Source code processing using expert systems

Damir Kirasić

University of Zagreb
Faculty of Electrical Engineering and Computing
Zagreb, Croatia

ABSTRACT

The problem of efficient source code processing – analysis, transformation and synthesis – is addressed. Programs written in object oriented programming language (e.g. Java) are transformed into the corresponding expert system facts and then processed by means of expert system rules and functions. After processing, these facts are translated back to the target source code. In case of program synthesis, appropriate expert system facts have to be declared before source code generation. To verify this paradigm a programming framework is developed. Although the implementation of the framework relies on the Java Expert System Shell and Java programming language, the overall concept is not tied to any particular expert system or programming language. Three example applications are described: program localization, try-catch generation and Enterprise JavaBeans creation.

KEYWORDS: Java, Jess, expert systems, knowledge based programming, program transformation

1 INTRODUCTION

Existing program development environments offer modest support for the tasks that are inevitable part of any serious software development process:

- Program code analysis/comprehension/reengineering
- Generation of repeating (or mechanical) parts of the program project
- Various program transformations (e.g. internationalization)
- Program synthesis.

These problems cover extremely vast areas of the software engineering field and a number of various approaches to solve them is even vaster. The difficulties in those areas are mainly caused by the fact that the program source code is kept as a plain, flat text file. Any kind of source code processing requires complex software. Input source code has to be parsed or preprocessed in some way in order to, at least, distinguish programming elements. Keeping source code as a formatted, structured entity seems to be much more convenient. Appropriate structure of the

programming elements could enable capabilities that are difficult to achieve with flat text representations.

In this paper a unified approach to above diverse problems is proposed. Almost all kinds of source code processing could be performed, or at least could be supported, by means of expert system – ES for short. As a first step, source code written in imperative or object oriented programming languages (e.g. Java, C/C++), could be translated into expert system facts. These facts are completely structured and any kind of processing of these facts can be done in the ES environment. Finally, facts can be transformed (e.g. translated) back into the target programming language. Figure 1 shows the overall architecture of the proposed paradigm.

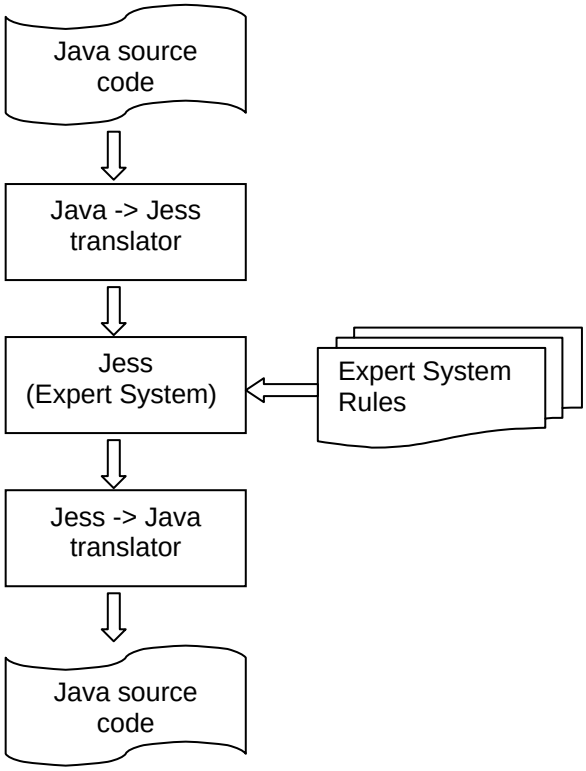

Figure 1: Source code processing using expert system

Expert system facts can be viewed as a program specification. In case of program synthesis, construction of the ES facts could be done as a first step and transformation of these facts into source code as a final step. It is important to notice that the main concepts advocated in this work are completely generic and independent of any particular expert system or programming language.

The aim of this paper is to present concrete implementation of the above ideas and to describe some experiences with it. The implementation described in this paper uses Java Expert System Shell - Jess (Friedman-Hill, 1997) as expert system and the Java programming language (Gosling, 2000). It is still in experimental phase and currently is envisioned as a programming framework that is used by other programmers. The framework can be used as a starting point for some specific applications such as source code analyzing or various transformations of the input source code.

The above ideas are not completely isolated from the current work in the field of automated software engineering but as far as author knows, there is no similar research project.

Extensible Markup Language – XML (Bray, 2000) is a fast growing technology with many supplementary tools available and at first sight it seems to be perfectly suitable for our purpose. There are at least two projects, which propose Java source code translation into XML (Clifton, 2001) (Badros, 2000). After translating source code into XML document, we could process that document by means of some XML-based program. Unfortunately, XML-based programs are (still) too complicated. Moreover, any new requirements on the type of processing would require rather complex and newly created program. For example, suppose we already have complete program that do one kind of input program transformation. In order to do new kind of transformation one would need completely new program. Because it is too difficult to foresee what programmers will need, we cannot provide that in advance. Programmers need flexible tools, which will provide simple and efficient means of adding new functionality when they are needed.

In spite of XML advantages, the expert system approach seems to be more natural solution. General-purpose expert systems are already well-established technology. ES environment provides simple, flexible and effective means of facts processing. Features such as inferring or addition of newly created facts are fundamental part of expert systems. If we could keep source code as ES facts then we could do any processing much easier (compared to XML approach). This paper describes the "expert system approach" approach.

In the program analysis field there are, as far as author knows, only few attempts to transform imperative language source code into ES facts. (Finkbine, 1994). Most of the work in this area is oriented toward expressing imperative language programs by means of declarative languages. See for example (Peralta, 1997).

Program transformation and program synthesis field seams to be more attractive for various project researches. Most of the work in this field is based on the mathematical theorem proving. See for example survey of deductive program synthesis in the paper from Manna and Waldinger (Manna, 1992). They combine first-order logic with quantifiers and connectives, and mathematical-induction principle. Another program synthesis project is conducted by Lowry, Van Baalen and Roach at NASA (Lowry, 1997)(Van Baalen, 1998). Their work is focused on domain-independent architecture that is specialized through declarative domain theory. As a final result they have "application-generator generator" that can generate Fortran applications composed of function calls and prewritten Fortran functions.

There is yet another successful attempt to combine declarative logic programming and source code generating described in the work of Kris De Volder (De Volder, 2000). In his work modified Prolog programming language is used to describe design patterns (Gamma, 1995). Prolog programs are parameterized and can be used as a basis for the "declarative code generator" system. In addition, these code generators can be combined to produce combinations of design patterns.

Following sections will elaborate main ideas presented in the introduction section. After brief overview of the Jess in section 2, the idea of representing program elements as expert system facts is presented in section 3. The following sections will address Java to Jess translation and Jess to Java translation. Three example applications are described in section 6.

2 JESS

Jess is a LISP like, command oriented rule engine based on the Rete algorithm (Forgy, 1982). It inherits most of the features from the CLIPS expert system shell (Giarratano, 1998). Unlike CLIPS Jess is completely written in the Java programming language.

The Jess environment can execute Jess scripts that basically consist of *facts, functions, rules* and *queries*. Facts are defined as *lists* which are asserted with **`assert`** or **`deffacts`** functions. Facts have two forms: *unordered* and *ordered*. Ordered fact would be something like

```
(import java.util.ArrayList hr.fer.pf)
```

Unordered facts has names for variables:

```
(class (name Test) (fields a b c)).
```

The *template* for the unordered fact must be defined before unordered facts are used. Functions look like lists but are defined with **`deffunction`**. Users can define their own functions by means of other Jess functions or by writing them in Java. Rules consist of if-part (or LHS) and then-part (or RHS). The "=>" stands between if-part and then-part. All the lists in the if-part are interpreted as facts and all the lists in the then-part are interpreted as function calls. If there are more then one fact in the if-part they are implicitly connected with logical AND. Queries are defined by means of **`defquery`** function and consist only of if-part. Queries return an iterator over the found facts.

Jess can be (but need not to be) used closely connected to Java. With Jess one can write complete Java applications or applets, create Java objects and call Java methods without any Java programming. For example, the following Jess script creates a **`TreeMap`** object, puts two key/value pairs in it and gets one value for the given key:

```
Jess> (bind ?aMap (new  Java.util.TreeMap)
Jess> (call ?aMap put "word1" "value1")
Jess> (call ?aMap put "word2" "value2")
Jess> (call ?aMap get "word2")
"value2"          ; the output of the "get" method
```

Java and Jess can collaborate in many ways:

- Most of the user code is written in the Java language from where Jess engine (also called Rete engine) is created and called. Jess is embedded in Java program.
- The user code is written entirely as a Jess script, the Java is called implicitly as in previous example.
- Most of the user code is written as Jess scripts from where user-written Java functions are called.

Let us see a self-explanatory example for the first scenario.

```
import jess.*;

public class TestJess
{
    public static void main(String[] args)
```

```
    {
        Rete r = new Rete();     // create new Rete engine
                                 // . . . create Jess command here
        r.executeCommand("jess-command")
                                 //  Do some more useful things here
        r.reset();               // reset the engine
        r.run();                 // run the Jess engine
    }
}
```

Jess does not support nested lists. In other words, a list cannot contain another list. The nesting capability is crucial (for XML for example) and is important for program representation because programs are by default nested structures. That fact influenced this work and the transformation of Java code into Jess facts as well as the structure of the Jess representation of the Java code. Following section describes this more thoroughly.

3 PROGRAMS AS FACTS

Programs written in C/C++, Java or other languages can be translated into expert system facts. This section will describe the possible goals of such translation. Although the examples and the discussion will be focused on the Java programs and Jess, the principles and concepts should be considered as universal.

The following example shows simple Java class and its equivalent expert system facts.

```
class Test {
    private int a;
    public double b;
    public String c;

    public int get_a() {return a;}
}
```

The representation of the above class could look like the following facts:

```
(class (name Test)(fields a b c) (methods get_a))
(field (name a) (class Test)(access private)(type int))
(field (name b) (class Test) (access public) (type double))
(field (name c) (class Test) (access public) (type String))
(method (name get_a) (class Test) (access public)
                (type int) (body "return a;"))
```

In order to define such facts, programmer must define Jess templates first. For example, field template could be defined as:

```
(deftemplate field (slot name)(slot class)(slot access)(slot type))
```

The templates will define the name and the structure of each fact we want to use. In this example we use simplified and incomplete templates and very simplified Java program in order to illustrate main ideas only. Real examples are more complicated.

The real implementation has defined two supplementary ID slots for every fact. These overhead slots are needed because Jess does not support nesting, as it is already mentioned. Moreover they

are needed because the sequence of the facts does not necessarily follow the program sequence. To enable proper fact sequence two additional slots are introduced. The first ID is unique, logical identification of the fact that can be used as a reference to thatfact. The second ID is a parent ID (pid) – the identification of the parent fact. These IDs enable traversing through facts in both directions top-down and bottom-up. For example, the following C/C++/Java assignment statement:

```
x = 5;
```

would be translated into the following facts:

```
(assign (id 101)(pid 100)(left 102)(operator "=")(right 103))
(simple-name (id 102)(pid 101)(name x))
(integer-literal (id 103)(pid 100)(value 5))
```

There are a couple of important points regarding expert system facts that represent some program that are worth to notice here:

- The facts have structure: this is not a plain text any more.
- We can gain access to all program elements (type of variables, name of methods, accessibility of the fields of the class, etc.)
- Any kind of processing is much easier (even if we do not use expert system)
- We already have a means of processing facts: functions, rules and queries
- If new functionality (type of processing) is needed it can be quickly added

Processing and analysis of such program facts can be done at various levels. At logical level we could, for example, find if a local variable is defined but not used in a method. At semantic level we could infer that the Test class (from the above example) is only a repository for a few variables. And finally we could look at facts from the other side. Facts can be viewed as the specification for the source code.

Jess facts look nice but they are useless without rules and functions that can process them. Let us see one simple rule, which will print out all public fields from every class it finds:

```
(defrule print-public-fields
    (field (name ?field-name) (class ?class-name) (access public))
    =>
    (printout t ?class-name " " ?field-name crlf)
)
```

Now, if we run Jess engine it will find all the fields in the classes that have access defined as **`public`** and print the names of the classes and the names of the fields to the standard output. The above rule has *if- part* and *then-part*. We can read this rule as: if there is a fact of type **`field`** with **`public`** access **`(field (name ...) (access public))`** then do the function in the then-part: **`(printout ...).`** When the field fact is found its slots will be stored in the corresponding *variables*. For example, class name will be saved in the **`?class-name`** variable. Even from this simple example we can see how simple it is to define such operations that are difficult to achieve with standard tools and flat text representations.

The above example can be extended and slightly complicated rule can be defined. New rule named **make-get-methods** will: (1) find all the fields which have no **get()** method, (2) rewrite that fields as private and (3) construct appropriate **get()** methods:

```
(defrule make-get-methods
    ?fieldId <- (field (name ?fn) (class ?c) (access ?a) (type ?t))
    (not (method (name ?mn&=(sym-cat get- ?fn)) (class ?c)
        (type ?t)(access public) ))
    =>
    (retract ?fieldId)
    (assert (field (name ?fn) (class ?c) (access private) (type ?t)))
    (assert (method (name (sym-cat get_ ?fn)) (class ?c)
        (type ?t) (access public) (body (str-cat "return " ?fn ";")) ))
)
```

The only new thing in the above example is the **"<-"** operator that is used to store the field fact reference into the variable **?fieldId**. This variable is used to retract that fact later.

The type of source code processing just described, and the similar ones not described, could be part of some integrated development environment. We could also imagine on the fly transformation i.e. the source code is transformed into facts while typing (instead of keyword coloring for example). This integration with other development tools is (still) out of scope of this work.

In order to write elegant rules that will process facts, obviously, there must be a program that will translate Java source code into the expert system facts first. After processing, facts have to be translated back into the source code. The list of facts will be bigger and less readable than the original Java code, but some expert system will deal with it – not human programmer.

4 TRANSFORMING JAVA PROGRAM INTO THE JESS FACTS

There are many possible methods of translating Java programs into Jess facts. The problem can be viewed as a much broader one – we have to translate source code written in one programming language into another form preserving all the features of the original program. This "another form" can be, but need not be, another programming language. The reverse process – translating "another" form back into the original one has to be defined also.

As already mentioned, the aim of this translation is to get input program into a more structured shape, a shape that can be easily processed.

A couple of possible solutions for the Java to Jess translation can be envisioned:

- Write the translator from scratch, possibly using existing tools (such as JavaCC or some lex/yacc variants).
- Use existing open source Java compiler and add new features to it.
- Use the existing Java to XML translator (Badros, 2000) and then translate XML to Jess (Leff, 2000).

In this work second approach from the above list is chosen mainly because of the availability of the excellent open source Java compiler named Jikes (see reference list for the Jikes homepage). Jikes is an IBM Java compiler written in C++. This is a fast Java compiler that has, among other things, unparse capability. That is, it can take the generated parser tree as input and produce source code from it. Instead of generating Java code, new unparse methods are written that will generate Jess facts. This approach was inspired by the JavaML project (Badros, 200) but it is the "clean table" implementation.

The very translation to Jess facts is done along with normal Java compilation. One has to use new Jikes option "**+uj**" to get facts for the input Java file:

```
jikes +uj Test.java
```

The above command would create new file named **Test.java.jess** that would contain the appropriate Jess facts. Other features, such as dump of the parse tree and debug feature of the unparser, helped a lot in the development of the new unparse methods. (All this features are available only if the whole compiler is compiled with debug option).

It is interesting to note that there already exists a complete solution for the translation problem. That solution would belong to the 3-th class of solutions from the list at the beginning of this section. Java can be translated into XML first and then XML could be translated to Jess facts. Tools are available for both of these translations. After initial testing of this approach it became clear that generated code was too big and too complicated. Although initial experience with this approach where unsatisfactory, this kind of translation could be extremely important if we want to have portability among different expert systems. XML could be used as a universal data format for different expert systems.

The output of the translator is a text file consisting of the Jess facts. Following points describe main ideas used in this implementation:

- For every type of parser node a fact is defined. For example, there are facts for: class, interface, field, method, string literal, etc. The syntax of the facts and the very names of the facts are defined in Jess templates. The unparse methods follow that syntax and names. Therefore, names and fact syntax are embodied into the translator (which we feel, could be improved).

- Due to Jess restrictions facts are "flat" i.e. there is no nesting. In another words, facts cannot contain other facts. This looks a little bit strange but facts are meant to be used by other programs (e.g. Jess rules and functions) not by programmers. Physical sequence of facts is unimportant. The exact sequence of facts is established by means of integer id and pid slots.

- All facts are defined as unordered facts. Every fact has a name (head) and two obligatory slots **(slot id)** and **(slot pid)**. **id** is used as a unique identifier of the fact and **pid** is used as a parent id – that is, it is id of the parent fact. For example if class fact has id 3 than the field fact that is part of that class will have pid set to 3.

The translation to Jess facts is quick. One will not even notice that some extra work is done along with standard Java to bytecode translation. When the number of input Java source code lines is compared with the number of lines (e.g. number of facts) in the Jess file, there will certainly be

more facts than input source lines. On the average there are 3 times more lines in the fact file then in the original Java source file.

5 TRANSFORMING JESS FACTS INTO JAVA PROGRAM

As in the case of Java to Jess translation, there can be more than one approach to Jess to Java translation also:

- Load facts into embeded Rete engine first, and than start to query engine for the facts and translate them to the Java source. Use program written in Java.
- Write completely (expert system) independent C/C++, Java or some other language based translator.
- Use defined templates as input syntax and generate translator for the known Java syntax.

The first approach is adopted because the processing of facts must be done in the Rete engine anyhow. At least in one phase of processing we have all the facts loaded into the Rete engine. The Rete engine itself should be embeded into the larger Java program for most applications.

The development of the translator was relatively simple because Jess facts already represent *parsed* Java program. The translator starts with "compilation unit" and descends down the tree to classes, interfaces and methods. It probably finishes with "simple names" and string literals. Every fact template has corresponding method that does its part of the job.

During the development of the Jess to Java translator, after the first version was finished, it appeared that the translation was terribly slow. After short analyzes it became clear that it was due to the slow **`getFactByLogicalId(int id)`** method that was used frequently. (The purpose of IDs is explained in section 3). This method iterated through ALL the facts stored in the Jess (Rete) engine and tried to match every fact with input id parameter. The iterating, that was done by the Java iterator, was slow and that was the main cause of the translator inefficiency. For example, 20 seconds were needed to translate 4500 facts into approximately 1500 lines of Java code.

In order to improve the speed of the translation, a new version (named **`ERete`**) of the existing Rete engine was defined. The **`ERete`** extends **`Rete`** class and adds new Java **`HashMap`** to store Jess facts. The id of the fact is used as a key for the **`HashMap`** .

```
class ERete extends Rete
{
    private HashMap  m_idTable;     // supplementary store

    // override original assert, retract, ... method

    public Fact assert(Fact f, Context cx)throws JessException
    {
        int id = f.getSlotValue("id").intValue(cx);
        m_idTable.put(new Integer(id), f);
        super.assert(f, cx);
    }
    . . .
```

```
    public Fact getFactById(int id) {
        return (Fact) m_idTable.get(new Integer(id));
    }
}
```

New methods **`assert, assertString`** and **`retract`** will save or retract facts from the supplementary **`HashMap`** first, and than forward the method call to the original Rete method. The **`findFactById`** method will find the required fact very quickly because the key for the **`HashMap`** is **`id`**. The overall result was that the translation speed was accelerated for about 9-10 times.

6 EXAMPLE APPLICATIONS

In order to verify the ideas and the implementation presented in the previous sections three simple application programs are written: Java program localization, try/catch generation and EJB creation (Matena, 2001). Although these example applications are meant to be used in different situations they all share common features:

- The example application is written as a Java program that has embeded Jess engine.
- Input is Java source code.
- Three basic steps are used: 1. Translate input Java program into Jess facts 2. Process facts 3. Translate Jess facts back to Java.

Following subsections will give brief overview of the example applications.

Program localization

Usually, programmers start with a version of program written for only one language or region. If, after some time, an application has to be adapted to another language, it has to be rewritten. The process of translation program to function in a specific locale is called localization.

This is exactly what this example application does. It takes Java source code, written for the specific language/region, as input, and generates a copy of input program for another language/region. The translation of the parts of the input program is done interactively – the programmer is asked to translate every translatable string literal into the new language unless the translation for the current word or phrase is already translated.

Internally, input source code is transformed into the Jess facts and all processing is done with the Jess facts. After replacing translated strings with new ones, Jess facts are transformed back into Java providing a copy of the input Java program for some other language.

At the core of this process is a simple Jess query that finds all **`string-literal`** facts:

```
(defquery find-string-literals
(string-literal (id ?id)(value ?value)(translatable TRUE))
```

String literals are printed to the screen and programmer is prompted for the translation. The translation is written into the "value" of the newly created string-literal fact that has the same "id"

as the original one. The old fact is then retracted and the new one is asserted. The "translatable" slot of the string literal template is meant to allow non-translatable string literals.

The whole example program is less then 300 lines of code and although it does not implement i18n standard for the Java applications, it still can be useful.

The process of designing application that is easily adoptable to various language or region is called internationalization. (The word "internationalization" is sometimes abbreviated as i18n because there are 18 letters between "i" and "n"). The full i18n application based on expert system processing is underway.

Try-catch factory

Second example takes as input Java source code that has no try-catch constructs but it calls methods and constructors that can throw exceptions. Such Java code cannot be compiled into bytecode but it can be transformed into Jess facts. The aim of the application is to generate missing try-catch statements. The main work is done in a few steps:

- All method and/or constructor calls are recognized by means of a trivial Jess query.
- *Exception knowledge base* is consulted to see if current method/constructor call can throw an exception and to determine which exception could it be.
- Try statement and appropriate catch statements are constructed and added.

The exception knowledge base is nothing else but the set of Jess facts that define methods and constructors in the format already defined by Jess templates. For example, a part of such knowledge base could be:

```
(constructor (name FileWriter)...(throws NoSuchFileException))
(method (name readLine)... (throws IOException))
```

This kind of try-catch generation will guarantee that the try-catch processing will be uniformly specified throughout whole software project. In case of special situations where some specific action is required, the catch part can be parameterized.

As you have probably already concluded, this kind of application can be used by lazy programmers that do not like to deal with try-catch or in case we want to unify try-catch statements throughout large software project.

Enterprise JavaBeans factory

Enterprise JavaBeans (Matena, 2001) are Java software components usually used in distributed business applications. While developing EJB based applications, testing can be complicated since EJB are executed (deployed) inside EJB containers and usually are surrounded with complicated network/database environments. It would be much more convenient that, at the start of the EJB-based project, it is possible to make a prototype application with *ordinary* Java classes. The main ideas could be tested at the early phase of project and build full-fledged EJB application later.

Using expert system framework, EJB factory application is written. EJB factory application can build EJB from non-EJB Java classes. It takes source code for some ordinary Java class as input and

produces all that is needed for EJB: Home interface, Remote interface and Bean class. The main job is done by: (1) finding the public methods and/or constructors in ordinary Java class and (2) generating appropriate interfaces. During this process some external template-code is used for Home and Remote interfaces. As in other example applications, all processing is done on Jess facts and finally facts are transformed back to the Java code.

The application can preserve programmers from writing mechanical, repeating parts of the EJB based software, thus it can improve programmers efficiency and add some uniformity to EJB project. This is the longest and most complicated example application presented here but all in all its basic implementation requires mere 700 lines of Java code. The whole job of EJB creation can be easily done by means of Java reflection also, but the approach presented here seems to be more flexible.

7 CONCLUSIONS AND FURTHER WORK

The key idea of having programming language elements translated into expert system facts seems to be healthy and fruitful. Once you have source code transformed into expert system facts almost any kind of analysis, any kind of modification is possible.

During the work on the example applications it became clear that some tasks (especially code generation) are tedious and too complicated. A need for some utility methods or some high-level libraries became obvious. Generally, some kind of "framework API" should be defined in order to clearly distinguish what is part of the framework and what is user code.

Among other things, during the example applications development it became clear that we need some kind of *partial translation* of Java code into the Jess facts. Not only complete programs should be translatable but also tiny pieces of incomplete code should be translated into the Java facts. It would be convenient to call translation without first storing Java source into files. To illustrate this let us se one simple example. Suppose we want to add following two lines of Java code into existing Jess facts:

```
catch (IOexception ioe) {
      ioe.printStackTrace(System.err); }
```

In order to accomplish that, one needs about 30 lines of user code. (Remember that these two lines of Java code are presented as approximately 6-7 facts and that for creating each fact one needs about 5-6 lines of Java code).

These yet-to-be written libraries should give us features for higher level programming, like facts constructing. We can even think of whole hierarchy of libraries that help us during the whole process of software development cycle.

The speed of Jess to Java translation is significantly improved (see Section 5). As already mentioned, both Java to Jess and Jess to Java translation should be determined only by the Jess templates. The details of translation like the names of the facts and slots should not be spread throughout the translator code.

Although the overall paradigm seems to be most suitable for the automated software analysis tasks, this area is almost not touched at all. The analyzing could be done at various levels. The lowest level could be just above the compiler warnings. Various naming or style consistency

conventions could be checked. At some higher level, inefficient parts of the input program could be detected and reported. Yet at some higher level, semantic features of the program could be grasped.

By means of existing implementation we could do something like inductive knowledge acquisition also. This means that we could try to extract algorithms or programming and design patterns at various levels. See for example design pattern creation with declarative logic programs (De Volder, 2000). The "only" things we have to add to the existing framework are expert system rules that would know how to extract desired features.

In the program transformation area we can envision various rewriting jobs. As already mentioned in the previous paragraphs (e.g. naming or style consistency checking, inefficiency), some future application could not only report these features but could rewrite some input program according to needed propositions.

In the program generation area the Jess facts could be used as a detailed program specification. These facts could be the end of some automated program synthesis process. In this process we would like to concentrate on *what* our future program has to do instead of *how* things should be done. These reflections inevitable lead us to the formal or non-formal program specifications and similar areas. For the overview of the formal specification paradigms and languages see (Lamsweerde, 2000).

And for the end of this paper, it is quite reasonable to think of completely new and highly specialized expert system that could serve source code processing needs more cleanly and more efficiently then the general-purpose expert system shells.

ACKNOWLEDGEMENTS

I thank Denis Gračanin, Mario Žagar, Damir Pehar and Ivica Čardić for useful suggestions regarding this work.

REFERENCES

Badros, G.J. 2000. JavaML: A Markup Language for Java Source Code. 9th International World Wide Web Conference, Amsterdam, May 15 - 19, 2000, The Netherlands

CLIPS, Homepage, Documentation: http://www.ghg.net/clips/CLIPS.html

De Volder, K. 2000. Implementing Design Patterns as Declarative Code Generators, University of Brittish Columbia, The department of Computer Science, Unpublished paper

Clifton, C. Guo L. Parzyszek K. 2001. A JDE in XML, Department of Computer Science, Iowa State University, Proposal for project, Available online via http://www.cs.iastate.edu/~cclifton/xmljde/proj-proposal/proj-proposal.html

EJB, Enterprise Java Beans 2.0 Specification: Available online via http://java.sun.com/products/ejb/2.0.html, [Accesed February 10, 2002]

Finkbine, R. B. 1994. A CLIPS template system for program understanding, Third Conference on CLIPS Proceedings (Electronic Version) *September 12–14, 1994. Lyndon B. Johnson Space Center*

Forgy, C.L. 1982. Rete: A Fast Algorithm for the Many Pattern/ Many Object Pattern Match Problem, Artificial Intelligence 19 (1982), 17-37

Friedman-Hill, E. 1997. *Jess, the Java Expert System Shell.* Sandia National Laboratories, Livermore, CA, 1997. http://herzberg.ca.sandia.gov/.

Gamma, E. Helm, R. Johnson, R. Vlissides J., 1995, Design Patterns, Addison Wesley, Reading, MA, 1995.

Giarratano, J. C. 1998. Expert systems: Principles and Programming, Brooks/Cole Pub. Co.

Gosling, J. Joy, B. Steele, G. L. and Bracha G. 2000, The Java Language Specification, Second Edition, Addison-Wesley, ISBN 0-201-31008-2

Jikes, IBM Open Source Java compiler, Homepage: http://oss.software.ibm.com/developerworks/ projects/jikes/ [accesed January 2002]

Leff, L. 2000. XML/Expert System Integration with Contracts and Litigation, http://ecitizen.mit.edu/ecap3.html

Lowry, M.R., Van Baalen, J., 1997, META_AMPHION: Synthesis of Efficient Doman-Specific Program Synthesis Systems, Automated Software Engineering 4, Kluwer Academic Publishers, pp. 199-241

Manna, Z., Waldinger, R., 1992., Fundamentals of Deductive Program Synthesis, IEEE Transactions on software Engineering, Vol. 18, No. 8, August 1992, pp 674-705

Matena, V. and Stearns B. 2001, Applying Enterprise JavaBeans, Addison-Wesley

Peralta, J. C., Gallagher J. P. 1997, Toward Semantics-based Partial Evolution of Imperative Programs, Technical Report CSTR-97-003, Department of Computer Science, University of Bristol, April 1997.

Van Baalen J., Roach, S., 1998. Using Decision Procedures to Accelerate Domain-Specific Deductive Synthesis Systems, LOPSTR '98 Proc. of the 8th intl. workshop on logic program synthesus and transformation 1998, Springer-Verlag Berlin Heidelberg, 1998

Van Lamsweerde, A., 2000 Formal Specification: a Roadmap, In the future of Software Engineering, A. Finkelstein (ed.), ACM Press, 2000.

Bray, T. Paoli, J. Sperberg-McQueen, C. M. 2000. Extensible Markup Language (XML), (Second Edition) , W3C Recommendation 6 October 2000, Available online via http://www.w3.org/TR/REC-xml [Accesed February 10, 2002]