi_1 \models \phi \\
\pi \models \phi_1 U \phi_2 & \iff & \exists i \geq 0 \bullet \pi_i \models \phi_2 \text{ and } \\
& & \quad \forall j \in [0, i-1] \bullet \pi_j \models \phi_1,
\end{array}$$

where $head(\pi)$ denotes the first state in π, and π_i the tail of π starting at the $i-1$th state (with $\pi_0 = \pi$). A TS satisfies a property $ts \models \phi$ iff all its executions satisfy ϕ.

Figure 10: Definition for LTL property

expression and important function $ExtMc\{fts, \phi\}$ respectively, all of which are the foundation of fLTL model. From these three terms, we can find the simpleness and meticulous parts in fLTL.

The author in [11] gave us a very detailed and reachable algorithm for the fLTL model. We won't list more details here. One can refer this paper for more information.

5. IMPROVED RESULTS

This section will list some significant results in this research area. Since many of papers are focusing one different perspectives, although all of them are decomposing the software features, we can't compare the these results from the beginning to the end. All we should do is to compare the

Definition 12. *An fLTL property ϕ is an expression $\phi := [\chi]\phi'$ where ϕ' is an LTL property and $\chi \in \mathbb{B}$ is a feature expression. An FTS satisfies ϕ, noted $fts \models \phi$, iff*

$$\forall p \in [\![\chi]\!] \cap [\![d]\!] \bullet fts_{|p} \models \phi'.$$

That is, each product of the FD that is included in the quantification yields a TS that satisfies the LTL property.

Figure 11: Definition for LTL feature expression

Definition 15. *Given a property ϕ and an FTS fts, $ExtMc(fts, \phi)$ returns* true *iff $fts \models \phi$. If $fts \not\models \phi$, it returns* false *and a set c of couples (e, px) where px is a nonempty set of products such that $\forall p \in px \bullet fts_{|p} \not\models \phi$ with e as a counterexample. Furthermore, it holds that*

$$\forall p \in [\![d]\!]_{FD} \bullet p \notin \bigcup_{(e,px)\in c} px \implies fts_{|p} \models \phi.$$

To simplify the notation, we write $px_{\not\models\phi}$ (respectively, $px_{\models\phi}$) to denote the set of products that violate (respectively, satisfy) ϕ.

Figure 12: Definition for function $ExtMc\{fts, \phi\}$

models doing the same job, such as $ALLOY$ vs. *FeatureAlloy*, etc.

5.1 CTL Model Checking[4]

The runtime estimation of our presented approach indicates an exponential worst case runtime for the verification of EU and EG properties. In order to determine the runtime behavior of our approach, they have realized the approach in a prototypical tool environment in order to apply it to examples. They applied their approach to two examples and verified for each example one property of each type (i.e. EX, EU, and EG). The first example is a small sample specification. It consists of five variable I/O-automata and an orthogonal variability model which specifies six variation points and 14 variants. Overall 189 products can be derived from this specification. The product automaton of the specification consists of 12.000 states and 29.000 transitions. The second example is a (realistic) specification consists of six variable I/O-automata and the orthogonal variability model of the specification consists of ten variation points and 46 variants and allows the derivation of 237 different products. The product automaton of the specification consists of more than 68.000 states and 174.000 transitions. See Fig 13

From this initial runtime evaluation (Fig 13) we can conclude that: 1) In both examples, the product construction

Property	Runtime (sample specification) 12.000 states / 29.000 transitions		Runtime (realistic specification) 68.000 states / 174.000 transitions	
	Product-construction	Verification	Product-construction	Verification
EX	99,72sec	0,27sec	203,7sec	1,7sec
EU	100,08sec	0,25sec	202,8sec	0,75sec
EG	99,92sec	4,25sec	202,7sec	32,93sec

Figure 13: Runtime for EX, EU and EG in [4]

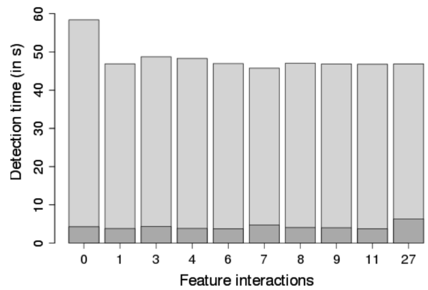


Figure 14: Times needed to prove that the individual interactions do not occur (brute force in light gray and variability encoding in dark gray).

requires a large amount computation time. This is not surprising, since the product construction suffers from the so called state explosion problem, i.e. the runtime of the product construction grows exponentially with the number of component automata.

2) For both examples, the verification of an EX property is fast compared with the overall runtime (0,27% for the first and 0,83% for the second example). This supports the results of the runtime evaluation which showed that the verification of EX requires linear runtime.

3) In both examples, the verification of an EU property is fast compared with the overall runtime for both examples (0,25% in the first and 0,37% in the second example).

4) For both examples, the verification of an EG property requires significantly more time (4% in the first and 14% in the second example) compared with the runtime required for verifying the other two properties (EX and EU)

5.2 Feature-aware verification[27]

They conducted a number of experiments with the e-mail product line. First, we generated all of its 40 products and checked them using both CBMC and CPAchecker. It turned out that with feature-aware verification, we were able to detect all feature interactions of Table I based on the feature-local specifications of the input features. If the model checker does not report a counterexample (i.e., none of the safety properties has been violated), we can be certain that the composition does not contain a feature interaction that violates the specification of the features involved.

To further explore their pros and cons, we compare the brute-force approach (i.e., checking all possible products) with the variability-encoding approach in terms of verification time. Our case study contains several unsafe feature interactions, so we made the comparison on a per-interaction basis. Specifically, we measured the runtime needed to find a feature interaction or to report that no feature interaction has been found. Because every specification is associated with a feature, both approaches need to consider only feature combinations that contain this feature; all other combinations trivially cannot violate the specification. See Fig 14, 15, 16 for detailed results.

5.3 Featured Transitions Systems and Feature LTL[11]

The entire set of results is given in Fig. 17.

A comparison to the naive algorithm implemented in SPIN

	Property		RUNTIME snip	snip -spin	spin + gcc	num (spin)	#STATES snip	snip -spin	spin + gcc	num (spin)	SPEEDUPS snip -spin	spin + gcc	vs spin
#1	Checking deadlocks and asserts.	sat	2.20	7.57	57.23	1.21	250 770	888 028	401 460	401 460	3.44	26.01	0.55
#2	!([]<> (stateReady && highWater && userStart))	64 good, 64 bad	1.48	4.96	59.1	0.88	151 783	490 526	180 190	180 190	3.35	39.93	0.59
#3	!([]<> stateReady)	64 good, 64 bad	1.29	4.38	59.07	0.95	138 719	445 440	151 136	151 136	3.40	45.79	0.74
#4	!([]<> stateRunning)	96 good, 32 bad	1.91	7.11	59.2	1.11	207 355	758 294	250 606	250 606	3.72	30.99	0.58
#5	!([]<> stateStopped)	unsat	3.34	6.27	58.85	0.6	185 017	373 516	1 664	1 664	1.88	17.62	0.18
#6	!([]<> stateMethanestop)	56 good, 72 bad	1.39	3.54	59.04	0.69	151 886	364 202	97 478	97 478	2.55	42.47	0.50
#7	!([]<> stateLowstop)	112 good, 16 bad	2.46	8.86	59.49	1.32	265 606	968 274	314 162	314 162	3.60	24.18	0.54
#8	!([]<> readCommand)	unsat	0.03	0.59	58.84	0.62	2 918	30 616	29 504	29 504	19.67	1961.33	20.67
#9	!([]<> readAlarm)	unsat	0.02	0.52	58.86	0.62	1 803	21 488	16 290	16 290	26.00	2943.00	31.00
#10	!([]<> readLevel)	unsat	0.06	0.75	58.79	0.6	7 315	48 230	3 072	3 072	12.50	979.83	10.00
#11	!(([]<> readCommand) && ([]<> readAlarm) && ([]<> readLev	unsat	0.36	2.24	60.24	0.65	45 414	188 484	88 684	88 684	6.22	167.33	1.81
#12	!([]<> pumpOn)	96 good, 32 bad	1.90	6.98	59.52	1.16	207 377	758 338	250 606	250 606	3.67	31.33	0.61
#13	!([]<> !pumpOn)	unsat	0.02	0.48	58.87	0.63	1 617	20 208	1 664	1 664	24.00	2943.50	31.50
#14	!(([]<> pumpOn) && ([]<> !pumpOn))	100 good, 28 bad	5.82	18.43	61.76	2.99	508 798	1 762 874	572 628	572 628	3.17	10.61	0.51
#15	!([]<> methane)	unsat	0.01	0.38	58.92	0.63	456	10 672	1 664	1 664	38.00	5892.00	63.00
#16	!([]<> !methane)	unsat	0.03	0.44	58.91	0.63	1 870	18 256	5 248	5 248	14.67	1963.67	21.00
#17	!(([]<> methane) && ([]<> !methane))	unsat	0.02	0.47	59.25	0.6	2 117	21 104	15 208	15 208	23.50	2962.50	30.00
#18	[] (!pumpOn || stateRunning)	sat	3.06	10.57	60.06	1.65	326 064	1 159 616	401 460	401 460	3.45	19.63	0.54
#19	[] (methane -> (<> stateMethanestop))	unsat	0.01	0.36	59.03	0.61	131	4 864	1 344	1 344	36.00	5903.00	61.00
#20	[] (methane -> !(<> stateMethanestop))	56 good, 72 bad	2.92	6.80	60.03	1.3	280 806	672 598	189 784	189 784	2.33	20.56	0.45
#21	[] (pumpOn || !methane)	unsat	0.01	0.29	58.92	0.61	41	2 432	768	768	29.00	5892.00	61.00
#22	[] ((pumpOn && methane) -> <> !pumpOn)	96 good, 32 bad	1.82	6.80	59.64	1.11	197 006	731 976	249 428	249 428	3.74	32.77	0.61
#23	(([]<> readCommand) && ([]<> readAlarm) && ([]<> readLev	112 good, 16 bad	3.09	10.07	63.14	1.75	316 182	1 070 784	401 490	401 490	3.26	20.43	0.57
#24	!<>[] (pumpOn && methane)	96 good, 32 bad	1.84	6.83	59.64	1.12	197 262	732 488	249 428	249 428	3.71	32.41	0.61
#25	(([]<> readCommand) && ([]<> readAlarm) && ([]<> readLev	112 good, 16 bad	3.00	9.97	63.42	1.8	308 812	1 056 044	399 186	399 186	3.32	21.14	0.60
#26	[] ((!pumpOn && methane && <>!methane) -> ((!pumpOn) U	112 good, 16 bad	5.28	16.17	61.93	2.68	509 577	1 682 562	626 044	626 044	3.06	11.73	0.51
#27	[] ((highWater && !methane) -> <>pumpOn)	unsat	0.05	0.63	59.05	0.57	4 430	38 384	8 288	8 288	12.60	1181.00	11.40
#28	!(<> (highWater && !methane))	unsat	0.03	0.59	58.73	0.6	3 746	35 584	6 432	6 432	19.67	1957.67	20.00
#29	(([]<> readCommand) && ([]<> readAlarm) && ([]<> readLev	unsat	1.35	3.44	78.14	1.29	140 051	326 762	124 864	124 864	2.55	57.88	0.96
#30	[] ((highWater && !methane) -> !<>pumpOn)	96 good, 32 bad	3.87	13.28	60.75	1.9	398 167	1 457 726	425 172	425 172	3.43	15.70	0.49
#31	!<>[] (!pumpOn && highWater)	unsat	0.03	0.71	59.2	0.61	3 696	34 876	3 136	3 136	23.67	1973.33	20.33
#32	(([]<> readCommand) && ([]<> readAlarm) && ([]<> readLev	unsat	1.20	2.73	62.33	0.75	105 401	196 000	41 292	41 292	2.28	51.94	0.63
#33	!<>[] (!pumpOn && !methane && highWater)	unsat	0.04	0.68	59.12	0.58	4 158	37 244	6 688	6 688	17.00	1478.00	14.50
#34	(([]<> readCommand) && ([]<> readAlarm) && ([]<> readLev	sat	4.76	15.76	64.38	2.77	441 063	1 551 742	662 792	662 792	3.31	13.53	0.58
#35	[] ((pumpOn && highWater && <>lowWater) -> (pumpOn U	104 good, 24 bad	2.35	8.15	61.36	1.31	246 288	870 136	288 400	288 400	3.47	26.11	0.56
#36	!<> (pumpOn && highWater && <>lowWater)	96 good, 32 bad	1.93	7.13	60.05	1.18	198 492	734 948	250 196	250 196	3.69	31.11	0.61
#37	[] (lowWater -> (<>!pumpOn))	96 good, 32 bad	2.02	7.40	59.59	1.15	218 552	805 600	242 986	242 986	3.66	29.50	0.57
#38	(([]<> readCommand) && ([]<> readAlarm) && ([]<> readLev	120 good, 8 bad	3.23	10.59	63.3	1.85	335 378	1 147 596	436 136	436 136	3.28	19.60	0.57
#39	!<>[] (pumpOn && lowWater)	96 good, 32 bad	2.04	7.55	59.5	1.14	218 540	805 576	242 978	242 978	3.70	29.17	0.56
#40	(([]<> readCommand) && ([]<> readAlarm) && ([]<> readLev	120 good, 8 bad	3.04	10.39	63.34	1.85	317 818	1 112 476	428 496	428 496	3.42	20.84	0.61
#41	[] ((!pumpOn && lowWater && <>highWater) -> ((!pumpOn)	sat	4.34	17.52	61.86	2.72	419 230	1 842 172	622 250	622 250	4.04	14.25	0.63
#42	!<> (!pumpOn && lowWater && <>highWater)	unsat	0.04	0.73	59.39	0.62	3 894	38 496	2 432	2 432	18.25	1484.75	15.50
#43	[MethaneAlarm]((([]<>readCommand) && ([]<>readAlarm) &	sat	2.65	6.87	32.09	1.37	276147	750042	297082	297082	2.59	12.11	0.52
#44	[MethaneAlarm]([] ((pumpOn && methane) -> <> !pumpOn)	48 good, 16 bad	1.40	4.34	30	0.79	151546	478256	173020	173020	3.10	21.43	0.56
#45	[MethaneAlarm]([] ((!pumpOn && methane && <>!methane	56 good, 8 bad	3.98	10.13	31.28	1.75	385231	1047486	413710	413710	2.55	7.86	0.44

Figure 17: The results of the benchmark experiments. The third column indicates how many products satisfied the property (sat means all and unsat means none). The colors in the last three columns visualize the speedup

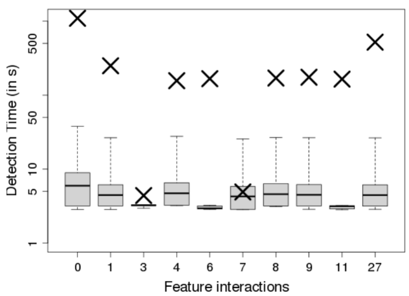


Figure 15: Times needed to find the individual interactions (brute force as box plots and variability encoding as crosses; y-axis in log scale)

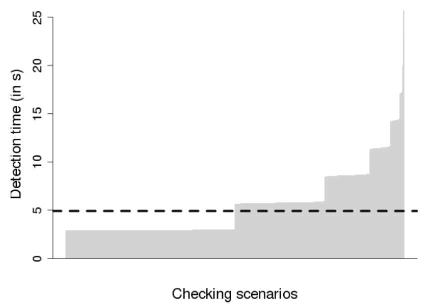
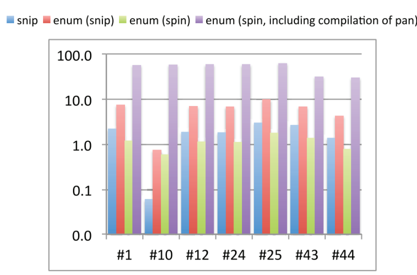
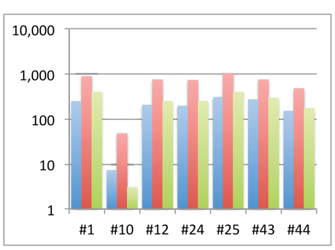


Figure 16: Times needed to find the unsafe feature interaction between Encrypt and Verify (brute force as bars and variability encoding as dashed line).



(a) Runtime (in seconds)



(b) State space (in thousands)

Figure 18: A subset of the results of Fig. 17

does not allow them to draw any conclusion as to the impact of the FTS algorithm. The results show that if the compilation time for SPIN's process analyzers is not counted, SPIN generally outperforms SNIP by a factor of two. Nevertheless, SNIP is generally faster on the properties violated by all products, for example, 10 times faster for (10). However, a fair comparison of SNIP and the script has to take all times into account. The compilation of a process analyzer takes about 60 seconds and accounts, on average, for 98 percent of the total runtime. SNIP thus clearly outperforms the script of the naive algorithm.

5.4 FeatureHouse[7]

In order to demonstrate the practicality of FEATUREHOUSE, [7] have composed software systems of different sizes written in different languages. Fig 19 is the summary for information on the software systems and their compositions. We highlight here only some interesting observations. The source code of all software systems of our study can be downloaded at the FEATUREHOUSE website.

From Fig 19, we can learn that:

1. Superimposition of FSTs scales to medium-sized software projects.

2. The time for annotating grammars is moderate and the depths of the generated FSTs depend on the composition granularity and the complexity of the language.

3. Artifacts written in languages whose structural elements may have identical names (whose elements are distinguished by the lexical order) have to prepared, in order to be superimposed. Basically, each FST node receives a unique name, as in GPL's XHTML documentation.

4. The order of an artifact's elements represented by FST nodes (terminals or non-terminals) may matter. If it matters, as in the case of #include, the technique of sandwiching can be used as a workaround.

5. Non-code or even unstructured artifacts, such as plain text files or binaries, can be integrated seamlessly with FSTCOMPOSER.

6. In practice, only a few composition rules are needed, which can be reused by different languages and which follow even fewer rule patterns.

7. Superimposition is applicable to a wide range of code and non-code languages including object-oriented languages, functional languages, imperative languages, document description languages, and grammar specification languages.

6. CONCLUSIONS AND THE FUTURE

Software feature decomposition is always a research hotspot since 2008. Many research tried to solve this problem in many ways. In this survey, they can see that some of these methods are based on the feature-model tree. They want to find a software product which can fulfill the constraints. On the other hand, some of them are trying to solve this problem by the algebra method. By the algebra method, they can decomposition the software by the logic operations. However, the most difficult part for algebra methods are how to express the real software, since the real existed software are so complex (think about various design pattern[28] and the method override in object-orient languages).

Although there exists many algorithms in this area, they can't solve this problem perfectly. For some algorithm, they can solve one kind of software or pattern, while for other patterns, they're not suitable. Consequently, we predict this

	COM	CLA	LOC	TYP	TIM	Description
FFJ	2	–	289	JavaCC	< 1 s	Feature Featherweight Java Grammar [3]
Arith	15	–	442	Haskell	< 1 s	Arithmetic expression evaluator
GraphLib	13	–	934	C	1 s	Low-level graph library[a]
GPL	26	57	2,439	Java, XML	2 s	Graph Product Line (Java Version) [26]
GPL	26	57	2,148	C#	2 s	Graph Product Line (C# Version)
Violet	88	157	9,660	Java, Text	7 s	UML Editor[b]
GUIDSL	26	294	13,457	Java	10 s	Configuration management tool [7]
Berkeley DB	99	765	58,030	Java	24 s	Oracle's Embedded Storage Engine [23]

COM: number of units of composition; CLA: number of classes; LOC: lines of code; TYP: types of contained artifacts; TIM: time to compose

[a] http://keithbriggs.info/graphlib.html [b] http://www.horstmann.com/violet/

Figure 19: Overview of the case studies in FeatureHouse

area is also full of challenge. In the future, more powerful tools will emerge. Cross-language tool is one example. Nowadays, many software systems contain more than one kind of developing language; and models written by different language "communicates" with each other. Although the current models can translate different kinds of developing languages, they can't handle the communication between these models. So, cross-language will become the useful tools.

7. THREATS TO VALIDITY

In this survey, we only explore seven papers completely. Although this papers were highly cited, some of them can't compare to others, since they're in the different catalog. Also, the latest one in these seven paper was published in 2013, which might not reflect the state-of-the-art techniques in 2015.

8. REFERENCES

[1] Norbert Siegmund, Sergiy S Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *Proceedings of the 34th International Conference on Software Engineering*, pages 167–177. IEEE Press, 2012.

[2] Paul Clements and Linda Northrop. Software product lines: practices and patterns. 2002.

[3] Krzysztof Czarnecki and Ulrich W Eisenecker. Generative programming. *Edited by G. Goos, J. Hartmanis, and J. van Leeuwen*, page 15, 2000.

[4] Kim Lauenroth, Klaus Pohl, and Simon Toehning. Model checking of domain artifacts in product line engineering. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pages 269–280. IEEE, 2009.

[5] Patrizia Asirelli, Maurice H ter Beek, Stefania Gnesi, and Alessandro Fantechi. A deontic logical framework for modelling product families. *VaMoS*, 10:37–44, 2010.

[6] Salvador Trujillo, Don Batory, and Oscar Diaz. Feature oriented model driven development: A case study for portlets. In *Proceedings of the 29th international conference on Software Engineering*, pages 44–53. IEEE Computer Society, 2007.

[7] Sven Apel, Christian Kastner, and Christian Lengauer. Featurehouse: Language-independent, automated software composition. In *Proceedings of the 31st International Conference on Software Engineering*, pages 221–231. IEEE Computer Society, 2009.

[8] Alessandro Fantechi and Stefania Gnesi. A behavioural model for product families. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers*, pages 521–524. ACM, 2007.

[9] Armstrong Nhlabatsi, Robin Laney, and Bashar Nuseibeh. Feature interaction: the security threat from within software systems. *Progress in Informatics*, 5:75–89, 2008.

[10] Muffy Calder, Mario Kolberg, Evan H Magill, and Stephan Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.

[11] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-Francois Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to ltl model checking. *Software Engineering, IEEE Transactions on*, 39(8):1069–1089, 2013.

[12] Dalal Alrajeh, Jeff Kramer, Alessandra Russo, and Sebastin Uchitel. Learning operational requirements from goal models. In *Proceedings of the 31st International Conference on Software Engineering*, pages 265–275. IEEE Computer Society, 2009.

[13] Christian Prehofer. Plug-and-play composition of features and feature interactions with statechart diagrams. *Software and Systems Modeling*, 3(3):221–234, 2004.

[14] Keith P Pomakis and Joanne M Atlee. Reachability analysis of feature interactions: A progress report. In *ACM SIGSOFT Software Engineering Notes*, volume 21, pages 216–223. ACM, 1996.

[15] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages

335–344. ACM, 2010.

[16] Christel Baier, Joost-Pieter Katoen, et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.

[17] Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kästner. Detecting dependences and interactions in feature-oriented design. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 161–170. IEEE, 2010.

[18] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander Von Rhein, and Dirk Beyer. Detection of feature interactions using feature-aware verification. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 372–375. IEEE Computer Society, 2011.

[19] Jia Liu, Don S Batory, and Srinivas Nedunuri. Modeling interactions in feature oriented software designs. In *FIW*, pages 178–197, 2005.

[20] Marcelo Sihman and Shmuel Katz. Superimpositions and aspect-oriented programming. *The Computer Journal*, 46(5):529–541, 2003.

[21] Tom Mens. A state-of-the-art survey on software merging. *Software Engineering, IEEE Transactions on*, 28(5):449–462, 2002.

[22] Christian Kästner and Sven Apel. Type-checking software product lines-a formal approach. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 258–267. IEEE, 2008.

[23] Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann, and Don Batory. *Language Independent Safe Decomposition of Legacy Applications Into Features*. Univ.-Bibliothek, Hochschulschr.-und Tauschstelle, 2008.

[24] Sven Apel, Christian Lengauer, Bernhard Möller, and Christian Kästner. An algebraic foundation for automatic feature-based program synthesis. *Science of Computer Programming*, 75(11):1022–1047, 2010.

[25] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[26] Moshe Y Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *1st Symposium in Logic in Computer Science (LICS)*. IEEE Computer Society, 1986.

[27] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Feature-aware verification. *arXiv preprint arXiv:1110.0021*, 2011.

[28] Harold Ossher and Peri Tarr. Hyper/j: multi-dimensional separation of concerns for java. In *Proceedings of the 22nd international conference on Software engineering*, pages 734–737. ACM, 2000.