МАТЕМАТИЧКИ ВЕСНИК
45 (1993), 23–27

UDC 519.767
оригинални научни рад
research paper

AUTOMATIC MODIFICATION OF SMALLTALK METHODS

R. Protić and D. Tošić

Abstract. This article focuses on the issue of making a kind of preprocessor (called Automatic Modifier of Smalltalk Methods — AMSM) for the programming language Smalltalk. A general algorithm, based on the automatic modification of Smalltalk methods, is proposed. All modifications of related methods are realized in Smalltalk itself. The main parts of method-code and relevant classes are presented. Also, some characteristics (and possible) applications of AMSM are exposed.

1. Introduction

The Smalltalk system is the ancestor of all object-oriented systems and provides one of the best implementation of the object-oriented paradigm. Smalltalk is a programming language and highly interactive programming environment. It was originally developed for the Xerox family of workstations, but now it is implemented on a variety type of computers. There are many books ([4], [5], ...) and articles ([1], [2], [7], ...) describing the features of Smalltalk system. We will mention here some of the features of Smalltalk significant for our work.

Smalltalk is an extendable language designed to make it easier to program and to build convenient programming environment. The characteristics of Smalltalk system are: the visual impact of bitmapped graphics, highly interactive user interface and increased flexibility in terms of user programmability.

For one familiar with the procedural style of programming, we are going to shortly explain some of frequently used terms from object-oriented programming in the procedural terminology.

Object is a package of data and description of its manipulation.

Class describes an object and the methods that it understands. The corresponding term in procedural terminology is *type*.

Method is the *procedure*-like desription of sequence of actions for transformation, examination and communication with the objects.

Message specifies one of an object's manipulations. It corresponds to the *procedure* or *function call*.

More detail description of relevant terms could be find in [4] and [6].

Keywords and phrases: Smalltalk, automatic modification, method, environment.

2. Why automatic modification of methods?

As in other programming languages there is a need for software tools to change and debug a running application. The writing code in Smalltalk is rather different than writing code in any other language because of specific organization. (The source code of methods is integrated into the virtual image (see [4] and [6]) and could not be modified like the other ASCII-files.) Of course, a programmer may use the Debugger and Inspector in Smalltalk to test and debug the code. So, one could inspect method by method to make all requested changes. If one should change ten or hundred methods on the previous way, it might be very tedious.

The most frequent changes should be done in the same way. For example, one should change the same message in many methods. Our goal was to make a general tool (AMSM) for an automatic modification of the same part of the code in many methods.

3. AMSM implementation in Smalltalk/V

The automatic modification involves four steps:

1. Taking out a source code of methods from Smalltalk hierarchy.
2. Transforming a source code into target code.
3. Making a standardized stream.
4. Filling the obtained stream into Smalltalk hierarchy.

Figure 1 illustrates the functioning of AMSM.

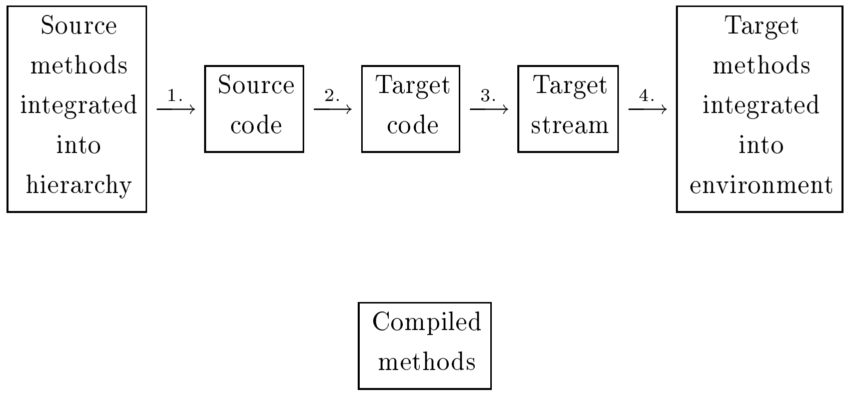
Source methods integrated into hierarchy
1.
Source code
2.
Target code
3.
Target stream
4.
Target methods integrated into environment
Compiled methods

Fig. 1

Smalltalk is a language enriched with a lot of classes and methods, well-studied for efficient realization of described process.

The core of AMSM is the method presented in Fig. 2. This method might be slightly modified depending on a real application.

```
! Behavior method !

modifClass: anObject
     "All methods of the receiver class are
modified according to the structure of anObject
and reintegrated into Smalltalk hierarchy."
     | str meth aux1 aux2 has |
meth := self selectors asArray.
str := ReadWriteStream on: String new.
str beginingOfStream: self.
has := false.
meth do: [ :met |
               aux1 := self sourceCodeAt: met.
               aux2 := aux1 transform: anObject.
               aux2 ~= nil
                  ifTrue: [ str nextChunkPut: aux2 ;
                                 cr.
                            has := true
                          ]
          ].
str endOfStream.
has ifTrue: [ str fileIn ]
```

Fig. 2

The proposed method calls following (new, nonstandard) methods:

`beginingOfStream`: prepares the first part of standardized stream;

`transform`: transforms a source code of method into target code;

`endOfStream`: marks the end of standardized stream.

The method `transform` depends on a kind of change that should be done. This method might be rather complicated. Its realization is mainly based on the classical parsing techniques.

4. Applications

AMSM is a general tool that might be used in the different applications. For example, it is very suitable for the further development of applications described in [3], [9], etc.

We applied AMSM to prepare an arbitrary PROLOG/V program for efficient interfacing with the external database. PROLOG/V is completely integrated into Smalltalk/V environment and the organization of PROLOG predicates corresponds Smalltalk methods. Because of that, AMSM was convinient to modify both PROLOG predicates and Smalltalk methods. To realize the method `transform` (Fig. 3) in this case, we created a new class `ParseStream` which contains the following methods: `isLogical:`, `modifyPredicate:`, `writeElementConstruct:`, `writeComment:`, `writeArgument:`, `writeString:`, `rewriteUntilBlank:`, `peekNonBlank:`.

```
! String method !

transform: aSymbol
"Checks if the receiver is a PROLOG method.
 If it is, all apperences of aSymbol( ... ) are
 changed by interpret( aSymbol( ... ))."
  | input lengthInput output aux length end mark ind |
lengthInput := ( input := ReadStream on: self ) size.
output := ParseStream on: String new.
length := (aux := aSymbol asString) size.
(output isLogical: input) ifFalse: [^nil].
ind := true.
end := lengthInput - length + 1.
[input position < end]
     whileTrue: [
        input peek ~= (aux at: 1)
            ifTrue: [ output writeElementConstruct: input ]
            ifFalse: [
                 mark := input position.
                 ((input nextWord) = aux and:
                    [(input peekNonBlank) == $( and:
                        [mark = 0 or:
                           [(input position: mark - 1 )
                              peek isAlphaNumeric not]]])
                 ifTrue:   [  input position: mark.
                              output modifyPredicate: input.
                              ind := false
                           ]
                 ifFalse:  [  input position: mark.
                              output nextPut: input next
                           ]
                 ]
            ].
ind ifTrue: [^nil].
[input atEnd]
     whileFalse: [output nextPut: input next].
^output contents ! !
```

Fig. 3

The methods: `writeComment:`, `writeArgument:`, `wrireString:` and `rewriteUntilBlank:` are used in the method `writeElementConstruct`.

5. Conclusion

AMSM adds to the standard Smalltalk environment new possibilities to manipulate methods and enhancements to existing tools.

The programming techniques to replace software components on the fly in a running program are described in [8]. The ability to modify a system while it is running is known as dynamic configuration. Although AMSM is not predicted for dynamic configuration, it is possible to add some new methods and adopt it for that. Therefore, AMSM is general enough to be adopted for a variety of environments and to meet demands of new application areas.

REFERENCES

[**1**] J. H. Alexander, M. J. Freiling, *Smalltalk-80 aids troubleshooting system development*, System & Software, **4**, 4 (1985), 111–118

[**2**] J. Diederich, J. Milton, *Experimental Prototyping in Smalltalk*, IEEE Software, May, 1987, 50–64

[**3**] E. Gold, M. B. Rosson, *Portia: An Instance-Centered Environment for Smalltalk*, in *OOPSLA '91 Conference Proceedings* (ed. Paepcke), **26**, 11 (1991), 62–74

[**4**] A. Goldberg, D. Robson, *Smalltalk-80: The language and its implementation*, Addison-Wesley, 1983

[**5**] G. Krasner (editor), *Smalltalk-80, Bits of History, Words of Advice*, Addison-Wesley, 1983

[**6**] G. Krasner, *The Smalltalk-80 Virtual machine*, BYTE, **6**, 8 (1981), 300–320

[**7**] J. Laursen, R. Atkinson, *Opus: A Smalltalk Production System*, in *OOPSLA '87 Conference Proc.* (ed. N. Meyrowitz), **22**, 12 (1987), 377–387

[**8**] M. Stadel, *Object Oriented Programming Techniques to Replace Software Components on the Fly in a Running Program*, SIGPLAN Notices, **26**, 1 (1991), 99–108

[**9**] A. Stritzinger, *Smalltalk: a slim application framework*, JOOP, **4**, 6 (1991), 11–18

(received 20 02 1992)

Matematički fakultet, Studentski trg 16, 11000 Beograd, Yugoslavia