Specification and Automated Recognition of Algorithmic Concepts with ALCOR *

Beniamino Di Martino[1] and Anna Bonifacio[2]

[1]Dipartimento di Ingegneria dell' Informazione -
Second University of Naples - Italy
[2] Covansys s.p.a - Rome - Italy
beniamino.dimartino@unina.it

Abstract. Techniques for automatic program recognition, at the algorithmic level, could be of high interest for the area of Software Maintenance, in particular for knowledge based reengineering, because the selection of suitable restructuring strategies is mainly driven by algorithmic features of the code. In this paper a formalism for the specification of algorithmic concepts, based on an automated *hierarchical concept parsing* recognition technique, is presented. Based on this technique is the design and development of ALCOR, a production rule based system for automatic recognition of algorithmic concepts within programs, aimed at support of knowledge based reengineering for high performance.

1 Introduction

The automatization of program comprehension techniques, even if limited to the algorithmic level, could be of high interest for the area of Software Maintenance, in particular for knowledge based reengineering to improve program performance.

We have designed and developed an original technique for automatic algorithmic concept recognition, based on production rule-based hierarchical concept parsing.

Based on this technique is the development of ALCOR [2], a tool for automatic recognition of algorithmic concepts within programs, aimed at support of program restructuring and porting for high performance [3]. First order logic programming, and Prolog in particular, has been utilized to perform the hierarchical concept parsing, thus taking advantage of Prolog's deductive inference rule engine. The input code is C, and the recognition of algorithmic concept instances within it is performed without any supervision from the user. The code recognition is nevertheless partial (only the concept instances specified in

---

* This work has been supported by the CNR - Consiglio Nazionale delle Ricerche, Italy (Agenzia 2000 Project *ALCOR* - n. CNRG00A41A), and by the Campania Reg. government (Projects "L41/96 Smart ISDN" and "Regional Competence Center on Information and Communication Technologies").

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2658, pp. 748–757, 2003.

the Alcor's inferential engine are recognized) and limited to the functional (algorithmic) level: concepts related to the application domain level of the source code aren't taken into consideration, in order to design a completely automatic procedure. These limitations are nevertheless irrelevant, due to the purpose of recognition: to drive parallelization strategies, the algorithmic level comprehension is sufficient, and it can be applied on partial portions of the code.

In this paper we describe a formalism for the specification of algorithmic concepts recognition, based on higher order attributed grammars.

The paper proceeds as follows: an overview of the developed technique is presented in sec. 2; in sec. 3 we describe a formalism for the specification of algorithmic concepts recognition, based on higher order attributed grammars. Sec. 4 provides with examples of specification of algorithmic concepts.

2 The Recognition Strategy

The recognition strategy is based on hierarchical parsing of algorithmic concepts. The recognition process is represented as a hierarchical abstraction process, starting from an intermediate representation of the code at the structural level in which *base concepts* are recognized; these become components of *structured concepts* in a recursive way. Such a hierarchical abstraction process can be modeled as a hierarchical parsing, driven by *concept recognition rules*, which acts on a description of *concept instances* recognized within the code.

The concept recognition rules are the production rules of the parsing: they describe the set of characteristics that allow for the identification of an algorithmic concept instance within the code.

The characteristics identifying an algorithmic concept can be informally defined as the way some abstract entities (the subconcepts), which represents a set of statements and variables linked by a functionality, are related and organized within a specific abstract control structure. By "abstract control structure" we mean structural relationships, such as control flow, data flow, control and data dependence, and calling relationships.

More specifically, each recognition rule specifies the related concept in a recursive way, by means of:

- a compositional hierarchy, recursively specified through the set of subconcepts directly composing the concept, and their compositional hierarchies;
- a set of conditions and constraints, to be fulfilled by the composing subconcepts, and relationships among them (which could involve subconcepts at different levels in the compositional hierarchy, thus not only the direct subconcepts).

In section 3 a formalism for the specification of the recognition rules is presented.

The properties and relationships which characterize the composing concepts have been chosen in such a way to privilege the structural characteristics respect to the syntactic ones. We have decided to give the data and control dependence relationships a peculiar role: they become the characteristics that specify the *abstract control structure* among concepts. For this purpose, they undergo an abstraction process during recognition, such as the concept abstraction process. This abstraction has been represented by the introduction of the notions of *abstract control and data dependence* among concept instances. The set of abstract data and control dependence relationships is produced within the context of the concept parsing process, and is explicitly represented within the program representation at the algorithmic level.

The direction of the concept parsing has been chosen to be *top-down* (descendent parsing). This choice is motivated by the particular task of the recognition facilities in the framework of the parallelization process. Since we are interested in finding instances of parallelizable algorithmic patterns in the code, an algorithmic recognition of the whole code is not mandatory: thus a top-down parsing (demand-driven), which leads to partial code recognition, is suitable, and allows for a much deeper pruning of the search space associated with the hierarchical parsing than the bottom-up approach.

The base concepts, starting points of the hierarchical abstraction process, are chosen among the elements of the intermediate code representation at the structural level. The code representation at the structural level (*basic representation*) is thus a key feature that affects the effectiveness and generality of the recognition procedure; we have chosen the *Program Dependence Graph* [4] representation, slightly augmented with syntactical information (e.g. tree-like structures representing expressions for each statement node) and control and data dependence information (edges augmented e.g. with control branch and data dependence level, type, dependence variable). Two main features make this representation suitable for our approach: (1) the structural information (data and control dependence), on which the recognition process relies, is explicitly represented; (2) it's an inherently delocalized code representation, and this plays an important role in solving the problem of concept delocalization. An overall *Abstract Program Representation* is generated during the recognition process. It has the structure of a *Hierarchical PDG* (HPDG), reflecting the hierarchical strategy of the recognition process.

3 A Formalism for the Specification of Algorithmic Concepts Recognition

Attributed grammars [8] have been selected as formalism for the specification of the recognition rules of the hierarchical concept parsing. Its effectiveness to specify relationships among structured informations, well exploited for the specification and structural analysis of programming languages, makes this formalism suitable for the program analysis at the algorithmic level too.

$CG = (G, A, R, C)$ is thus our *Concept Grammar*, with $G = (T, N, P, Z)$ its associated context-free grammar.

In the following the several components of the grammar are described, together with their relationships with the recognition process.

The set of *terminal symbols* of the grammar, T, represents the *base concepts*. These are terminals of the hierarchical abstraction process: they are thus elements of the program representation at the structural level. We give the role of base concepts to the elements of the structural representation which represent the executable program statements. The set of grammar terminals is thus, for Fortran 77: $T = \{\texttt{do}, \texttt{assign}, \texttt{if}\}$

The set of *nonterminal symbols* of the grammar, N, represents the algorithmic concepts recognized by the concept parsing.

The set of *start symbols* of the grammar, Z, represents the subset of algorithmic concepts, named PAPs (Parallelizable Algorithmic Patterns), which are associated with a specific set of parallelization strategies.

The set of production rules of the grammar, P, specifies the composition of the concepts represented by the lhs non-terminal symbols, and the relationships and constraints to be fulfilled by the instances of their subconcepts, represented by the rhs symbols.

The syntax of a production rule is as follows:

```
Rule =
    rule concept →
        composition
            { subconcept }
        condition
            [ local LocalAttributes ]
            { Condition }
        attribution
            { AttributionRule }

LocalAttributes =
    attribute : Type { attribute : Type }
```

concept $\in N$
subconcept $\in N \cup T$
attribute $\in A$
Condition $\in C$
AttributionRule $\in R$

A production rule specifies:

- a set {*subconcept*} of (terminal and nonterminal) symbols which represent the set of subconcepts composing the concept represented by the lhs symbol *concept*;
- the set {Condition} of the production's conditions; it represents the set of relationships and constraints the subconcept instances of the set {*subconcept*} have to fulfill in order for an instance of `concept` to be recognized;
- the set {AttributionRule} of the production's attribution rules. These assign values to the attributes of the recognized concept, utilizing the values of at-

tributes of the composing subconcepts (in this way we restrict the attributes of the grammar CG to be only synthesized ones).

Optionally, local attributes can be defined (the definition is marked by the keyword **local**).

The set of grammar *conditions*, C, is composed of predicates, or predicative functions, defined as follows:

```
Conditions =
    |  Condition & Conditions
    |  Condition '|' Conditions
    |  ¬ Conditions
    |  if Conditions then Conditions [ else Conditions ]

Condition =
    |  [(] Conditions [)]
    |  condition ( [ ParameterList ] )
    |  ['{'] ParameterList ['}'] := condition ( [ ParameterList ] )
    |  attribute = attribute
    |  attribute ≠ attribute
    |  attribute ∈ attributelist
    |  ∀  Condition   Condition
    |  ∃  Condition   s.t.   Condition

ParameterList =
    Parameters { , Parameters }

Parameters =
    |  [ attribute : ] Type
    |  ( [ attribute : ] Type   '|'   [ attribute : ] Type )
```

Conditions have grammar attributes, of different types, as parameters; in the case of predicative functions, they return a set of values (to be assigned to attributes of the corresponding type) if the condition on the input parameters is verified, and an undefined value otherwise. The conditions present in a grammar rule represent the constraints the corresponding concept has to fulfill, or the relationships among its composing subconcepts. The conditions represent constraints imposed to the parsing process, because if one of them is not satisfied, the current application of the corresponding rule fails, and the corresponding concept instance is not recognized. Conditions can be composed, by means of the usual logical connectives, the universal and existential quantifiers can be utilized, and alternatives can be tested by means of the conditional connective **if**. Conditions can applied to local attributes, and to attributes of symbols at the right hand side of the production. This constraint, together with the presence of synthesized attributes only, ensures that all the attributes which could be needed by a condition assume defined values when the rule is fired. This allows for the conditions' verification to be performed during the parsing process.

The set of grammar *attributes*, A, is composed of the attributes associated to the grammar symbols. As already mentioned, the attributes associated to a concept add an auxiliary semantical information to the semantical meaning represented by the concept; in addition, they permit to verify constraints and properties of the concept, and relationships among concepts; finally, they characterize the particular instances of the concept: two occurrences of a grammar symbol, presenting different values for at least one attribute, represent two different instances of the same concept.

4 Examples of Algorithmic Concepts' Specifications

We now highlight the flexibility and expressivity power of the formalism for the specification of the hierarchy, the constraints and the relationships among concepts through an example.

The algorithmic concept we consider is the *matrix-matrix product*, and the concept *vector-matrix product*, which is the subconcept of the previous. Due to space restrictions, we leave unexplained, even though we mention, all the other hierarchically composing subconcepts. The generic *vector-matrix product* operation is usually defined (in a form which includes application to multidimensional arrays) as:

$$VMP^{hkl} = [\cdots, DP_m^{hk}, \cdots] = \left[\cdots, \sum_i e(A(\cdots, \overset{h}{i}, \cdots) \times B(\cdots, \overset{k}{i}, \cdots, \overset{l}{m}, \cdots)), \cdots\right] \tag{1}$$

where DP^{hk} is the generic *dot product* operation, where $e(x)$ is a linear expression with respect to x, whose coefficient is invariant with respect to the i index of the sum. The vector-matrix operation involves the h-th dimension od the A array (the "vector") and the k-th and l-th dimensions of the array B (the "matrix"). The result is a monodimensional array, which can be assigned to a column of a multidimensional array.

The recognition rule for the `matrix_vector_product` is presented in figure 1, with related attributes presented in table 1. We illustrate the 1 in the following.

Its main component is the `dot_product` concept. The other components are two instances of the `scan` concept. The first scan instance (`scan[1]`) scans a dimension of the array, which records the result: this is specified by verifying that the scan statement is an assignment (to elements of the scanned array), sink of a dependence chain whose source is the assignment statement which records the result of the dot product. The second scan instance (`scan[2]`) scans the l-th dimension (cfr. eq: 1) of one of the arrays involved in the dot product. It has to be of course different from the dimension (k-th in 1) involved in the dot product. The two scans must share their `count_loop` subconcepts, in order to scan the two dimensions of the two arrays at the same time. These dimensions have to be completely scanned, in sequence (for both the scan concepts, the *range* attribute

Table 1. Attributes of the `matrix_vector_product` concept.

Attributes	kind
`in`	: instance
`hier`	: hierarchy
`vector_struct`	: struct of `ident` : identifyer `inst` : expression `dot_prod_subscr_exp` : expression `dot_prod_subscr_pos` : integer `dot_prod_index` : identifyer endstruct
`matrix_struct`	: struct of `ident` : identifyer `inst` : expression `dot_prod_subscr_exp` : expression `dot_prod_subscr_pos` : integer `dot_prod_index` : identifyer `matr_vec_subscr_exp` : expression `matr_vec_subscr_pos` : integer `matr_vec_index` : identifyer endstruct
`res_vector_struct`	: struct of `ident` : identifyer `inst` : expression `matr_vec_subscr_exp` : expression `matr_vec_subscr_pos` : integer `matr_vec_index` : identifyer endstruct

must be `whole` and *step* must be `sweep`). Finally, the `dot_product` instance must be control dependent from the `scan[1]` instance.

The *matrix-matrix product* operation is defined in terms of the matrix-vector product, as:

$$MMP^{hskl} = \left[\cdots, VMP_m^{hkl}, \cdots\right] = \begin{bmatrix} \cdots & \cdots & \cdots \\ \cdots & DP_{mn}^{hk} & \cdots \\ \cdots & \cdots & \cdots \end{bmatrix} = \begin{bmatrix} \cdots & \cdots & \cdots \\ \cdots & \sum_i e(A(\cdots, \overset{h}{i}, \cdots, \overset{s}{m}, \cdots) \times B(\cdots, \overset{k}{i}, \cdots, \overset{l}{n}, \cdots)) & \cdots \\ \cdots & \cdots & \cdots \end{bmatrix} \tag{2}$$

```
rule matrix_vector_product →
    composition
        scan[1]
        dot_product
        scan[2]
    condition
        local countLoopList1, countLoopList2 : [hierarchy]
                TERM[1], TERM[2] : instance
        control_dep(dot_product,scan[1],true)
        scan[1].hier = -(-,countLoopList1,-(TERM[1],-))
        TERM[1] = assign
        subexp_in_exp(scan[1].array_scan.array_inst,TERM[1].lhsExp)
        scan[2].hier = -(-,countLoopList2,-)
        countLoopList1 = countLoopList2
        scan[1].array_scan.scan_index = scan[2].array_scan.scan_index
        scan[1].range = scan[2].range = whole
        scan[1].stride = scan[2].stride = sweep
        ((scan[2].array_scan.array_inst = dot_product.array1_struct.inst &
          scan[2].array_scan.subscr_pos ≠ dot_product.array1_struct.subscr_pos)
         |
         (scan[2].array_scan.array_inst = dot_product.array2_struct.inst &
          scan[2].array_scan.subscr_pos ≠ dot_product.array2_struct.subscr_pos)
        )
        TERM[2] = last(dot_product.accum_struct.stmList)
        dep_chain(TERM[2],TERM[1])
```

Fig. 1. Recognition rule of the `matrix_vector_product` concept.

where VMP^{hkl} is the matrix-vector product, defined in 1, and DP^{hk}_{mn} is the dot product operation. The operation involves the h-th and s-th dimensions of the A array, and the k-th and l-th dimensions of the B array. The result is a bidimensional array, which can be assigned to a page of a multidimensional array.

The recognition rule for the `matrix_matrix_product` is presented in figure 2. We illustrate the 2 in the following.

Its main component is the `matrix_vector_product` concept. The other components are two instances of the `scan` concept. The first scan instance (`scan[1]`) scans a dimension of the array, which records the result of the matrix-vector product. The second scan instance (`scan[2]`) scans the s-th dimension (cfr. eq: 2) of the "vector" array of the matrix vector product. It has to be of course different from the dimension (h-th in 2) involved in the matrix-vector product. The two scans must share their `count_loop` subconcepts, in order to scan the two dimensions of the two arrays at the same time. These dimensions have to be completely scanned, in sequence (for both the scan concepts, the *range* attribute must be `whole` and *step* must be `sweep`). Finally, the `matrix__product` instance must be control dependent from the `scan[1]` instance.

```
rule matrix_matrix_product →
    composition
        scan[1]
        matrix_vector_product
        scan[2]
    condition
        local countLoopList1, countLoopList2 : [hierarchy]
        control_dep(matrix_vector_product,scan[1],true)
        scan[1].array_scan.array_inst =
          matrix_vector_product.res_vector_struct.inst
        scan[1].array_scan.subscr_pos ≠
          matrix_vector_product.res_vector_struct.matr_vec_subscr_pos
        scan[1].hier = -(-,countLoopList1,)
        scan[2].hier = -(-,countLoopList2,-)
        countLoopList1 = countLoopList2
        scan[1].array_scan.scan_index = scan[2].array_scan.scan_index
        scan[1].range = scan[2].range = whole
        scan[1].stride = scan[2].stride = sweep
        scan[2].array_scan.array_inst =
          matrix_vector_product.vector_struct.inst
        scan[2].array_scan.subscr_pos ≠
          matrix_vector_product.vector_struct.dot_prod_subscr_pos
```

Fig. 2. Recognition rule for the `matrix_matrix_product` concept.

5 Conclusion

In this paper a production-rule based hierarchical concept parsing recognition technique, and a formalism for the specification of algorithmic concepts have been presented.

The main contributions of the work presented can be summarized in the following: – definition of a formalism for the specification of algorithmic concepts, based on Higher Order Attributed Grammars, suitable for expressing in a flexible but exact way the compositional hierarchy and relationships among them, at any level within the hierarchy; – systematic utilization of structural properties (control, data dependence structure) more than syntactical one, in the definition and characterization of the algorithmic concepts; the utilization of powerful techniques for the analysis at the structural level (such as array dependences), and their abstraction; – utilization of symbolic analysis techniques for expressions, to face the syntactic variation problems not solvable through the algorithmic characterization at the structural level; – development of a technique for automated *hierarchical concept parsing*, which implements the mechanism of hierarchical abstraction defined by the grammar, utilizing the first order logic programming (Prolog); – representation of the concept instances recognition within the code, and of their hierarchical structure, through an *Abstract Hierarchical Program De-*

pendence Graph; – focus on algorithms and algorithmic variations within code developed for scientific computing and High Performance Computing.

References

1. T.J. Biggerstaff, "The Concept Assignment Problem in Program Understanding", Procs. *IEEE Working Conf. on Reverse Engineering*, May 21-23, Baltimore, Maryland, USA, 1993.
2. B. Di Martino, "ALCOR - an ALgorithmic COncept Recognition tool to support High Level Parallel Program Development", in: J. Fagerholm, J. Haataja, J. Jrvinen, M. Lyly, P. Rback, V. Savolainen (Eds.): *Applied Parallel Computing. Advanced Scientific Computing*, Lecture Notes in Computer Science, n. 2367, pp. 150-159, Springer-Verlag, 2002.
3. A. Bonifacio, B. Di Martino "Algorithmic Concept Recognition support for Skeleton Based Parallel Programming", Proc. of *Int. Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'03)* - Nice (FR), 22-26/4 2003, Apr. 2003. IEEE CS Press.
4. J. Ferrante, K.J. Ottenstein and J.D. Warren, "The Program Dependence Graph and its use in Optimization", *ACM Trans. Programming Languages and Systems*, 9(3), pp. 319-349, June 1987.
5. J. Grosh and H. Emmelmann, "A Tool Box for Compiler Construction", *Lecture Notes in Computer Science*, Springer - Verlag, n. 477, pp. 106-116, Oct. 1990.
6. "Puma - A Generator for the Transformation of Attributed Trees", Compiler Generation Report n. 26, GMD Karlsruhe, July 1991.
7. M.T. Harandi and J.Q. Ning, "Knowledge-Based Program Analysis", *IEEE Software*, pp. 74-81, Jan. 1990.
8. D. E. Knuth, "Semantics of context-free languages", *Math. Syst. Theory*, 2(2) pp. 127-145, 1968.
9. W. Pugh, "A practical algorithm for Exact Array Dependence Analysis", *Communications of ACM*, 8(35), pp. 102-115, Aug. 1992.
10. H. Vogt, S. Swiestra and M. Kuiper, "Higher Order Attribute Grammars", Proc. of *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 131-145, June 1989.
11. L.M. Wills, "Automated Program Recognition: a Feasibility Demonstration", *Artificial Intelligence*, 45, 1990.