Pattern recognition in the MAISA tool

Lilli Nenonen, Juha Gustafsson
Department of Computer Science
University of Helsinki, Finland

August 21, 2000

1 Introduction

In this document we focus on finding patterns in class diagrams. However, other types of diagrams are structurally similar, so the techniques discussed here can also be applied to them. Diagram specific problems can still occur, although most of them are caused by the incompleteness of design information.

An effective way of recognizing patterns is using the constraint satisfaction problem (CSP) technique. The pattern information is collected into a pattern library that is used by the pattern recognition algorithms as well as the human users of the patterns. The MAISA tool provides a user interface to the library for the human users.

In Chapter 2 we discuss the constraint satisfaction problem in general. In Chapter 3 we present a simple, but relatively effective algorithm for this problem and in Chapter 4 we discuss user interaction as a way to extend the basic algorithm. The algorithm including the extensions discussed here has been implemented by Pauli Misikangas [Mis1997].

In Chapter 5 the pattern library and its implementation is presented. A brief example of pattern recognition is discussed in Chapter 6 and conclusions are drawn in Chapter 7.

2 Constraint satisfaction in pattern searching

Constraint satisfaction [Kum1992, Mac1977, Mac1992, MoH1986] is a generic problem that can be applied to wide variety of tasks, in our case pattern searching. The CSP setting consists of variables and a set of constraints restricting the values that can be assigned to those variables. *Unary constraints* (denoted as P_i) restrict the values for a single variable, while *binary constraints* (denoted as P_{ij}) represent a condition for a pair of variables. The CSP is often modeled as a graph, where the nodes represent the variables and the arcs represent the constraints.

Formally, CSP can be stated as follows [Mac1977]:

$(\exists x_1)(\exists x_2)...(\exists x_n)P_1(x_1)\wedge P_2(x_2)\wedge ...\wedge P_n(x_n)\wedge P_{12}(x_1,x_2)\wedge P_{13}(x_1,x_3)\wedge ...\wedge P_{n-1n}(x_{n-1},x_n)$, where P_{ij} is included when $i<j$.

In practical terms, variable domains (D_i) must consist of a finite number of discrete values. Even so, the solution of trying out all combinations is too slow. In addition, most combinations would make no sense, so it's no use to try them. We may try a particular value several times, even if there is no way that value could be a solution for a given variable. Therefore we must find a way to effectively prune out impossible candidates.

It is not always possible or practical to find a complete solution. If we allow *partial satisfiability* [FrW1992], we may accept those solutions that violate (to a certain extent) some of the constraints. In this situation, the constraints do not offer only exclusive alternatives. We may define our criteria separately for each case. A disadvantage of this technique is that the number of potential solutions may go up quite rapidly. Some research has been done regarding the case of partial satisfiability and it may suit our problem, as the patterns themselves are not always well-defined (discussed further in Chapter 4).

The time-complexity of CSP algorithms is discussed e.g. in [MaF1985], which gives the AC-3 algorithm the value $O(a^3n^2)$, where n is the number of nodes and a is the size of the variable domains (considered simply to be equal for all domains). For sparse graphs a smaller value of $O(a^3n)$ is given.

We define our pattern searching problem as a CSP in the following way:

- The variables (nodes) represent the roles of a pattern
- The variable domains are initialized to contain all the names (identifiers) of the diagram(s) in question
- Unary constraints represent conditions for a single role (e.g. the element in role X must be of type abstract class).
- Binary constraints represent conditions between two roles (e.g. the class in role X must be a subclass of the class in role Y).

For each pattern we have a result, i.e. the bindings that describe this particular pattern. The number of these bindings depends on the pattern in question. A binding is a pair $\{role, element\}$, where *role* is the name of the role and *element* is the diagram element that appears in that role, e.g. in the **Abstract Factory** pattern two of the roles are *AbstractFactory* and *AbstractProduct*.

3 Constraining search-space

A simple and useful way of testing the candidate values is *backtracking*, where the conditions are tested for each value. If the conditions are not met, that value is discarded. Before we try backtracking, we make sure that there are no unsuitable values in the domain of each variable. This means that if we require that a certain variable can only have class-typed values, then we can prune all attributes, methods etc. from its domain. This way we can make the number of candidates as small as possible.

3.1 AC-3 algorithm

The most trivial requirement is node consistency. Node i is *node consistent*, iff $\forall x \in D_i, P_i(x)$ holds. The following algorithm ensures node consistency.

```
procedure NC-1:
begin
      for i ← 1 until n do
         begin
             D_i ← {x ∈ D_i | P_i(x)}
         end
end
```

Thus for example, all attribute-entities would be pruned from the domain of a variable having a constraint that allows only solutions of type class.

Arc consistency is defined in a similar fashion: Arc (i, j) is *arc consistent*, iff $\forall x \in D_i$ such that $P_i(x)$ holds, $\exists y \in D_j$ such that $P_j(y)$ and $P_{ij}(x, y)$.

A single arc can be revised using the following procedure REVISE that returns a boolean value. The idea is similar to the node consistency. We delete all values from the domain of the originating node D_i, for which there are no 'legal' arcs. Discussion of arc consistency can be found in [MoH1986].

```
procedure REVISE((i,j)):
begin
      DELETE ← false
           for each x ∈ D_i do
           if ∄y ∈ D_j, such that P_ij(x, y) then
             begin
                 delete x from D_i
                 DELETE ← true
             end
      return DELETE
end
```

The AC-3 algorithm utilizes the first node consistency and then the arc consistency revision algorithm as follows. We denote the entire CSP graph with G and the respective set of arcs (constraints) with $arcs(G)$. Additionally we denote the current (non-consistent) set of arcs with Q, which means that the algorithm ends, when Q is empty.

```
procedure AC-3:
begin
      NC-1
      Q ← {(i, j)|(i, j) ∈ arcs(G), i ≠ j}
      while Q not empty do
        begin
           select and delete any arc (k, m) from Q
           if REVISE((k, m)) then
                Q ← Q ∪ {(i, k)|(i, k) ∈ arcs(G), i ≠ k, i ≠ m}
        end
end
```

After the domains have been made consistent, we search correct bindings among the remaining values that satisfy the current set of constraints. In the simple case we have only one value for each variable.

3.2 Extensions

Many design patterns are 'related' to each other in the sense that they have common elements (see e.g. metapatterns in [Pre1995]). These relationships may be taken advantage of in two ways: the ordered search of patterns and the use of auxiliary facts [Mis1997]. This information is updated as the search proceeds, when a particular pattern is being searched, new facts are added. These facts can then be utilized later on when searching other patterns. Consider, for example, that we are searching for instances of *Abstract Factory* pattern which has the *Factory Method* pattern as its prerequisite. We would now take advantage of a new fact *Factory Method* that has been added while searching for instances of *Factory Method* pattern.

The program currently uses three types of files: diagram, pattern and pattern index files. Diagram files contain the diagram information represented as Prolog facts. A pattern file (one per pattern) contains the pattern (i.e. the respective set of constraints) also as Prolog facts. The index file includes (for each pattern):

- Prerequisites (i.e. those patterns that are searched before this one)
- Default file names for pattern and output files
- Auxiliary facts that are generated, when this pattern is being searched

4 Interaction

The AC-3 (and generally any other purely syntactic) algorithm may still produce a large number of false positives, when we have a non-trivial task like finding vaguely defined design patterns. To make matters worse, several fairly common design patterns have features that are very difficult or even impossible to model as a set of constraints. In these cases human intuition and insight is valuable, if we may present the dubious bindings for the user.

Many design patterns are too abstract to be easily represented syntactically [Gam1995]. The situation becomes even more complex if we require a fine-grained classification of separate pattern instances. Consider the situation where finding instances of the metapattern **1:1 Connection** [Pre1995] is not enough, but we want to make the distinction between **Bridge** and **Command** patterns (cf. the example in Chapter 6). Their syntactic structure is alike so an attempt to automatically recognize them would not be very accurate.

Another related problem is that in many cases the design diagrams simply do not contain enough information. Associations are a typical example. This notation has quite a lot of expressive power. An association can be implemented using several different techniques. A common case would be to include an attribute in one of the classes containing a reference to the other class or to have a class that calls a method of another class. During the design phase the more general representation is usually enough. Either we do not know the implementation details, or we do not wish to fix them. However, in order to recognize many common design patterns (such as **Abstract Factory** and **Builder**), we need to know these connections explicitly. In these cases we either have to include more detailed information in the UML diagrams or try to find the patterns using incomplete information. The former alternative is not viable in practice, as in most cases we simply do not have (or even need) the required level of detail in the design phase. As a solution to the latter case partial satisfiability techniques might be worth investigating.

Even when dealing with correct positive instances of design patterns, the number of possible bindings can become large (e.g. when searching for **Composite** or **Mediator** patterns). The number of elements that can participate in a certain role in a pattern is not limited. The basic CSP algorithm would try to find them all. This is also a situation, where human interaction would be quite helpful.

An important issue is that the rules describing the patterns are correct. This is even more important, if the semantics of the pattern are complex. Missing or false constraints may either produce a number of false positives (which can be frustrating) or false negatives (which is what we would most like to avoid). This issue might seem obvious, but considering the small semantical subtleties many patterns have, finding the correct representation for a pattern is not necessarily trivial.

To be of any use this kind of interaction naturally requires a highly knowledgeable user (knowledge of both design patterns and the problem domain is essential). It must be emphasized that interaction is usually not required, the AC-3 algorithm produces results relatively fast even when working with larger domains (for more detailed results, see [Mis1997]).

Many features discussed here, such as the verification of potential bindings or the presentation of design patterns, require that the MAISA user interface is extended. For the time being, only a textual presentation is available. In the future, more usable alternatives will be developed.

5 The Pattern library

The user can update the pattern library in the pattern window of the MAISA tool or with the common file handling tools of the operating system (the former way is more user friendly to novice users, while the latter is quicker and more efficient). Patterns or pattern groups can be added, edited and removed from the library. These functions are intended for administrative use. Patterns and pattern groups can also be opened and closed in the user interface of the tool without adding of removing them from the library. These functions are intended to more frequent use.

A pattern is identified with pattern name and group name. There can be patterns with the same name belonging to different pattern groups, for example versions of the same pattern.

The physical pattern library is a directory. The root directory contains one subdirectory for each pattern group. The group name is the name of the subdirectory, with underline characters replaced with spaces. The group directories contain one subdirectory for each pattern belonging to the group. The name of the pattern directory is the name of the pattern, with spaces replaced with underline characters. The group names and the pattern names do not contain any underline charecters and the directory names do not contain any spaces, in order that the directory structure works in the environments where file names with spaces are not supported. The root directory also contains one prolog file which is common to all the patterns and which is called **common.pro**. It defines the order in which the patterns are recognized, as explained in the previous chapter. The user can edit the file with the MAISA tool but it is not generated by the tool. Each he pattern directory contains five files used by the MAISA tool: prolog, description, statistics, metadata and figure file (naturally it can also contain other files connected to the pattern, for example sample source code and additional figures and diagrams).

Figure 1 shows the structure of the pattern library directory. Note that there is one group directory also for the metapatterns [Pre1995], because an instance of a metapattern can be construed as more than one design patterns presented by Gamma et. al. [Gam1995], depending on the semantics. Metapatterns themselves can also be recognized, without mapping them into design patterns. The following list explains how the pattern files are used.

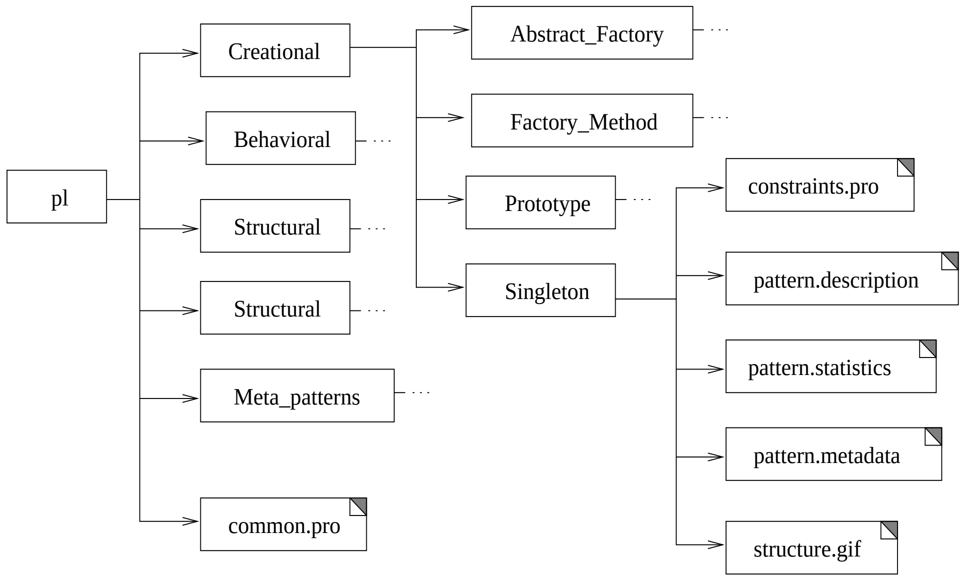

Figure 1: Structure of the pattern library directory.

constraints.pro The Prolog file that contains the structural requirements (or definition) of the pattern. The pattern recognition is based on these files. The file is not generated by the MAISA tool but the user can edit or set it with the tool.

pattern.description Text file that contains the same information of the pattern as the Design Patterns book [Gam1995] (see pages 7-9). The information of the file is for the human user, not the algorithms. The file is not used in the pattern recognition but only by the user interface of the MAISA tool. The file is generated by the MAISA tool on the basis of the data fed by the user.

pattern.statistics Text file that contains statistical information of the pattern: how many times it has been recognized, how many of these have been confirmed and how many of these have been false recognitions. The statistics file is automatically generated and updated by the MAISA tool and the user can reset the statistical information with the tool.

pattern.metadata Text file that contains metadata of the pattern that is used by pattern based estimation algorithms: LOC per pattern instance, LOC per client of the pattern and the performance factor of the pattern. The metadata file is generated by the MAISA tool on the basis of the data fed by the user.

structure.gif Figure file that describes the structure of the pattern. Optional. The file can not be edited or generated with the MAISA tool, but it can be set with the tool.

The formats of the text files are simple. The statistics and metadata files contain name-value pairs, one pair in a row. The name and the value are separated by a colon. The values are typically numeric, while the names are special keywords. The keyword sets can be easily extended in the future if needed. The keywords of the statistics file are:

- **WRONG** false recognitions
- **CONFIRMED** confirmed recognitions
- **TOTAL** all recognitions

A sample statistics file:

```
WRONG: 5
CONFIRMED: 12
TOTAL: 50
```

Metadata files are not yet in use, because the information contained in it will be used by the pattern based estimation algorithms, which are not yet in use in the MAISA tool. The exact types of the values are not yet fixed. The keywords of the metadata files are (the set can be extended in the future):

- **PERFORMANCE** performance factor
- **LOCPI** LOC per a pattern instance
- **LOCPC** LOC per a pattern client

The description file is divided into paragraphs, according to the documentation technique presented in the Design Patterns book [Gam1995]. The paragraphs are separated by an empty row and they can be in any order. The first line of the paragraph contains the mere paragraph name. The paragraphs are identified by their names, which are keywords. The structure of the paragraph depends on its name, as described in the following:

- **INTENT** What the pattern does. Any text except empty rows (the paragraph separator).
- **ALSO KNOWN AS** A comma-separated list of other names of the pattern.
- **MOTIVATION** What design problem the pattern solves. Any text except empty rows (the paragraph separator).
- **APPLICABILITY** A list of points where the pattern can be used. The first row of each list item begins with an asterisk.
- **PARTICIPANTS** A recursive list of the roles of the pattern. Each list item presents a role and contains a list of the tasks belonging to the role. The first row of a role item begins with an asterisk that is followed by the role name. The following rows of the role item begin with ">" that is followed by a task description.
- **COLLABORATIONS** A list of points where the elements of the pattern co-operate. The first row of each list item begins with an asterisk.

- **CONSEQUENCES** A list of consequences of using the pattern. The first row of each list item begins with an asterisk.
- **IMPLEMENTATION** Implementation specific tips on using the pattern. Any text except empty rows (the paragraph separator).
- **SAMPLE CODE** Sample code fragments, with possibly some ordinary text. Any text except empty rows (the paragraph separator).
- **KNOWN USES** Examples of the pattern found in real systems. Any text except empty rows (the paragraph separator).
- **RELATED PATTERNS** Pattern's relations to other patterns. Which other patterns should be used with this one. Any text except empty rows (the paragraph separator).

A sample description file:

```
INTENT
The pattern is used to...

ALSO KNOWN AS
Name 1, Name 2, Name 3

MOTIVATION
The pattern is useful, because...

APPLICABILITY
* Point 1
* Point 2
* Point 3

PARTICIPANTS
* Role A
> task 1
> task 2
* Role B
> task 3
> task 4
> task 5

COLLABORATIONS
* Point A
* Point B

CONSEQUENCES
* Point X
* Point Y
* Point Z

IMPLEMENTATION
The pattern can be implemented in this way...

SAMPLE CODE
Class MyClass{
...
```

```
}

KNOWN USES
The pattern has been used in...

RELATED PATTERNS
The pattern can be used with pattern X and...
```

6 Example

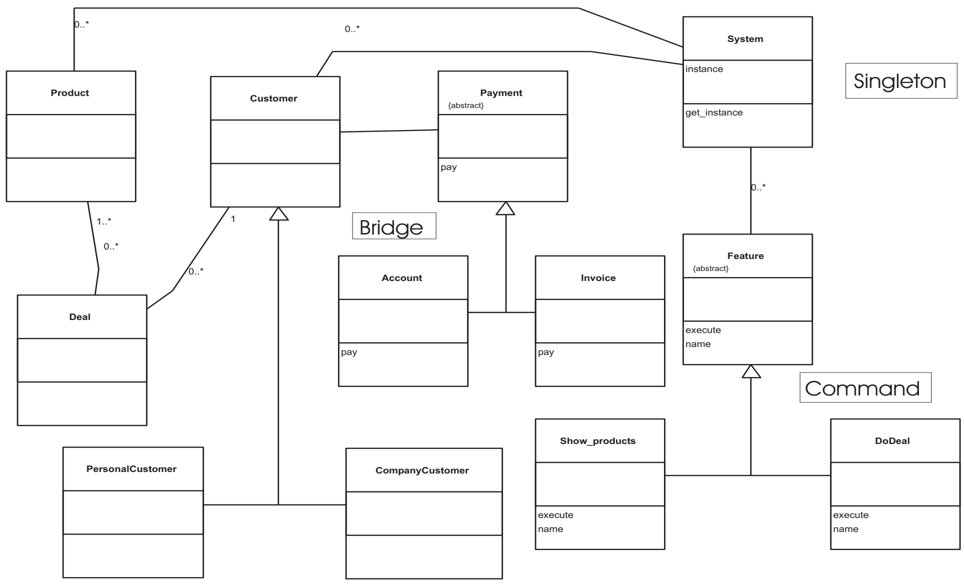

Figure 2: Shopping system class diagram.

Let us take an example of pattern detection. Figure 2 presents the class diagram of an Internet shopping system. The classes of the system are:

Customer (abstract class) The interface of the customers.

PersonalCustomer, inherits Customer Contains the data of personal customers.

CompanyCustomer, inherits Customer Contains the data of business customers.

Payment (abstract class) Defines the payment method. If the customer has an account, it is used. The default method is to send an invoice. The customer can create an account on-line and instantly use it.

Account, inherits Payment Defines the user account of the customer.

Invoice, inherits Payment Defines the invoice information.

System The main program.

Feature (abstract class) The interface of the plug-in features of the shopping system.

Show_products, inherits Feature A plug-in feature that allows the customer to browse the products available in the shopping system.

DoDeal, inherits Feature A plug-in feature for buying products. Creates a Deal object.

Product The products of the shopping system, containing product information such as price.

Deal Connects the customer with the products he has bought. A Deal instance is created for every buying event, a customer can have zero or more Deals, one or more Products belong to a Deal and one Product can concern zero or more Deals. The Customer association defines how the order is paid, and the Customer object defines its payment method.

The system contains three design patterns [Gam1995], also shown in Figure 2. First, System is a **Singleton** pattern. Only one System instance can be created and if several users need the services of the System class, the System instance can be shared. Second, the payment method of the Customer class is implemented as a **Bridge**. The customer can create and use a new account on-line, and new payment methods (e.g., a connection to a web bank) can be added without changing the Customer class. Customer has the role *Abstraction* of the Bridge pattern, PersonalCustomer and CompanyCustomer have the role *RefinedAbstraction*, Payment has the role *Implementor*, and Invoice and Account have the role *ConcreteImplementor*. Third, the plug-in features of the system are implemented as **Command** objects. As the shopping system and its customer services evolve, they need to be added and modified without changing the main program. System class has the role *Application* of the Command pattern, Feature has the role *Command* and Show_products and DoDeal have the role *OpenCommand*. Note that System also has the role *Singleton* of the Singleton pattern.

The Singleton instance is recognized directly. The other patterns are corresponding to the metapattern **1:1 Connection** [Pre1995], and the differences between them are semantical, not structural. Hence the Bridge and Command instances are recognized indirectly as instances of 1:1 Connection. In fact, the diagrams must be quite accurate to fulfil the needs of the pattern recognition algorithm. In this example, the class diagram in Figure 2 is accurate enough for a human reader. But in order to recognize 1:1 Connection instances, the class Customer has to be augmented with information about calling a method of the class Payment and similarly, the class System has to be augmented with information about calling a method of the class Feature.

7 Conclusions

We have discussed the technique for searching design patterns that is currently being implemented in MAISA. Major open issues are how well this technique performs with incomplete information and what kind of user interface the interaction procedure needs.

The representation formalism in MAISA is highly generic. Thus it is possible to try out different kinds of algorithms dealing with this problem or the extensions discussed earlier. Also, the mechanics of pattern recognition described here apply also to other types of diagrams. In fact, many open issues regarding class diagrams may have little or no effect in other diagrams. Most problems are caused by the use of structures that have a high level of abstraction.

We have a few ideas about evolving the pattern library and its user interface. Pattern groups could have own descriptions, explaining the nature of the group. HTML could be used as the format of the pattern.description files, to allow the pattern descriptions to be read with web browsers. All the textual pattern files (pattern.description, pattern.statistics and pattern.metadata) could be unified into one file. In principle, also the figure file could be unified with the text files into a single file but then the pattern file would not be textual any more and HTML could not be used as the file format.

References

[FrW1992] Eugene Freuder, Richard Wallace: *Partial constraint satisfaction*, Artificial Intelligence 58,1-3(1992), 21-70.

[Gam1995] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley 1995.

[Kum1992] Vipin Kumar: *Algorithms for constraint-satisfaction problems.* AI Magazine 13,1(1992), 32-44.

[Mac1977] Alan Mackworth: *Consistency in network of relations.* Artificial Intelligence 8,1(1977), 99-118.

[Mac1992] Alan Mackworth: *The logic of constraint satisfaction*, Artificial Intelligence 58,1-3(1992), 3-20.

[MaF1985] Alan Mackworth, Eugene Freuder: *The Complexity of Some Polynomial Network-Consistency Algorithms for Constraint Satisfaction Problems*, Artificial Intelligence 25,1(1985), 65-74.

[Mis1997] Pauli Misikangas: *Suunnittelumallien automaattinen tunnistaminen olio-ohjelmista*, Master's Thesis, University of Helsinki, Department of computer science, C-1998-1, 1997.

[MoH1986] Roger Mohr, Thomas Henderson: *Arc and path consistency revisited*, Artificial Intelligence, 28(1986), 225-233.

[Pre1995] Wolfgang Pree: *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.