Automated Quality Analysis of Component Software for Embedded Systems

Jens H. Jahnke
University of Victoria
Department of Computer Science
P.O. Box 3055
V8W3P6 Victoria, B.C.
Canada
jens@acm.org

Jörg Niere, Jörg Wadsack
Department of Mathematics and Computer Science
University of Paderborn
Warburger Str. 100
33098 Paderborn
Germany
[nierej|maroc]@uni-paderborn.de

ABSTRACT
The Java programming language has gained increasing importance for the development of embedded systems. To be cost efficient, such systems have to cope with significant hardware restrictions which result in certain software programming restrictions. Recently, companies have started to apply Java component technology also in the area of embedded systems. Components are pieces of software with a defined interface which can be reused in different applications. Typically, components are not developed under programming restrictions for specific embedded systems, because those restrictions depend highly on the underlying hardware. Executing such software on a micro controller with very limited resources often results in unforseen problems, e.g., in a memory overflow. Failure to detect such problems in an early stage might lead to significant costs, e.g., for replacing software on thousands of produced controllers. In this paper we present a semi-automatic approach to inspect Java source code in order to check for predefined hardware dependent restrictions. As an application domain we have choosen Java Smart Cards, which are very popular today, introduce their specific restrictions, and present how to inspect Java code to ensure that all restrictions are considered.

Keywords
Java, embedded systems, quality assurance, smart cards, pattern detection, graph rewrite rules

1 INTRODUCTION

The Java programming language conquers more and more areas of information technology. Platform independence is one of Java's main features and lets developers abstract from different operating systems or window managers. Nowadays, Java virtual machines are available for nearly each personal computer, workstation, or network server. Net-centric computing has become a new dimension in application software. Companies have started to develop net-centric computers as a consequence of this emerging technology. A main characteristic of net-centric computers is that major parts of their operating system and application software resides on the internet. Applications can be downloaded on demand and executed directly on the local machine.

Recently, companies have started to use Java for developing software for embedded systems. Coffeemakers, refrigerators, or hearths are programmed with Java and may communicate with one another over a global network. Several prototypes for such applications already exist, e.g., refrigerators which keep track of their contents and are able to place orders automatically. New operating systems like JINI [BH99] support these kinds of net-centric embedded systems.

Application software for embedded systems is typically executed by micro controllers. New controller generations include a Java Virtual Machine placed as a chip on the controller. Controller software can be programmed in Java and updated easily using component technologies like Java Beans [Pra97].

Programming micro controllers leads to problems like 10 years ago in the 'traditional' computer world. Micro controllers respectively embedded systems are typically very much resource bounded. Memory is often limited to a size under 1 MB. Embedded system software underlies several programming restrictions, e.g., memory usage of a program must be fixed after an initialization phase. Otherwise, the program may produce a memory overflow and the controller crashes. If the controller is sold million times, and such an error forces the replacement of the controller, the resulting costs are very high. To avoid such a replacement and to reduce the resulting costs, the software for embedded systems undergoes rigid testing before the controller goes in mass production.

Recently, companies have started to employ component programming technology for applications in embedded systems. Java beans can be plugged in a running application or applet [HC98a, HC98b]. Java beans are functional components with an interface that documents all provided methods. In case of embedded systems, this documentation is typically insufficient to assess whether a given component fulfills the special restrictions for micro controllers. Hence, additional information must be extracted from the Java bean itself by investigating its source code. In this paper, we apply our approach techniques to the source code of a sample component for readability. Still, our approach is not limited to source code analysis but can also be applied to Java Byte code. This is important since many commercial beans are available in Byte code only.

In a student research project called YarYar[1], we have started to develop a technique and tool support to partially automate this activity. In this approach, a tool searches the source code of a component for recurring *patterns* which indicate problems for its application in the context of embedded systems. We use *fuzzy reasoning techniques* to overcome the intrinsic problem of diversity of different implementation variants of the same conceptual pattern. This paper discusses primarily results of this ongoing research initiative. We consider the *Java Smart Card* [HNS99] as an application example and show how to extract information about memory usage from the source code.

The rest of this paper is structured as follows: Section 2 introduces Java Smart Cards and the restrictions coming along with such cards. In Section 3 solutions to overcome the memory problem are presented. Section 4 shows the specification of patterns via graph rewrite rules and introduces the need for fuzzyness. The following Section 5 introduces the execution semantics of the graph rewrite rules enhanced with fuzzyness. Related work is presented in Section 6 and Section 7 summarizes our work and gives some future perspectives.

2 JAVA SMART CARDS

Smart Cards are often used in our daily life, e.g., in form of phone cards, cards carrying health insurance information, or ‘digital’ money. Smart cards in comparision to magnetic stripe cards do not store data only, but are able to execute commands on the card. The heart of a smart card is a single chip microprocessor with a defined size and interface layout for the connection points. The chip is included in a plastic card of the size of standard credit cards (cf. Figure 1).

The technology applied in plastic cards has changed dramatically since their invention 1950 by Diners Club. Currently, magnetic stripe cards are very popular and rewritable thermo stripes can be placed on the plastic card to display further human readable information without having a terminal or card reader. The first smart card used as an identification card has been developed in 1968. In general, a smart card is a small portable computer. Today, this computer consists of a central processing unit (CPU) running at 5 MHz and mostly a cryptographic co-processor. Operating systems are stored in a read only memory (ROM) of sizes about 16K. Temporary data can be stored in a random access memory (RAM), which has typically a size of 4K. Applications, file systems, or any other data, which has to be stored over the connection time of the card to a reader must be stored in an electrical eraseable programmable read

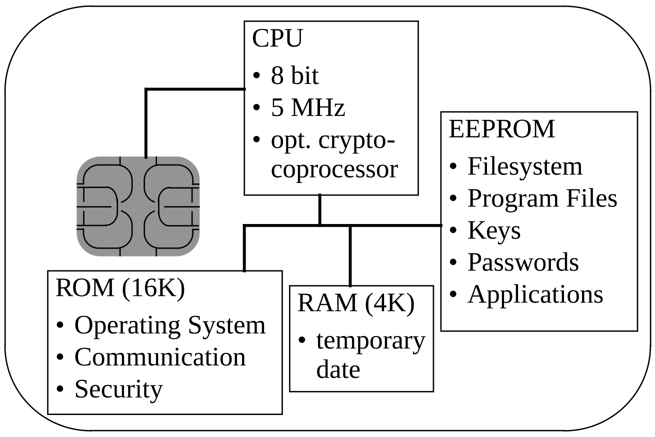


Figure 1 Smart card and components

only memory (EEPROM).

The application of EEPROMs impose several restrictions, e.g., with respect to memory allocation. High voltage over a relatively long period of time is necessary to allocate or free memory. Therefore, smart cards do not have a *memory mangement* on the card itself. In particular, the virtual machine on a Java smart card misses a garbage collector. Other limitations are that no security manager exists, no multiple threads are allowed, and objects cannot be cloned.

To avoid memory overflows, application software (cardlets) has to be tested carefully before the card goes in mass production. Current practice is to test the application in a simulation environment and check if a memory overflow is produced. If this test succeeds, the application is certified as memory overflow safe and the card is produced. In practice, this black box test works well for cards that operate for short periods only.

Preventing memory overflows can also be done in the design and implementation phase. For example, all objects needed for an application will be created in the constructor of a class. Instantiation of the first class results in constructing the entire object structure of the application during runtime. Consequently, the memory usage of an application programmed with the above paradigm, can be calculated after the initialization of all objects. Following this implementation strategy results in a short testing phase. Still, most Java components have not been developed according to this strategy.

Analogously to hardware restrictions, several software restrictions exist for Java Smart Cards. For example, the java.lang package is completely different. A String class is not supported and an int variable is always 32-bit long. Components developed for PC’s, workstations, or other less unresticted hardware platforms could not run on the a Java smart card, because the virtual machine could not execute the byte code.

However, developing components for Java smart cards lead to many programming restrictions and have to be developed explicite for Java cards and typically for a defined Java card from one manufacturer. But components are produced from different developers all over the world and they do not know

1. Fuzz**y** P**a**tte**r**n based Qualit**y** Assurance for Java Sm**a**rt Ca**r**ds

all restrictions for all Java cards in the world. Consequently, components, which should run on a special Java smart card has to be assessed for a special Java smart card. Assessing in this case means, that the component fulfills all restrictions of the smard card, the component should run on.

The following section describes a semi-automatic approach to assessing such components according to their applicability for Java Smart Cards.

3 APPLICATION EXAMPLE: MEMORY MANAGEMENT ON JAVA CARDS

Due to the restrictions of the Java virtual machine, card applets have to manage memory on their own. The easiest way to do this is to ensure that all objects needed for the cardlet are created in constructors or static initialization methods (see Section 2). This programming strategy imposes severe restrictions on the developer but the resulting software is easy to test.

Off-the-shelf components developed all over the world typically do not fulfill all restrictions of a certain Java Smart Card. This results from the variations of Java Smart Cards and their different restrictions. In general, Java software components are developed with less restrictions. Therefore, they have to be carefully inspected before using them on a card. Such an inspection aims to support the developer in selecting suitable components and detecting potential problems in an early stage. A semi-automatic inspection tool should highlight all fragments in the source code that contain (possible) violations of platform-dependent programming restrictions. After inspecting a component and testing it afterwards thoroughly, certificates could be assigned to the component with respect to the same controller. Components already assigned with certificates need not be analyzed again.

In the following, we present a facility to inspect components based on the source code. The main idea stems from the design pattern community [GHJV95, JZ97] and aims to detect *clichés*[1] in the source code and gain knowledge about the design of a problem. Declaring clichés as poor means that their existence in the code may cause possible problems in the applet during runtime. Such *poor clichés* identify parts of the source code and do not take in account further control or data flow analysis. As an example we use the property of limited memory in cardlets and show how to inspect them. We will start with poor clichés extractable directly from the source code and reduce their number by further control and data flow analysis.

The general idea of our approach is in a first step the detection of poor clichés in a restrictive way. Namely by parsing the source code we mark as poor occurance of one or an accumulation of keywords. The identification of such keywords for cliché detection depends on the application context. This paper presents a first approach for memory allocation. In a second step we try to annul or at least relax the parts of the source code marked as poor. For this again the identification of what we call *NeverMind* cliché is related to the application context. A NeverMind cliché can annul more or less (relax) a poor cliché, for this purpose we employ fuzzyness.

1. We use *cliché* as a synonym of ‘implementation pattern’, and *design pattern* refers, to [GHJV95].

```
 1: class Profile {
 2:     private List transactions;
 3:     private List shops;
 4:     private static ListIterator listIterator = null;
 5:
 6:     public Profile () {
 7:         transactions = createTransactions ();
 8:         shops = createShops ();
 9:     } // constructor
10:
11:     private List createTransactions () {
12:         List tmpTransactions = new List (10);
13:          . . . // default values
14:         return tmpTransactions;
15:     } // create Transactions
16:      . . .
17:     private List createShops () { . . . }
18:
19:     public ListIterator iteratorOfTransactions () {
20:         return new ListIterator (transactions);
21:     } // iteratorOfTransactions
22:
23:     public ListIterator iteratorOfShops () {
24:         if (listIterator == null) {
25:             listIterator = new ListIterator (shops);
26:         }
27:         return listIterator;
28:     } // iteratorOfShops
29: } // class Profile
```

Figure 2 Sample code of class Profile

Figure 2 shows the Java source code for class Profile. The class can be used to store profile information of a person using a card. Transactions and the corresponding shops can be stored in attributes transactions and shops (line 2-3). A standard container List is used to store the data. Both lists are limited, e.g., list transaction can contain 10 entries (see line 12). The limitation ensures that both lists do not allocate memory after their initialization but fix it to a defined size.

Following the programming paradigm, that every object needed must be created in the constructor or static initializer, class Profile has to be declared as a possible source of failure. Placing the bodies of method createTransitions and createShops into the constructor, the resulting code is closer to the paradigm, because objects are created in the constructor. But this lets the code become less readable and the creation of a ListIterator object in method iteratorOfTransactions and iteratorOfShops are still possible failure points. This example shows that one of the main challenges with developing an automated approach to *source code inspection* (i.e. *pattern detection*) is to cope with the large diversity of possible implementation variants of the same pattern. We will cover this issue later in this paper by employing *fuzzy pattern* matching techniques.

Our approach is called semi-automatic, because a tool searches the source code for instances of poor cliché and generates an annotated representation of the code that is presented to the developer for further manual investigations. Internally, the source code is represented as an abstract syntax tree (AST). Hence, the domain model for the cliché annotation represents an extension of this data structure. This extension is represented in Figure 3 as a UML class-diagram (cf. [BRJ99]).

We start with a modified *Composite* design pattern (cf.[GHJV95]), where classes Annotation and SyntaxTreeNode inherits from class Node. The central class Annotation is derived from the notion of *collaboration* in UML (cf. [BRJ99]). We use the qualified association annotations to represent relationships between annotations and between annotations and annotated increments (represented by instances of class SyntaxTreeNode). The annotations are qualified over a name and denoted as '<<name>>' at the corresponding edges (cf. Figure 4). Two additional classes inherit from class Annotation, namely Cliché and Component. Clichés correspond to implementation patterns in contrast to Components, which represent parts in the AST or control or data flow analysis. Class Cliché has two subclasses Poor and NeverMind. Instances of class Poor annotate poor clichés in source code, i.e., possible violations of imposed restrictions in the code. Therefore, class KeywordNew is introduced as subclass of class Poor. On the other hand, poor clichés may be annulled or relaxed. If a poor cliché is annulled, the inspector can ignore it and accordingly the annotation will not be highlighted. Spots which annul poor clichés are annotated

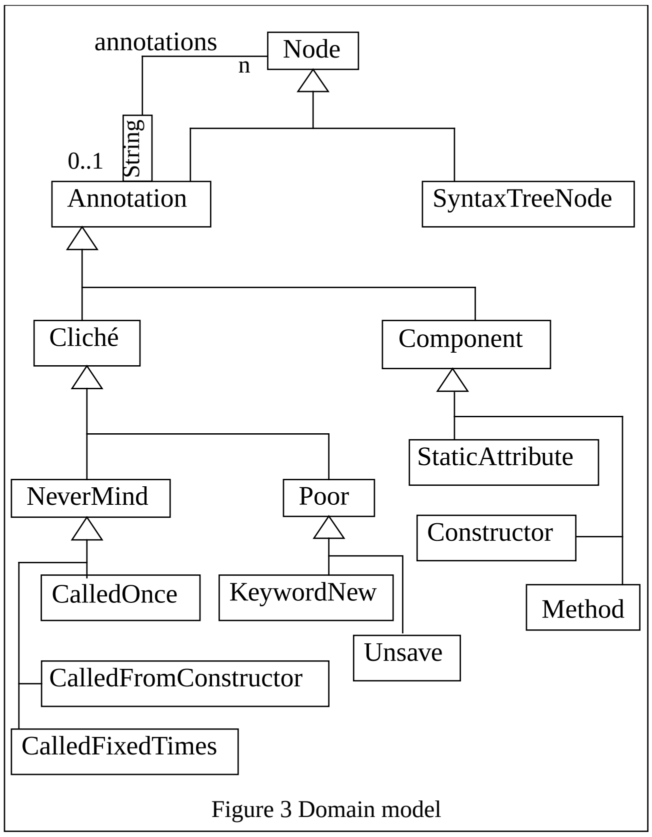


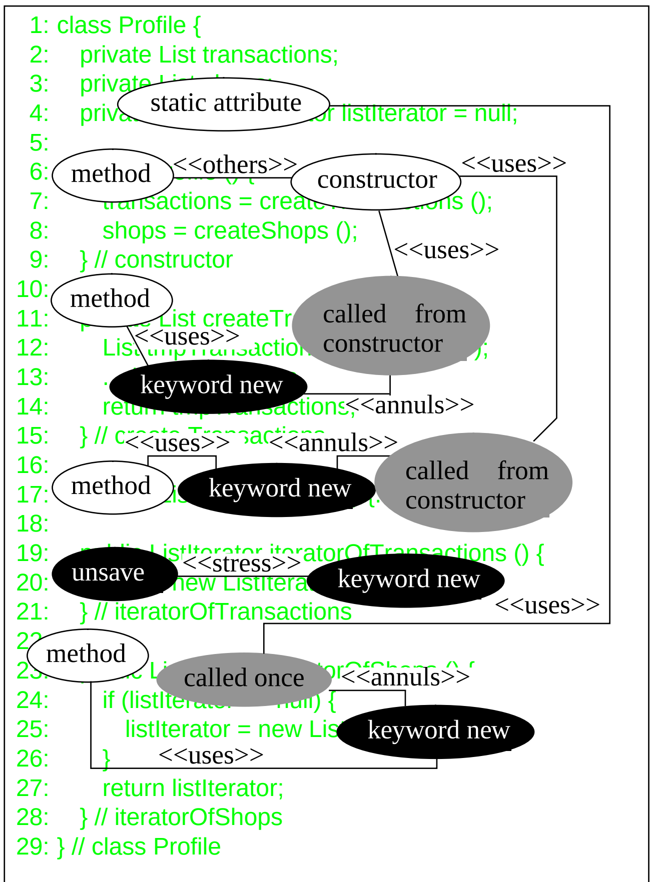
Figure 3 Domain model

with instances of subclasses of class NeverMind. Two examples for such classes are CalledOnce and CalledFromConstructor. CalledFromConstructor refers directly to the programming paradigm mentioned above and CalledOnce implies that there is only one object during runtime. In case of relaxing clichés, the inspector could define a value, up there the corresponding annotation is shown. An example of a subclass of NeverMind which relaxes a poor cliché is CalledFixedTimes. To stress real failure points, class Unsave is used. Class Component models additional information used for example to identify a cliché's context (see classes Method, Constructor and StaticAttribute in Figure 3). Our method starts with so called fragments (instances of subclasses of class Cliché in Figure 3). Fragments are information that cannot be refuted such as key words used in the source code or other information gained from data flow or control flow analysis of the same code, definetively. For example, we take the occurance of the keyword new as a fragment, because it indicates that there might be a possible memory overflow constructed. For class Profile there are four occurances of the keyword new, one in line 12, line 20, and one in line 25. The fourth occurance is not shown in Figure 2, but both methods, createShops and createTransitions, construct a new List object. A second fragment resulting from the source code is the existence of methods. Class Profile consists of five methods, including the constructor which occurance is the third fragment we use. Data flow and control flow analysis of the class results in two more fragments. First, method createTransition and createShops are called from the constructor method only. Concerning the new keyword in line 25, control flow analysis had detected that the assignment is done in an if-statement's body, which is executed only one time, because the attribute listIterator is assigned with the new interator and is nowhere else modified in the class. Fragment 'unsave' annotates the 'keyword new' fragment and indicates the only real problem point in the source code.

With these fragments we are able to construct a graph, where fragments are the nodes and associations are edges. (Figure 4). Annotations of the code are made via associations of type annotations in the AST. Figure 4 shows the *annotation graph* as an overlay. This is only for readability reasons. Users will see only the critical (unsave) spots in the code highlighted. The annotation graph contains three different kind of nodes and four different occurances of association annotations. Nodes represented with a white background and black font are so called informational fragments (instances of class Component). Those fragments are used for conclusions and have no effect on the evaluation of possible problems in the source code. Possible problems, we call them *poor nodes* are displayed with black background and white font. Here such nodes are 'keyword new' nodes, because they are representing possible memory overflow fragments (instances of class Poor). Grey background and black font nodes represents instances of class NeverMind. Such a node annuls connected poor nodes. For example, in Figure 4 the 'called from constructor' node annuls the poor 'keyword new' node, which means that there is no possible memory overflow caused by this statement.

This retains also for the second ‘called from constructor’ and the ‘called once’ node. Overall, there is only one poor ‘keyword new’ node left, which can not be annuled and is annotated via ‘unsave’ nodes connected via ‘<<stress>>’ links. Only the creation of a new list iterator in method iteratorOfTransitions is a possible failure point and should be highlighted in the code.

Now, we are able to analyze cardlets based on there source code and extract problems in the implementation efficently by reducing general poor fragments and keeping track only on those, which are really problematic. The following section introduces a specification opportunity of clichés and the need for fuzzyness.



```
class Profile {
    private List transactions;
    private
    priv                 r listIterator = null;

       transactions = create              ons ();
       shops = createShops ();
    } // constructor

           List createTr
       Lis                                 );

       return tmpTransactions;
    } // c

    public ListIterator iteratorOfTransactions () {
               new ListItera
    } // iteratorOfTransactions

        L                      orOfShops () {
       if (listItera          null) {
          listIterator = new Lis
       }
       return listIterator;
    } // iteratorOfShops
} // class Profile
```

Figure 4 Fragments and relations

4 CLICHÉS SPECIFICATION

Graphs like the annotation graph and graph transformations are offen defined by *graph grammars* [Roz97]. Graph grammars allow a manipulation of graphs via rewrite rules, specified in a high level specification language without caring about the execution. Figure 5 is an example of a graph rewrite rule. In general, the execution of a graph rewrite rule consists of two parts. In the first part variables in the rule are matched to nodes in the working graph (host graph) of the system. Typically, this is done via an isomorphic embedding of the subgraph of the rule into the host graph. The second part executes the modifications of the host graph specified in the rule.

Graph transformation systems become more and more popular in the last years. The Progres system [Zün95] and the AGG System [Tec] follow a classical approach, where the host graph as well as the rewrite rules has to be specified in the system itself and the rules could only modify the host graph internally. Control structures allow to bring the rewrite rules in a defined order. Both systems provide an interface to access the graph and execute rules from outside the system.

Fujaba [FNT98, FNTZ98] follows the approach of a *seamless integration* of graph rewrite rules and traditional object-oriented programming languages. Fujaba uses UML [Rat] for the specification of software systems. The node types of the host graph could be specified via an UML class diagram and instances of classes called objects represent the host graph during execution time. UML activity diagrams are used to specify control structures and each activity might contain a graph rewrite rule specified by an UML collaboration diagram (cf. Figure 5). The combination between control flow and graph rewrite rules are called methods. Fujaba provides also a Java code generator to get an executable implementation out of the specification.

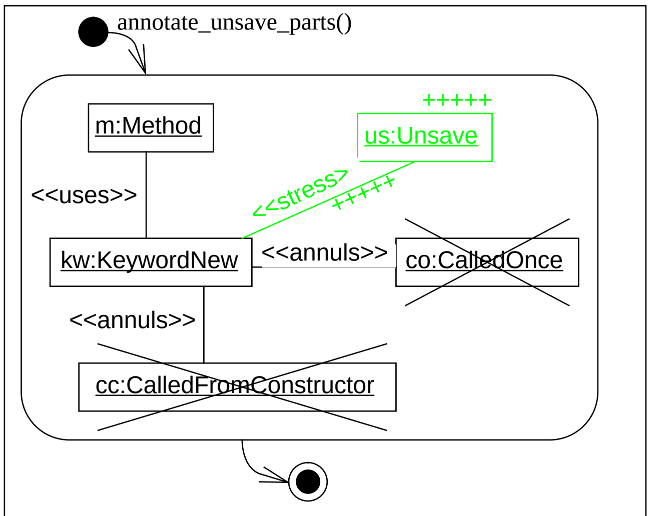


Figure 5 Example rule looking for unsave parts

In general, we use the Fujaba system for the specification of clichés and the domain model. The domain model is specified via a class diagram and the clichés through methods including graph rewrite rules. Each cliché results in a search of a subgraph by executing a graph rewrite rule. If the subgraph could be matched to the host graph, a new annotation is created.

The method annotate_unsave_parts specified in Figure 5 starts at the start activity, executes the graph rewrite rule and after successful or non-successful execution it ends at the stop activity. The method specifies the cliché for annotating unsave parts in the source code. Therefore, the graph rewrite rule searches for a node in the graph of type Method and binds it to variable m:Method in the rule[1]. Next, a node of type KeywordNew is bound to variable kw. Those two nodes have to be connected via a <<uses>> link. The two crossed

1. Variable notation corresponds to object notation in UML collaboration diagrams (i.e. name:type)

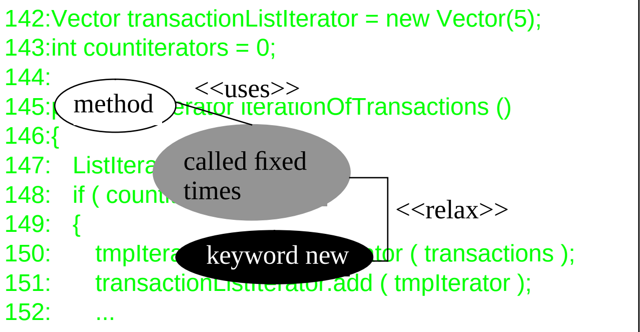
out variables co of type CalledOnce and cc of type CalledFromConstructor specify that there must not be a node in the graph reachable from the node bound to variable kw via an <<annuls>> link. If such a subgraph has been found and nodes are bound to the variables, a new node of type Unsave is created in the graph and linked via a <<stress>> link to the node bound to variable kw. The creation of the node and the link is specified through (green/grey) plusses at the corresponding variable and link.

In this case graph rewrite rules are sufficient for the specification of clichés, because the nodes of the graph represent non-refutable fragments in the source code. For example, the 'method' node occures in the graph if there is a underlying method declaration in the source code. Even the occurance of constructors and static attributes could be taken from the syntax graph of the source code, easily. Data flow and control flow analysis results in the occurance of the 'called once' and 'called from constructor' nodes. All these nodes eval in a boolean answer. The pattern matching semantics of graph rewrite rules support these kinds of boolean answers. Variants of a cliché result in an own definition for each variant and the number of rules can explode.

```
142:Vector transactionListIterator = new Vector (5);
143:int countIterators = 0;
144:
145:public ListIterator iteratorOfTransactions ()
146:{
147:   ListIterator tmpIterator;
148:   if ( countIterators < 6 )
149:   {
150:      tmpIterator = new ListIterator ( transactions );
151:      transactionListIterator.add ( tmpIterator );
152:      countIterators++;
153:   } else
154:   {
155:      for ( int i = 0; i < countIterators; i++ )
156:      {
157:         tmpIterator = transactionListIterator ( i );
158:         if ( !tmpIterator.isUsed () )
159:         {
160:            i = countIterators;
161:         } // endif
162:      } // for
163:   } // endif
164:   if ( tmpIterator.isUsed () )
165:   {
166:      throw new RuntimeExeption( ...// Error message );
167:   }
168:   return tmpIterator;
169:}
170:...
```

Figure 6 Complex code for method iteratorOfTransactions

In general, using uncertainty reduces the number of rules and allows to deal with miner than 100% clearity. Figure 6 shows a sample code fragment of a personal management of ListIterators. The extension of method iteratorOfTransactions uses a vector which can contain five iterators (cf. line 142)



```
142:Vector transactionListIterator = new Vector(5);
143:int countiterators = 0;
144:
145:(     erator iterationOfTransactions ()
146:{
147:   ListItera
148:   if ( count
149:   {
150:      tmpItera                  tor ( transactions );
151:      transactionListIterator.add ( tmpIterator );
152:      ...
```

Figure 7 Example for a CalledFixedTimes instance

and manages its usage within the class. In line 150 the keyword new occurs which indicates a possible memory overflow, cf. Section 3. Again the occurance of keyword new is encapsulate in an if statement (see line 24 in Figure 2). But the vector is used in the part below and explicitely may be modified. Ensuring that the number of itertaors in the vector does not exceed its capacity or double entries may result in memory leaks needs heavy further analysis. Each implementation variant results in an own cliché specification. This results from the fact, that the limitation of number of iterators and the handling can be implemented in multiple ways. Even using different container classes may result in a complete different implementation. Therefore, we provide fuzzyness to model relaxation of possible memory overflows. The above scenario is shown in Figure 7 with the already introduced annotation graph. Note, the edge between node 'called fixed times' and 'keyword new' corresponds to the annotations association in the domain model in Figure 3 and is named '<<relax>>' and not '<<annuls>>'. Relaxing means that this is not a 100% fragment but may be a 80% one, because we left out heavy further automatic investigations on the source code but let infer the reengineers knowledge to save time.

As mentioned before, catching every possible variant of

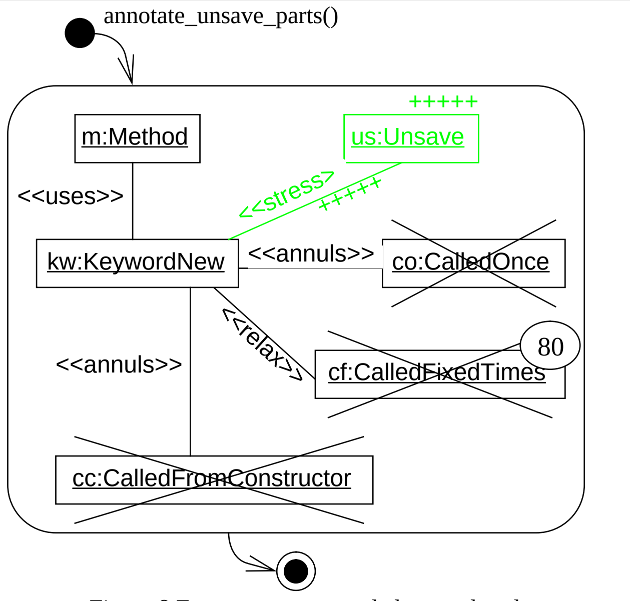


Figure 8 Fuzzy pattern extended example rule

managing list iterators, results in traditional graph transformation systems into one rule for each variant. A possible solution to overcome this problem is to soften the exact match semantics of graph rewrite rules. Therefore, Fujaba and the other systems allows to specify a variable in the rule as optional, which means that a corresponding node will be bound to the variable if there exists one. Otherwise the variable will be left unbound. Optional variables are not sufficient enough in this case, because they do not support fuzzyness. We define fuzzyness in a graph rewrite rule with a circle in the upper right corner of a variable and call them *fuzzy pattern*. Figure 8 shows the extended rule of Figure 5 with variable cf of type CalledFixedtimes and fuzzy value 80. The notation of fuzzy values does not correspond to a notation in UML, but the values correspond to attributes of the class and showing them in the upper right corner of the variable instead as attributes in the attribute compartment is just display option.

The fuzzy value of variable cf is taken into account when calculating the fuzzy value of the created 'unsave' node. Variables without special fuzzy value refer implicite to a value of 100. We do not show them in the rule refering to the readability. The fuzzy value of the new node created when the rules could be matched is calculated from the values of the other variables in the rule.

Remember, a reengineer will only see 'unsave' nodes highlighted in the source code. The fuzzy value of an 'unsave' node represents a kind of trustability of the result. Typically, the reengineer has to look for nodes with a fuzzy value minor than a defined limit and could overwrite the value to zero or 100. The changes effect a reevaluation of the fuzzy values of the depending nodes. Therefore, our approach is semi-automatic. Experiences have shown that such a semi-automatic approach causes better results, because a user or reengineer could interact with the tool and infer her/his knowledge.

5 DETECTING CLICHÉS

The detection mechanism for specified clichés is based on the formalism of *Generic Fuzzy Reasoning Nets* (GFRN) [JSZ97, Jah99]. This formalism which has initially applied in the domain of data reverse engineering facilitates the specification and execution of analysis rules and processes and incorporates a notion of uncertainty. In principle, a GFRN is a graphical network of *predicates* (with oval shape) and *implications* (represented as boxes) which are connected by arcs (cf. Figure 9). Each implication has an associated *confidence value* (CV). Based on the theory of possibilistic logic [DLP94], the semantics of a CV is a lower bound of the necessity that the corresponding implication is valid. Arcs are labeled with formal parameters that can be used to specify constraints for implications. Negations in implications are represented by arcs with black arrow heads.

For each fuzzy pattern, we create a GFRN by using a canonical translation procedure. Figure 9 shows the GFRN representation of the the sample fuzzy pattern shown in Figure 8. Nodes that are searched in the fragment graph are represented by predicates rendered in grey color while the annotation which represents the goal of the analysis is represented as a so-called *dependent* predicate 'Unsave' rendered in black.

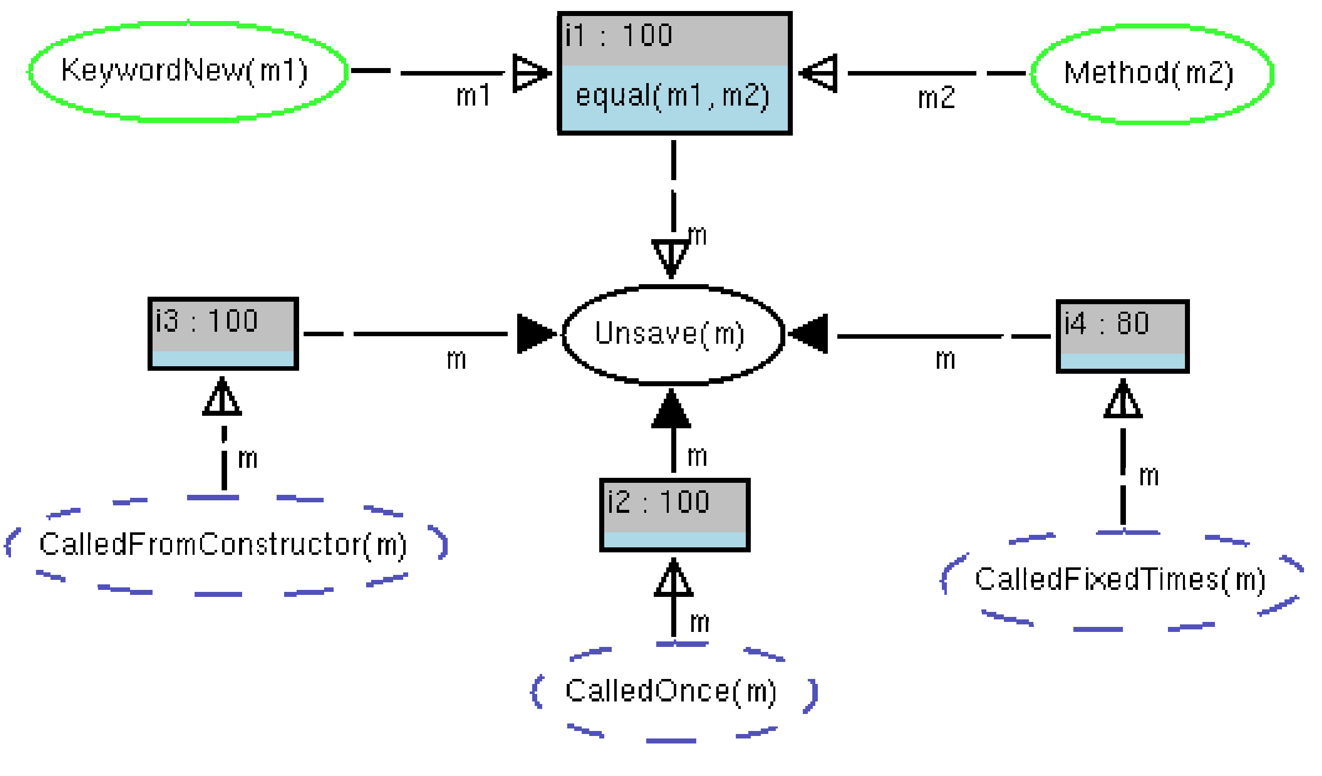


Figure 9 Generic Fuzzy Reasoning Net

For efficient fuzzy pattern analysis, we propose a two-step process: firstly, the fragment graph is searched for a subgraph that matches all positive nodes in the corresponding graph rewrite rule (cf. Figure 8). Subsequently, our detection strategy aims to extend each such match by a match for the negative (canceled) nodes in the rewrite rule. The GFRN formalism facilitates the specification of such a strategy by distinguishing between so-called *data-driven* and *goal-driven* predicates. Matches for data-driven predicates (represented with solid grey outline) are searched at the beginning of the analysis process (cf. 'KeywordNew' and 'Method' in Figure 9). If such matches can be found and all implication constraints hold the GFRN execution mechanisms creates instances of the corresponding dependent predicate ('Unsave' in Figure 9). Subsequently, the fragment graph is searched for matching instances of goal-driven predicates (with dashed grey outline). Since, these goal-driven predicates have been created for negative nodes in the corresponding graph rewrite rule, their existence annuls or relaxes the existence of the concluded poor clichés (cf. arcs with black arrow heads in Figure 9). The CV associated to each implication specifies a measure for the degree of this relaxation. Note that CV of 80 for implication 'i4' corresponds to the fuzzy value of CalledFixedTimes in Figure 5. According to typical fuzzy inference operators, an overall valuation for each tentative poor cliché is based on the difference between the maximum positive CV and the maximum negative CV. We refer to [JH98] for details on the GFRN inference engine.

6 RELATED WORK

Rayside and Kontogiannis have developed tool-based techniques to minimize the size of Java application software for small-size electronic devices [RK99]. A similar technique is proposed by Korn et al. for selective regression testing of Java components [KCK99].

In [HN90] program analysis is based on an *Event Base* and a *Plan Base*. In a first phase events are constructed from source code. Plans are used to consume one or more events and fire a new event which correspond to the plans intention.

An automatic approach to extract semantics information from source code is presented by Wills [Wil94]. Analogously to our approach, Wills uses graph rewrite rules to specify implementation patterns in terms of so-called *program plans*. Program plans are structured in a hierarchical design library, i.e., abstract plans consist of aggregations of several more simple plans. A specific library stores program-plans for different domains, e.g., Wills presents a library for sorting algorithms. Wills follows a bottom-up strategy to detect plans according to this hierarchy. However, the inherent complexity of this bottom-up search limits the practical usage of the approach to source code about 1000 lines. Quilici uses an indexing technique and combine top-down and bottom-up detection to overcome this problem [Qui94].

In [KSRP99], Keller et al. present a semi-automatic approach to find design patterns [GHJV95] in source code is introduced. These design pattern are represented in UML notation [BRJ99], namly in CDIF format. Matching algorithms have to be implemented for each design pattern by hand. Rademacher employs graph transformations to extract design patterns automatically and to refacture poor design pattern with good design pattern [Rad99].

7 CONCLUSIONS AND FUTURE WORK

This paper presents an approach for automatic quality analysis of components for embedded systems. Embedded systems come along with hardware limitations and components have to fulfill partial many programming restrictions. As application example we use Java Smart Cards. Those cards are memory limited and there is no garbage collector on the card. Cardlets, applications running on the card, have to manage their memory usage by themselfs. We introduce a solution to inspect cardlets if they produce no memory overflow by defining fuzzy patterns. Fuzzy patterns are defined using graph grammars. The huge number of implementation variants for memory management results in a fuzzy pattern definition for each variant. To catch many variants in one fuzzy pattern definition we introduce fuzzyness into the definition. This fuzzyness let us deal with uncertainty. As detection mechanism for fuzzy patterns, we use Generic Fuzzy Reasoning Nets.

We are currently working on a first implementation of our approach. Therefore, we enhance the Fujaba system, which already supports the instanciation of design patterns and a rudimentary mechanism to extract design patterns out of Java source code. In the YarYar student research group we plan to enhance our approach by inheritance and polymorphism like in object-oriented languages. Fuzzy patterns will inherit from other fuzzy patterns and during the extraction process, polymorphism will be used. Therefore, fuzzy patterns must provide a defined interface and inheritance as well as polymorphism must be mapped to the underlying Generic Fuzzy Reasoning Nets. The advantage of using inheritance is to specify sub fuzzy patterns of a more general super fuzzy pattern without defining a complete new one. For example, a fuzzy pattern detecting aggregations between classes could inherit from a general association detecting fuzzy pattern. Because typically, the difference between them is an explicite deletion of the aggregated objects when using aggregations.

The current prototype of the Fujaba environment is available as free software and comprises about 230 000 lines of pure Java code. Additional information and the current release version of the Fujaba system can be downloaded via:

`http://www.uni-paderborn.de/cs/fujaba/`

REFERENCES

[BH99] H. Bader and W. Huber, editors. *Jini*. Addison Wesley, 1999.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, Reading, Massachusetts, USA, 1 edition, 1999.

[DLP94] D. Dubois, J. Lang, and H. Prade. Possibilistic logic. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and*

Logic Programming, pages 439–503. Clarendon Press, Oxford, 1994.

[FNT98] T. Fischer, J. Niere, and L. Torunski. *Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling (in german)*. Master's thesis, University of Paderborn, Paderborn, Germany, July 1998.

[FNTZ98] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In G. Engels and G.Rozenberg, editors, *Proc. of the 6th Int. Workshop on Theory and Application of Graph Transformation, Paderborn, Germany*. Springer Verlag, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[HC98a] C.A. Horstmann and G. Cornell. *Core Java 2, Volume 1: Fundamentals*. Java Series. Prentice Hall, first edition, 1998.

[HC98b] C.A. Horstmann and G. Cornell. *Core Java 2, Volume 2*. Java Series. Prentice Hall, first edition, 1998.

[HN90] M. T. Hanrandi and J. Q. Ning. Knowledge Based Program Analysis. In *Journal IEEE Software, volume 7, number 1*, pages 74–81, January 1990.

[HNS99] U. Hausmann, M.S. Nicklous, and T. Schäck. *Smart Card Application Development Using Java*. Springer Verlag, 1999.

[Jah99] J.H. Jahnke. *Management of Uncertainty and Inconsistency in Database Reengineering Processes*. PhD thesis, University of Paderborn, Paderborn, Germany, September 1999.

[JH98] J.H. Jahnke and M. Heitbreder. Design Recovery of Legacy Database Applications based on Possibilistic Reasoning. In *Proceedings of 7th IEEE Intl. Conf. of Fuzzy Systems (FUZZ'98). Anchorage, USA.*. IEEE Computer Society Press, May 1998.

[JSZ97] J.H. Jahnke, W. Schäfer, and A. Zündorf. Generic Fuzzy Reasoning Nets as a basis for reverse engineering relational database applications. In *Proc. of European Software Engineering Conference (ESEC/FSE)*, number 1302 in LNCS. Springer Verlag, September 1997.

[JZ97] J.H. Jahnke and A. Zündorf. Rewriting poor Design Patterns by Good Design Patterns. In Serge Demeyer and Harald Gall, editors, *Proc. of the ESEC/FSE Workshop on Object-Oriented Re-engineering*. Technical University of Vienna, Information Systems Institute, Distributed Systems Group, September 1997. Technical Report TUV-1841-97-10.

[KCK99] J. Korn, Y.-F. Chen, and E. Koutsofios. Chava: Reverse Engineering and Tracking of Java Applets. In *Proc. of the 6th Working Conference on Reverse Engineering, Atlanta, USA*. IEEE Computer Society Press, October 1999.

[KSRP99] R.K. Keller, R. Schauer, S. Robitaille, and P. Page. Pattern-Based Reverse-Engineering of Design Components. In *Proc. of the 21th Int. Conf. on Software Engineering, Los Angeles, USA*, pages 226–235. IEEE Computer Society Press, May 1999.

[Pra97] S. Prashant. *Java Beans developer's resource*. Prentice Hall, 1997.

[Qui94] A. Quilici. A memory-based approach to recognizing programming plans. *Communications of the ACM*, 37(5):84–93, May 1994.

[Rad99] A. Radermacher. Support for Design Patterns through Graph Transformation Tools. In *Proc. of Int. Workshop and Symposium on Applications Of Graph Transformations With Industrial Relevance (AGTIVE), Kerkrade, The Netherlands*, LNCS. Springer Verlag, 1999.

[Rat] Rational Software Corporation. *UML documentation version 1.3 (1999). Online at http://www.rational.com*.

[RK99] D. Rayside and K. Kontogiannis. Extracting Java Library Subsets for Deployment on Embedded Systems. In P. Nesi and C.Verhoef, editors, *Proc. of the 3rd European Conference on Software Maintenance and Reengineering (CSMR), Amsterdam, The Nederlands*, pages 102–110. IEEE Computer Society Press, March 1999.

[Roz97] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific, Singapore, 1997.

[Tec] Technical University of Berlin. *AGG, the Attributed Graph Grammar system. Online at http://www.tfs.cs.tu-berlin/agg*.

[Wil94] L.M. Wills. Using Attributed Flow Graph Parsing to Recognize Programs. In *Int. Workshop on Graph Grammars and Their Application to Computer Science*, Williamsburg, Virginia, November 1994.

[Zün95] A. Zündorf. *PROgrammierte GRaphErsetzungsSysteme*. PhD thesis, RWTH Aachen, 1995.