Specification and automatic recognition of algorithmic concepts within programs

Beniamino Di Martino

Institute for Software Technology and Parallel Systems,
University of Vienna - Austria
and
Dipartimento di Informatica e Sistemistica,
University "Federico II" - Naples - Italy
`dimartin@par.univie.ac.at, dimartin@grid.unina.it`

Tech. Rep. n. 97-14, November 1997

Abstract

In this paper a formalism for the specification of algorithmic concepts recognition is presented. It is coupled to an automated *hierarchical concept parsing* recognition technique. It allows for a successful handling of the languages and the kind of codes, algorithms and optimizations connected to scientific and high performance computing. Its flexibility and expressivity power for the specification of the hierarchy, the constraints and the relationships among concepts is exemplified through a case study.

1 Introduction

The problem of assigning concepts to code does not seem to be automatically solvable in its general form [2], because the concepts oriented to human domains are inherently ambiguous, and their recognition heavily relies on *a priori* knowledge on the particular domain. Nevertheless, if the recognition is limited to the *algorithmic level*, the task seems to be manageable. The winning strategy [16, 8] is to structure the recognition process as an increasingly *abstraction process*, performed in a hierarchical way: elementary concepts are recognized first, and then they become components of larger grained concepts, in a recursive way. A number of problems have to be dealt with, when pursuing the task of automatic algorithm recognition. These have been remarked in [16], and range from *syntactic variation*, *delocalization* (the implementation of a concept can be spread throughout the code), *implementation variation* (an abstract algorithmic functionality could be expressed by more than one concrete algorithm), *overlapping implementations* (instances of distinct algorithms could share code portions), *data structure implementation variations*, to *code optimizations*.

The automatization of program comprehension techniques, even if limited to the algorithmic level, could be of high interest for the area of *Program Parallelization*, because the selection of suitable parallelization strategies is driven by algorithmic features of the code.

Recognition of the algorithms that the code implements would help the parallelization process, by allowing the introduction of heuristics and an extensive pruning of the search space associated to the transformation selections, and thus enabling the application of more aggressive code transformations. The tasks to which algorithmic recognition is applicable include (1) automatic data distribution, (2) selection of sequences of optimizing code transformations, (3) replacement of code implementing recognized functionalities by optimized sequential libraries, and (4) parallel code generation and communication optimization – through code replacement with parallel library calls and high-level collective communication primitives.

Preexisting techniques for automated program comprehension seem to be inadequate to treat the languages and the kind of code and algorithms connected to the parallelization domain. Here, the main issues to be faced are represented by the limited expressiveness of the imperative languages utilized for scientific computation (Fortran), with the consequent explosion of syntactic and implementation variations, the complexity of algorithms, and the fact that many programs are optimized with respect a specific target architecture.

It induced us to design and develop an original technique for the automatic recognition of Parallelizable Algorithmic Patterns. It deals with recognition of algorithmic concepts within optimized code, recognition of irregular computations, and in general solve all the above mentioned problems of the recognition, within the context of imperative languages typically used for scientific computation (we focus on Fortran 77 and 90).

In this paper, after giving an overview of the developed technique (sec. 2), we present (sec. 3) a formalism for the specification of algorithmic concepts recognition, based on higher order attributed grammars. Its flexibility and expressivity power for the specification of the hierarchy, the constraints and the relationships among concepts is exemplified in sec. 4, through a case study.

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; such relationships are specified in a recursive way: an instance of the concept C_i is defined as *abstract data (control) dependent* on an instance of the concept C_j iif exists a determinate pattern of abstract data (control) dependence relationships among the subconcepts composing the instance of C_i and the instance of C_j. This pattern is characteristic of each concept C_i, and specified within its recognition rule. 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* [7] 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 (see Fig. 1). It has the structure of a *Hierarchical PDG* (HPDG), reflecting the hierarchical strategy of the recognition process. As long as the parsing process proceeds and more and more abstract concepts are recognized, they are represented as nodes in increasingly higher layers of the HDPG. The nodes of this graph are connected by two kind of edges. The *hierarchy edges* connect each node representing a concept to the lower layer nodes representing its subconcepts. The graph structure determined by this kind of edges represents the hierarchy of abstraction; this structure is generally a tree, excepted in the case of shared concepts, i.e. when a concept instance is subconcept of more than one concept (see Fig. 1). The *dependence edges* link together nodes that have abstract control and data dependence relationships between them. Note that, during the recognition process, dependence edges for the newly created abstract concept nodes are inherited from those of the composing subconcept nodes in a way that is characteristic of each concept (see the annotated edges in fig. 1).

The described approach to algorithmic pattern recognition permits successful handling of the problems arising in the context of program comprehension, within the context of imperative languages like Fortran and scientific computation. The syntactic variation problem is solved by: (1) characterizing inter-statement level concepts with the structural properties of control and data dependence; (2) representing program expression by means of abstract structures (v. sec. 3) and performing symbolic analysis of them by means of functions which manipulate the

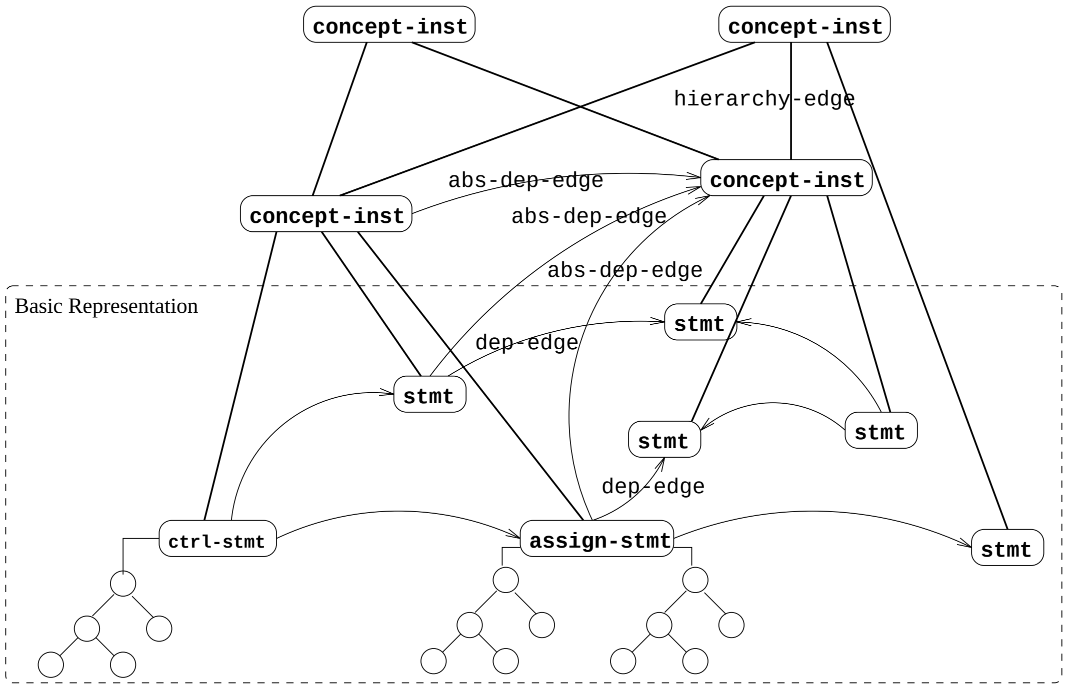


Figure 1: Abstract program representation.

abstract structures. The delocalization problem is solved by the characteristics of the Abstract Program Representation which: (1) is based on an inherently delocalized structural representation (Program Dependence Graph); (2) has a global scope of visibility, so that the active rule can attempt to match all instances of concepts already recognized, at every level of the abstraction. Although this characteristic in principle increases the complexity of the process, the systematic use of control and data dependence relationships to characterize concepts allows the application of rules to be driven by the locality typically present in the source program. In this way complexity can be maintained at an acceptable level, without constraining the delocalized recognition capability. The implementation variation problem is solved by the backtracking feature of the recognition process. More specifically, backtracking allows the specification of one concept by means of multiple rules; each rule specifies a different algorithmic implementation of the same concept. However, backtracking has also its drawbacks. If on one hand it makes the recognition procedure more powerful and general, on the other hand it makes the search complexity to grow exponentially with the code size. Nevertheless, as we have observed above, both the top down approach and the summarization of derived subconcepts within HPDG nodes should prune the search space considerably making practical the analysis of non trivial pieces of code. Finally, the overlapping implementation problem is solved by the global scope of visibility of the representation, and by the fact that the parsing mechanism does not restrict the use of a subconcept to one rule, thus allowing the recognition even in presence of shared concept instances.

An important consequence of the features just discussed is the independence from restructuring techniques, that modify the original code before and during the recognition process to deal with delocalized code and implementation variations. This means that our approach does not need a canonical form for concept implementations (even though pre-applied restructuring transformations could be still useful in certain situations to speed up the recognition process).

A prototype following the described technique has been developed [4] (its implementation is described in [6]). It has been integrated within the *Vienna Fortran Compilation System* (VFCS) [18], an integrated and interactive environment for parallelization of sequential code (Fortran 77-90) and the development and compilation of High Performance Fortran (HPF) data-parallel code. It relies on Prolog as a system shell and takes advantage of Prolog's deductive inference-rule engine to perform the hierarchical concept recognition. Moreover, it utilizes the structural analysis of the input code performed by the VFCS front-end, as a basis to build the internal representation of the program to be analyzed. The output of the tool is a graphical browser that permits the visualization of the hierarchical description of the recognized concepts, together with their implementation within the program code, as it will be exemplified in sec. 4.

3 A formalism for the specification of algorithmic concepts recognition

Attributed grammars [10] 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.

(Terminal symbols $\equiv$ Base Concepts) 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: $T = \{\texttt{do}, \texttt{assign}, \texttt{if}\}$

(NonTerminal symbols $\equiv$ Concepts) The set of *nonterminal symbols* of the grammar, N, represents the algorithmic concepts recognized by the concept parsing.

(Start Symbols $\equiv$ PAPs) 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.

(Productions $\equiv$ Recognition rules) 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* $\rightarrow$
    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 attributes 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**).

(Language generated by CG $\equiv$ concepts' implementations) For each $z \in Z$, the language $L(z, CG)$ generated by the grammar CG is the set:

$$L(z, CG) = \{w \in \mathcal{P}(T) \;\; t.c. \;\; z \overset{CG}{\Rightarrow} w\}$$

That is, a set $w \in \mathcal{P}(T)$ of program's statements belongs to the grammar language iif it can be derived from one of its start symbols z. $L(z, CG)$ thus represents the set of all the implementations of the concept represented by z (which can be recognized by the grammar productions).
w is a *sentence* of CG; it represents a set of program instructions which is associated to a concept.

(Meaning of a sentence $\equiv$ "meaning" of the associated concept) The *meaning* of a sentence w of CG, derived from the symbol z, is the set of values of the attributes of z.
The values of the attributes associated to the symbol z representing the concept, add an auxiliary semantical information to the semantical meaning represented by the concept; moreover, these values characterize the particular instance of the concept, that is its implementation w.

(Derivation tree $\equiv$ compositional hierarchy) The correspondence between the concept grammar symbols and algorithmic concepts, determines a correspondence between the derivation tree of a symbol and the compositional hierarchy of the associated concept. The derivation tree nodes represent the instances of recognized concepts. The sons of a given tree node represent the instances of the subconcepts which compose the concept associated with that node.
Actually, in our case the derivation tree represents an (acyclic) graph. To take into account the problem of code sharing among concepts, we allow that a same concept instance can be a

component of more than one concept. This leads the compositional hierarchy to have an acyclic graph structure; this graph structure is represented within the derivation tree by associating an attribute to each node, which represents the particular instance of the concept. Two derivation tree nodes having the same instance number represent two occurrences of the same instance of a concept.

(Conditions $\equiv$ properties and rel. among concepts) 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.

(Attributes $\equiv$ concept instances characterization) 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.

Attributes are of different types; moreover, they can be structured; we have defined three derived types: the list, the structure and the tree (this latter only for attributes of kind `hierarchy` and `exp_structure`

The types are defined as follows:

Type =
| instance
| identifier
| value
| type
| hierarchy
| expression
| exp_instance
| exp_structure
| Enumerative
| List
| Structure

Enumerative = '{' *ident* { , *ident* } '}'

List = '[' Type ']'

Structure = struct of { FieldDef } endstruct

FieldDef =
| *field* : Type
| case *tag* : Enumerative of { TagFieldDef } endcase

TagFieldDef = *tagvalue* { , *tagvalue* } :: { *field* : Type } ;

An attribute a associated to a symbol X will be referred as $X.a$.

Values are assigned to attributes by means of the *attribution operator* `:=`. It is possible to assign alternative values to attributes, depending on conditions, by means of the conditional connective **if**:

attribute := **if** Condition **then** *attribute* **else** *attribute*

Attributes of kind *instance* are associated to every symbol, terminal and nonterminal. Their values identify the instances of the associated concepts. It is important not to confuse the concept instance with the occurrence of a concept during the parsing process: an *instance of a concept* refers to a determinate implementation of that concept in the code, while an *occurrence of a concept* refers to the occurrence of its symbol within the derivation tree. Two different occurrences of a concept could refer to the same concept instance (it means that the concept instance is shared by two different concepts at a higher layer). The instance attribute of a symbol can be univocally represent the values of the other attributes (two concept occurrences representing the same concept instance will have equal attributes): thus it is utilized, in the rules specification, to access the values of non-local attributes, belonging to symbols not present in the rule. For sake of compactness, we allow the instance attribute of a symbol to inherit the attributes (and values) of the symbol. For instance, given a symbol X whose instance attribute is in and having other attribute $attr$, we have:

$$\mathrm{X.in.attr} \equiv \mathrm{X.attr} \equiv \mathrm{in.attr}$$

Attributes of kind *hierarchy* are also associated to every symbol. Their values identify the derivation trees of the associated symbols, and thus represent the hierarchical structure of the associated concepts. Due to the presence of such attributes, the *CG* grammar becomes a *higher order* attributed grammar [15]. The structure of their values is the following (*linear predicative notation*):

ConceptHier =
| *concept* (*concept.in* , ConceptHier {, ConceptHier })
| *stm_pointer*

Such higher order predicative terms represent in a suitable way the hierarchical structure of concepts: the functor of the term represents the concept, its first argument represent the concept instance, and the following arguments represent the composing subconcepts, in a recursive way (the terminals of this structure actually represent pointers to the syntax tree nodes of the statements corresponding to the base concepts).
The hierarchy attribute is fundamental to specify relationships among concepts at any level in the compositional hierarchy, because of the presence of the instance attributes within them: such attributes inherit the other attributes of the corresponding concept occurrence, making their values visible to the environment of the current rule.

Attributes of kind *List* denote lists of homogeneous attributes. The following operators can be applied to attribute lists: the operator $\in$ (thus considering the list as a set), the usual functions *car* and *cdr* to extract the first element and the rest of the list, the function *elnum* to extract an element by position (thus considering the list as an array) and the function *last* to extract the last element. Finally, the *pairing* function, applicable to two lists with the same cardinality:

$$A = [\cdots, a_i, \cdots];\ B = [\cdots, b_i, \cdots]$$
$$C = A \odot B = [\cdots, [a_i, b_i], \cdots]$$

Attributes of kind *expression* represent the expressions present in the statements' code. This kind is actually a derived type, structured as follows:

$$\text{expression} = \begin{array}{l} \text{struct of} \\ \quad \texttt{instance} : \text{exp\_instance} \\ \quad \texttt{actual} : \text{exp\_structure} \\ \text{endstruct} \end{array}$$

The field `instance` represents the instance of the expression within the code, while the field `actual` represent the structure of the expression.
The structure of the values of the attributes of kind *exp_structure* is the following (*predicative notation*):

Terminals	Attributes	kind	Values
`assign`	`in`	instance	
	`hier`	hierarchy	`assign(in,stp)`
	`LhsExp`	expression	
	`RhsExp`	expression	
`do`	`in`	instance	
	`hier`	hierarchy	`do(in,stp)`
	`IndexExp`	expression	
	`InitExp`	expression	
	`LastExp`	expression	
	`StepExp`	expression	exp. or `null`
`if`	`in`	instance	
	`hier`	hierarchy	`if(in,stp)`
	`CondExp`	expression	

Table 1: Attributes associated to terminal symbols.

ExpHierarchy =
 | CommBinOp (ExpHierarchy , ExpHierarchy)
 | BinOp (ExpHierarchy , ExpHierarchy)
 | UnOp (ExpHierarchy)
 | *arr_el* (ExpHierarchy { , ExpHierarchy })
 | *scal*
 | *val*

CommBinOp =
 add | mul | eqv | neqv | and | or | eq | ne
BinOp =
 minus | div | exp | sign | lt | le | gt | ge | not

arr_el, *scal* : identifier
val : value

Terminals of this tree structure are attributes (of kind *identifier*) representing variables and constants of the program, and attributes (of kind *value*) representing numerical or boolean values.

In table 1 the attributes associated to terminal symbols are listed.

3.1 An overview of grammar Conditions

In table 2 the main grammar conditions defined are listed, grouped following the kind of the attributes to which they are applied.
Conditions on attributes of kind *identifier* verify the type and the kind of data structure of the variable associated to a program identifier. In Fortran 77 the defined data structures are the constant, the scalar and the array.

Conditions on attributes of kind *exp_instance* allow for the verification of properties on the whole expression, or for the access to subexpressions having determinate properties. This latter feature is possible by composing (by means of the logic connectives) the functions that allow for the access to subexpressions (*inst_in_exp*, *subexp_in_exp*, *exp_in_stm*) with the conditions

Conditions on attributes of kind *identifier*:
type := constant(identifier)
type := scalar_var(identifier)
{type, numDim : int, dimList : [int]} := array_var(identifier)
Conditions on attributes of kind *exp_instance*:
value := val_inst(exp_instance)
identifier := const_inst(exp_instance)
identifier := scal_var_inst(exp_instance)
{identifier, subscrList:[exp_instance]} := arr_el_inst(exp_instance)
(value | identifier) := constant_inst(exp_instance)
identifier := var_inst(exp_instance)
(value | identifier) := elem_inst(exp_instance)
{op1 : exp_instance, op2 : exp_instance} := binary_exp(exp_instance)
oper : exp_instance := unary_exp(exp_instance)
{(value | identifier), subExp : exp_instance} := inst_in_exp(exp_instance) subExp : exp_instance := inst_in_exp((value | identifier), exp_instance) inst_in_exp((value | identifier), exp_instance)
subexp_in_exp(subExp : exp_instance, exp_instance)
exp_in_stm(exp_instance, stm_instance : instance)
Conditions on attributes of kind *exp_structure*:
resExp : exp_structure := replace_elem_with_exp(ident : identifier, repExp : exp_structure, exp : exp_structure)
val : value := constant(exp : exp_structure)
{termExp : exp_structure, coeffExpList : [exp_structure]} := linear_resp_to(exp : exp_structure, identList : [identifier])
{termExp : exp_structure, coeffExpList : [exp_structure]} := linear(exp : exp_structure)
equivalent(exp_structure, exp_structure)
struct_equiv(exp_structure, exp_structure)
Conditions on attributes of kind *instance*:
control_dep(sink : instance, source : instance, branch : {true, false})
data_dep(kind : {true,anti,output}, sink : instance, source : instance, var : identifier, level : int | `infty`)
{inList : [instance], varList : [identifier]} := dep_chain(start : instance, end : instance, kind : {true,anti,output}, level: int | `infty`)

Table 2: Conditions.

on them. Typical conditions are the verification that the (sub)expression be an instance of a program element (*val_inst, arr_el_inst,* $\cdots$; these return the identifier, or the value), or that it has a certain structure (*binary_exp, comm_exp,* $\cdots$).

Conditions on attributes of kind *exp_structure* allow for the inspection of the structure of expressions, and allow for the application of operations on it, like normalization, replacement, evaluation or comparison. It is possible to build expressions, with the use of the linear predicative notation previously defined, and values of attributes of kind *exp_structure* and *identifier* or *value*. The expression built can be assigned to an attribute (of kind *exp_structure*), using the attribution operator `:=`. In the following we describe some of the defined predicates.

The predicative function *replace_elem_with_exp* transforms the expression `exp` to `resExp`, by replacing each instance of `ident` within `exp` with the expression `repExp`.

The predicative function *constant* verifies if the expression `exp` is constant (composed of constants and values); it can be evaluated, and its value assigned to `val`.

The predicative function *linear_resp_to* verifies if the expression `exp` can be linearized with respect to the list of variables in `identList`, that is put in the form:

$$coeff_1 * i_1 + coeff_2 * i_2 + \cdots coeff_m * i_m + term$$

with $identList = \{i_1, i_2, \cdots, i_m\}$ and $coeff_1$, $coeff_2$,$\cdots$, $coeff_m$, $term$ independent of $i_1, i_2, \cdots, i_m$. It returns the coefficient expressions.

The predicative function *linear* verifies if the expression `exp` can be linearized, that is put in the form:

$$coeff_1 * i_1 + coeff_2 * i_2 + \cdots coeff_m * i_m + term$$

with $coeff_1, coeff_2, \cdots, coeff_m, term$ constant expressions. It returns the coefficient expressions.

The predicate *equivalent* verifies if the expressions `exp1` and `exp2` are equivalent. The equivalence property is tested accordingly to the following semantics, increasingly more general:

1. the expressions are both constant: they are evaluated, and the value compared;

2. the expressions can be linearized with respect to the same set of variables, and the corresponding coefficient expressions are equivalent;

3. the expressions are *structurally* equivalent (that is: present an isomorphic structure, taking into account the commutativity property) and have the corresponding variables coincident.

The predicate *struct_equiv* verifies if two expressions are structurally equivalent (without necessarily having coincident variables).

Conditions on attributes of kind *instance* are applied to concept instances.

The conditions *control_dep* and *data_dep* represent the relationships of (abstract) control and data dependence among concept instances. The attributes `source` and `sink` refer to the concept instances source and sink of the dependence. The parameter `branch` of the *control_dep* condition represent the branch of control where the dependence appears. The parameter `kind` of the *data_dep* condition represent the kind of dependence (`true`, `anti` or `output`), while `level` represent the dependence level (`infty` for loop independent dependences); `var` is the identifier of the variable causing the dependence.

The domain of these two predicates is the set of (recognized) concept instances; it is built during the parsing process: thus the truth table for these predicates is built during the parsing process.

The predicative function *dep_chain* can be applied to instances of the base concepts representing assignments only. It is verified if a path exists, within the PDG, between the assignments `start` and `end`, composed only of data dependence edges, of kind `kind` and level `level`. This path is composed of assignment statements, whose instances `inList` are returned by the function, which also return the corresponding variables involved in the dependence chain.

4 Example of specification, recognition and application to code restructuring

In this section the features of the formalism and the procedure for algorithmic specification and recognition are exemplified through a case study.

The algorithmic concept we consider is the *matrix-matrix product*, and the concept *vector-matrix product*, which is the subconcept of the previous.

The generic *vector-matrix product* operation is usually defined (in a form which includes application to multidimensional arrays) as:

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

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 2, with related attributes presented in table 3. We illustrate the 2 in the following.

Its main component is the `dot_product` concept. The other components are two instances of the `scan` concept. Due to space restrictions, these two concepts are presented (recognition rules) in Appendix, but are left unexplained. 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 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:

$$MMP^{hskl} = \left[\cdots, VMP^{hkl}_{m}, \cdots\right] = \begin{bmatrix} \cdots & \cdots & \cdots \\ \cdots & DP^{hk}_{mn} & \cdots \\ \cdots & \cdots & \cdots \end{bmatrix} =$$

$$\begin{bmatrix} \cdots & \cdots & \cdots \\ \cdots & \sum_i e(A(\cdots, \overset{h}{i}, \cdots, \overset{s}{m}, \cdots) \times B(\cdots, \overset{k}{i}, \cdots, \overset{l}{n}, \cdots)) & \cdots \\ \cdots & \cdots & \cdots \end{bmatrix} \qquad (2)$$

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

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

Table 3: Attributes of the `matrix_vector_product` concept.

```
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])
```

Figure 2: Recognition rule of the `matrix_vector_product` concept.

```
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
```

Figure 3: Recognition rule for the `matrix_matrix_product` concept.

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 3, with related attributes presented in table 4. We illustrate the 3 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.

We now present some examples of recognition of instances of the matrix-matrix product concept (and consequently the composing subconcepts matrix-vector product and dot product). We show how the recognition of the hierarchical composition of such instances can drive a suitable code restructuring and replacement with calls to optimized libraries (e.g. HPF Intrinsics) which takes into account implementation variations and optimizations present in the code under examination.

The first example (a code excerpt directly taken from the BLAS 3 implementation of the matrix-matrix product) is shown in fig. 4. The result of the recognition process for this code excerpt is presented in figure 5. This figure (and the following) exemplify the browsing capabilities of

Attributes	Kind
`in`	: instance
`hier`	: hierarchy
`matrix1_struct`	: struct of `ident` : identifyer `inst` : expression `dot_prod_subscr_exp` : expression `dot_prod_subscr_pos` : integer `dot_prod_index` : identifyer `matr_matr_subscr_exp` : expression `matr_matr_subscr_pos` : integer `matr_matr_index` : identifyer endstruct
`matrix2_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_matrix_struct`	: struct of `ident` : identifyer `inst` : expression `matr_vec_subscr_exp` : expression `matr_vec_subscr_pos` : integer `matr_vec_index` : identifyer `matr_matr_subscr_exp` : expression `matr_matr_subscr_pos` : integer `matr_matr_index` : identifyer endstruct

Table 4: Attributes of the `matrix_matrix_product` concept.

```
   do 10 i=1,N
     do 20 j=1,N
       z(i,NP,j) = beta*z(i,NP,j)
20   continue
     do 30 k=1,N
       temp = alpha*x(i,k,L)
       do 40 j=1,N
         z(i,NP,j) = z(i,NP,j) + temp * y(M,k,j)
40     continue
30   continue
10 continue
```

```
z(:,NP,:) = beta*z(:,NP,:)
z(:,NP,:) = MATMUL(alpha*x(:,:,L),y(M,:,:))
```

```
   do 10 i=1,N
     z(i,NP,:) = beta*z(i,NP,:)
     do 30 k=1,N
       temp = alpha*x(i,k,L)
       z(i,NP,:) = z(i,NP,:) + temp*y(M,k,:)
30   continue
10 continue
```

Figure 4: Matrix matrix product instance and restructuring: first example.

the prototype: a recognized concept instance within a code excerpt is represented graphically in its hierarchical composition; recognized concept instances and their composing subconcepts can be related back to the implementing source code (highlighting it in the text window) simply by clicking on the corresponding graph nodes in the graph window, as it is shown in fig. 5, for the overall concept instance and two composing subconcepts. The components of a (sub)concept are also represented, once clicked on its node, as highlighted nodes in the graph.

In this example, the product is between arrays X and Y, both tridimensional ones. It involves the first and second dimension of X (the second involved in the dot product), and the second and the third of Y (the second involved in the dot product). The result is recorded in a page of the array Z, along the first and third dimension, whose values are updated after being scaled by `alpha`. The values of the page of X are also scaled by `beta`, before the dot product operation. These operation are intermixed with the product implementation, but this does not represent a problem for the recognition. Moreover, the code excerpt presents an optimization, constituted by (1) a loop interchange between the loop involved in the dot product (n. 12 in fig. 5) and the one involved in the matrix vector product (n. 14), and (2) the introduction of the temporary variable `temp`. This implementation variation is easily dealt by the recognition technique: the loops are properly recognized as belonging to their own subconcepts, and the use of the temporary variable is represented by a dependence chain (`dep_chain`), as can be seen in fig. 5.

The remaining parts of figure 4 show how the code excerpt can be restructured in a F90 code, with a call to the `MATMUL` intrinsic, or without it, if the optimization already present in the code is sufficient.

The second example is a code excerpt representing a tiled implementation of the matrix-matrix product; it is shown in fig. 6. The result of the recognition process for this code excerpt is presented in figure 7.

This implementation of the matrix-matrix product contains two optimizations, which represent interesting implementation variations: the utilization of a temporary *accumulator* variable (`accum`), and the *tiling* of the loop nest iterations. The first optimization is taken into account within the recognition of the matrix-vector product subconcept, and is represented with a dependence chain (cfr. fig. 7b); the tiling optimization is taken into account within the recognition of the component scans, by means of the strip-mining variation of them (cfr. fig. 7)

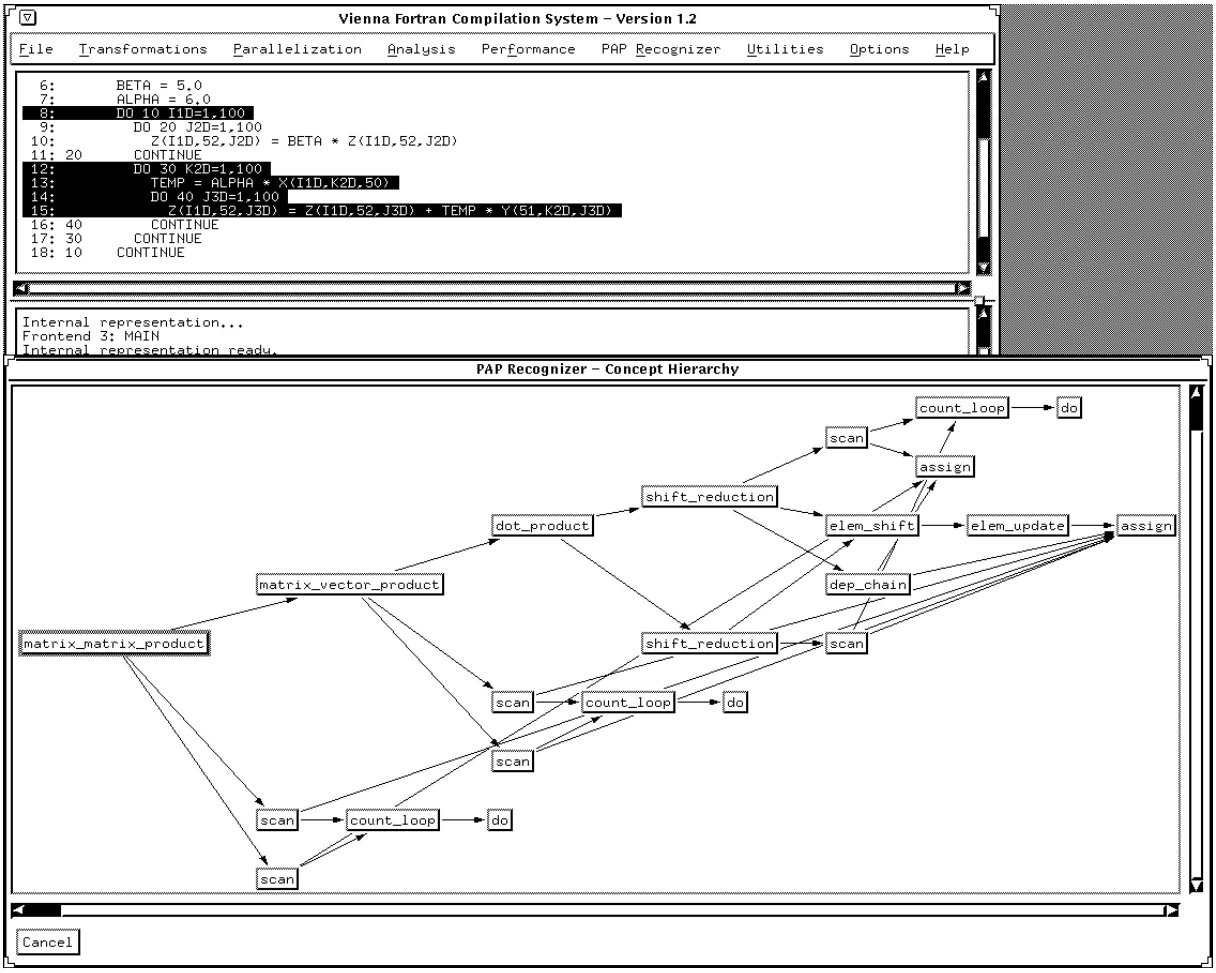
The remaining parts of figure 6 show how the code excerpt can be restructured in a F90 code, in different ways: either it can be totally replaced by a call to the `MATMUL` intrinsic; or both the previous optimizations can be preserved, but the dot product subconcept instance can still be replaced by a call to the `DOT_PRODUCT` intrinsic; or only the tiling optimization can be preserved: this time the matrix product operation, restricted on each tile can be replaced by a `MATMUL` call (note the proper setting of the range bounds for the parameters x and y of the `MATMUL` call).

5 Related work

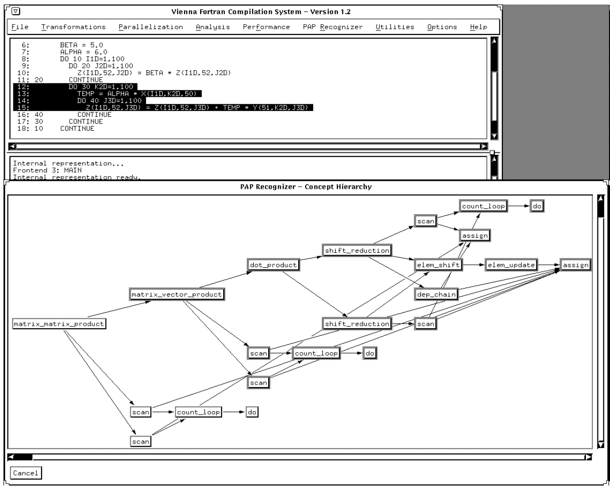
The approach taken in developing this algorithmic recognition technique is mainly inspired by the works of Wills [16] and Ning [8].

The common points are: the hierarchical abstraction process of the recognition, the *code-driven* approach (i.e. mainly the information that can be extracted from the source code is exploited, avoiding a systematic use of models of the application domain of the program), and the use of a program representation at the structural level.

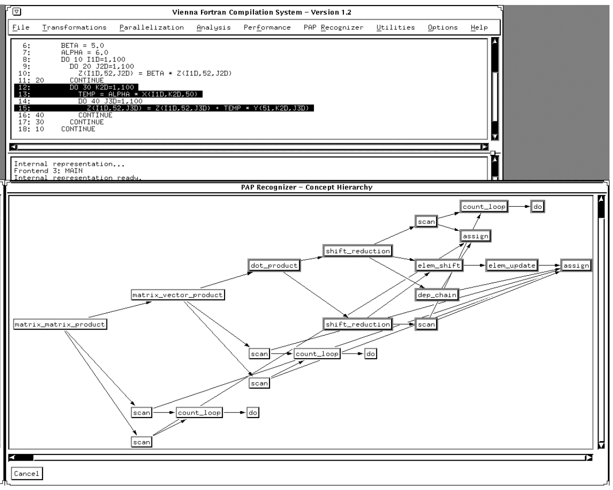
The differences, besides the aims of the recognition (these two works are targeted towards Reverse Engineering, while ours towards parallelizing compilation) and the target languages (the first works on Lisp, the second on Cobol, ours on Fortran), are essentially in the abstract program representation, in the abstraction procedure, in the properties and compositional structures



`matrix_matrix_product`



`matrix_vector_product`



`dot_product`

Figure 5: Recognition of `matrix_matrix_product` and its subconcepts in the first example.

```
   do 10 ii=1,LB
     do 10 jj=1,MB
       do 10 kk=1,NB
         do 10 i=(ii-1)*LD+1,ii*LD
           do 10 j=(jj-1)*MD+1,jj*MD
             accum = 0.0
             do 40 k=(kk-1)*ND+1,kk*ND
               temp1 = x(i,k)
               temp2 = y(k,j)
               accum = accum + temp1 * temp2
40           continue
             z(i,j) = accum + z(i,j)
10 continue
```

```
z = z + MATMUL(x,y)
```

```
   do 10 ii=1,LB
     do 10 jj=1,MB
       do 10 kk=1,NB
         do 10 i=(ii-1)*LD+1,ii*LD
           do 10 j=(jj-1)*MD+1,jj*MD
             accum = 0.0
             accum = DOT_PRODUCT( x(i,((kk-1)*ND+1):(kk*ND) ),&
                                  y( ((kk-1)*ND+1):(kk*ND),j))
             z(i,j) = accum + z(i,j)
10 continue
```

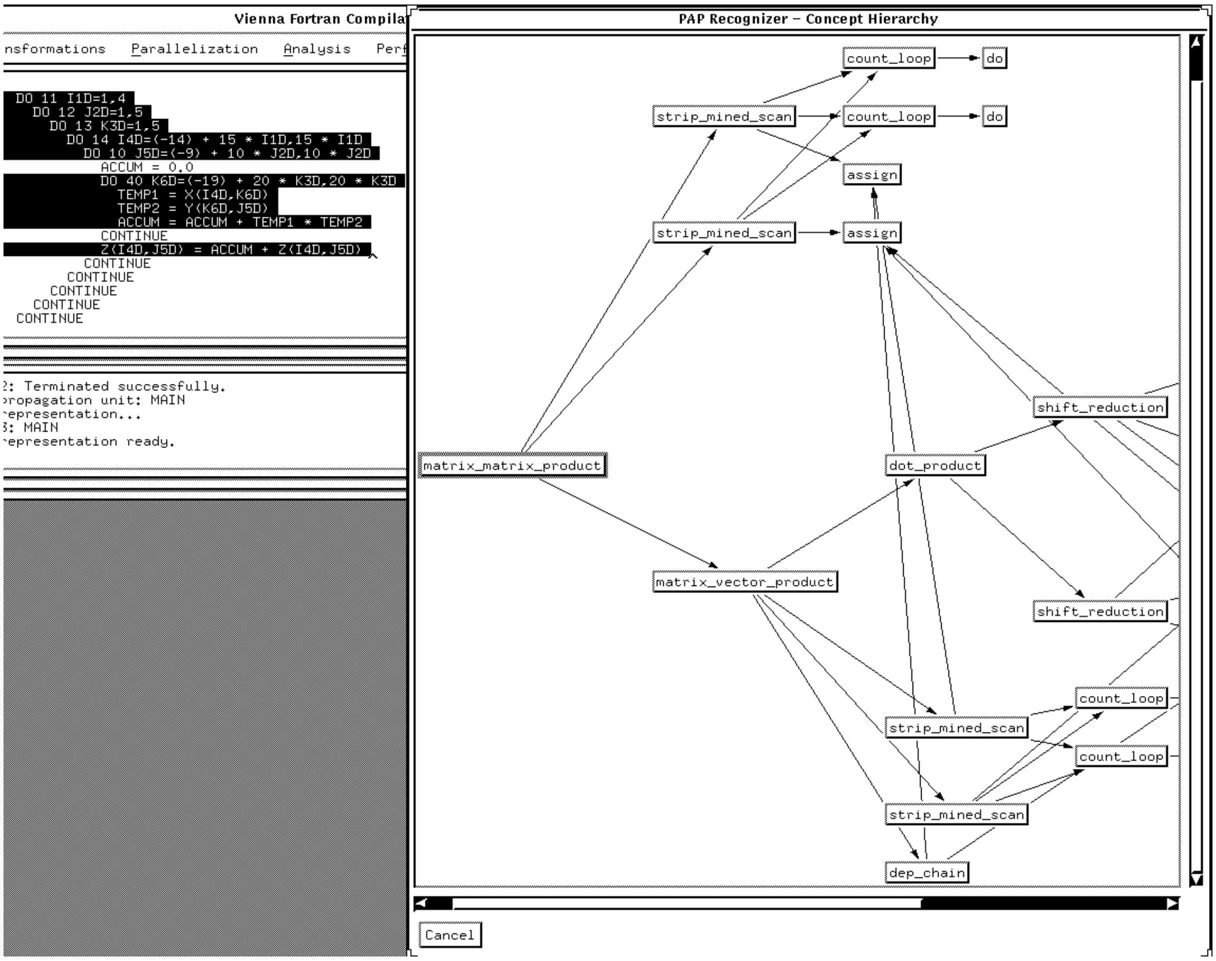
```
   do 10 ii=1,LB
     do 10 jj=1,MB
       do 10 kk=1,NB
         z( ((ii-1)*LD+1):(ii*LD),((jj-1)*MD+1):(jj*MD) ) = &
           z( ((ii-1)*LD+1):(ii*LD),((jj-1)*MD+1):(jj*MD) ) + &
           MATMUL(x(((ii-1)*LD+1):(ii*LD),((kk-1)*ND+1):(kk*ND)),&
                   y(((kk-1)*ND+1):(kk*ND),((jj-1)*MD+1):(jj*MD)))
10 continue
```

Figure 6: Matrix matrix product instance and restructuring: second example.

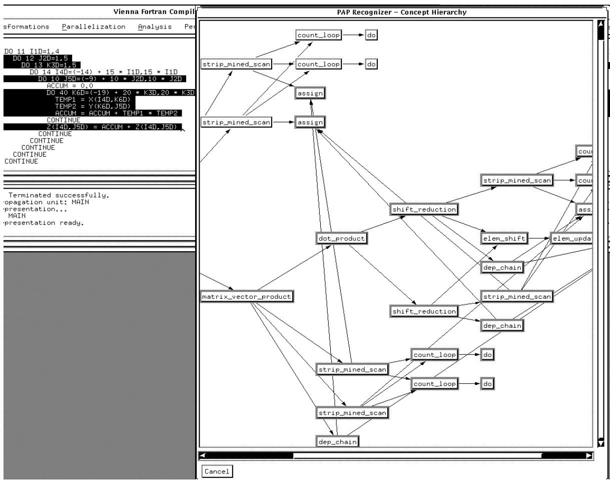
which specify the algorithmic concepts, and in the specification formalism.

Wills' *GRASPR* is based on the abstract program representation of Plan Calculus, that is essentially a control-data flow graph, and eliminates the control flow edges, substituting them with control environment information, encoded as graph node and data-flow edge attributes. Our basic program representation is a Program Dependence Graph (augmented with syntactical informations). The control structure is thus explicitly represented, directly as dependence; the data-flow information is, at the same way, directly represented as dependence. The set of structures we add on the basic PDG representation (especially the tree-like node attributes representing the sub-expressions of statements) allow for a more detailed level of representation of the program, which covers the syntactical level, and that is better suited to deal with imperative languages like Fortran.

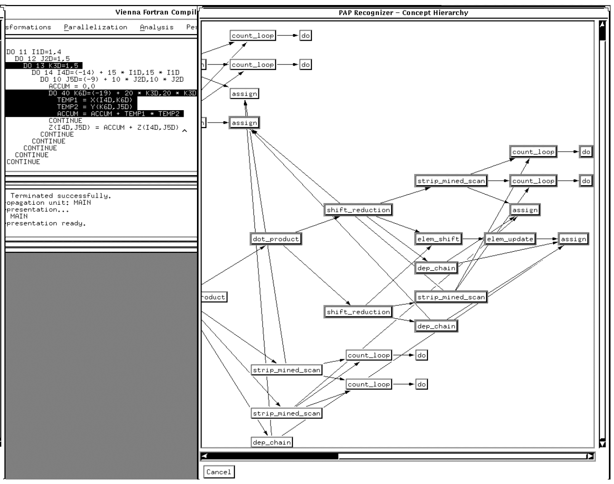
The hierarchical abstraction performed in the Wills' approach is a graph parsing process, thus the representation and the concepts' specifications are in form of graphs; our specification formalism is based on higher-order attributed grammars, the parsing mechanism is constrained by sets of conditions on attributes (which represent constraints on concepts and relationships among them). We believe that the specification formalism based on higher order grammar is



matrix_matrix_product



matrix_vector_product



dot_product

Figure 7: Recognition of matrix_matrix_product and its subconcepts in the second example.

more flexible and expressive than the one based on graphs; in particular: (1) the graph based formalism well represents the structural relationships among concepts, but not that well the conditions on them, and constraints and syntactical features; while the expressivity of productions allows for a more natural specification of constraint and relationships, of syntactical nature too. (2) The higher order attributes mechanism allows for the specification of relationships among concepts at different levels in the composition hierarchy, while the graph based formalism allow to specify (in a simple way) relationships only among the direct components of a concept. (3) the concept specification by means of a graph, thus with a fixed number of edges, makes very rigid the definition of the relationships a concept can have with other concepts. This is, for us, a limitation of the approach; a witness of this is the fact that the edges represent only data-flow relationships, that can be associated to a concept specification, but not the control flow relationships, and the dependency relationships.

The Ning's approach is based on a program representation, an Abstract Syntax Tree (AST) augmented with control and data flow-dependence informations, which includes the syntactical and structural level, but is dominated by the syntax, as opposed to the Program Dependence Graph. This difference is reflected on the recognition representation: the Ning's approach annotates the AST with semantical information and abstract concepts, while ours builds an abstract PDG, which explicitly represents the hierarchies among concept instances, and the control-data dependencies among them, in an abstract way.

The Ning's approach to recognition, based on inference rules, is more flexible than ours, based on parsing: it can allow, in principle, for partial specification and for probabilistic derivation, and could include informations from other domains than the code. Conversely, the formalism is less precise, and the features that are really utilized for the concepts specification remain in the syntactical and, partly, in the structural domain. The dependence information is utilized, on control and on data, but the data dependence analysis is limited to scalars (DU/UD chains), not covering array dependences.

Previous works have focussed on the application, to the parallelization area, of techniques that can be viewed as being, in a restricted sense, "algorithmic recognition". Such techniques are called in the literature *idiom recognition* or "pattern matching", and refer to the detection of elementar functionalities in the code, such as reductions or recurrences. They essentially perform a matching of canonical forms within the code, thus working at the syntactical level, with no consideration of strategies for hierarchical abstraction.

– A pioneristic work on idiom recognition was developed by Wolfe [17] and implemented within the Parafrase system [11]. – Eave [3] is an expert system for the interactive vectorization of Fortran programs. It includes a simple syntactical pattern matching tool which detects reductions. – Redon and Feautrier [14] propose an approach, based on algebraic specification, for recurrences detection; it eventually relies on syntactical pattern matching. – The method of Pinter and Pinter [13] is based on a Program Dependence graph as program representation; this is then modified making use of rewrite rules and utilizing a list of graph patterns. The use of a structural representation such as the PDG makes this approach non limited by the syntactical variation problem, but the non-structured specification of the graph patterns denies to the approach the flexibility proper of a hierarchical specification and recognition.

Currently, three other efforts aim at the application of program understanding techniques to parallelizing compilation: work by Kessler [9], and proposals of Bhansali et al. [1] and of Metzger [12]. A comparison between Kessler's approach and ours is in [5].

6 Conclusion

In this paper a 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, 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 Dependence Graph*; – focus on Fortran, and treatment of algorithms and algorithmic variations within code developed for scientific computing and High Performance Computing.

A first version of the prototype is operatinal, and able to recognize dozens of concepts related to numerical computation. We are currently working on the development of a tool supporting the translation of sequential Fortran code into HPF, based on the technique described. It will accept structured Fortran 77/90 code, and will produce fully annotated HPF code, with insertion of calls to optimized parallel libraries, such as the HPF intrinsics.

References

[1] S. Bhansali, J.R. Hagemeister, "A Pattern Matching Approach for Reusing Software Libraries in Parallel Systems", Proc. of *Workshop on Knowledge Based Systems for the Reuse of Program Libraries*, Sophia Anthipolis (FR), Nov. 1995.

[2] T.J. Biggerstaff, "The Concept Assignment Problem in Program Understanding", Procs. *IEEE Working Conf. on Reverse Engineering*, May 21-23, Baltimore, Maryland, USA, 1993.

[3] P. Bose, "Interactive Program Improvement via EAVE: an Expert Adviser for VEctorization", Proc. *Int. Conference on Supercomputing*, pp. 119-130, July 1988.

[4] B. Di Martino, "Sviluppo di Tecniche di Riconoscimento Automatico di Algoritmi nel Codice per il Supporto alla Compilazione Parallelizzante", PhD Thesis, Dipartimento di Informatica e Sistemistica, Università "Federico II" di Napoli, Feb. 1996, (in Italian).

[5] B. Di Martino and C.W. Kessler, "Program Comprehension Engines for Automatic Parallelization: A Comparative Study", Proc. of *First Int. Workshop on Software Engineering for Parallel and Distributed Systems*, Berlin (D), Chapman & Hall, Mar. 1996.

[6] B. Di Martino, G. Iannello, "PAP Recognizer: a Tool for Automatic Recognition of Parallelizable Patterns", Proc. of 4^{th} *IEEE Workshop on Program Comprehension*, Berlin (D), IEEE, Mar. 1996.

[7] 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.

[8] M.T. Harandi and J.Q. Ning, "Knowledge-Based Program Analysis", *IEEE Software*, pp. 74-81, Jan. 1990.

[9] C. W. Kessler, "Pattern-Driven Automatic Program Transformation and Parallelization", Proc. *3rd EUROMICRO Workshop on Parallel and Distributed Processing", San Remo*, Italy, Jan. 1995.

[10] D. E. Knuth, "Semantics of context-free languages", *Math. Syst. Theory*, 2(2) pp. 127-145, 1968.

[11] B. Leasure, *The Parafrase Project's Fortran Analyzer. Major module documentation*, Tech. Rep. n. 85-504, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1985.

[12] R. Metzger, "Automated Recognition of Parallel Algorithms in Scientific Applications", Proc. of *Workshop on Plan Recognition at IJCAI'95*, Aug. 1995.

[13] S. Pinter and R. Pinter, "Program Optimization and Parallelization Using Idioms", Proc. *ACM SIGPLAN Principles of Programming Languages*, pp. 79-82, 1991.

[14] X. Redon and P. Feautrier, "Detection of Recurrences in Sequential Programs with Loops", Proc. *PARLE 93, Springer LNCS vol. 694*, pp. 132-145, 1993.

[15] 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.

[16] L.M. Wills, "Automated Program Recognition: a Feasibility Demonstration", *Artificial Intelligence*, 45, 1990.

[17] M.J. Wolfe, *Optimizing Supercompilers for Supercomputers*, Ph. D. Thesis, Tech. Rep. n. 82-1009, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1982.

[18] H. Zima and B. Chapman, "Compiling for Distributed-Memory Systems", *Proceedings of the IEEE*, Special Section on Languages and Compilers for Parallel Machines, 81(2), pp. 264-287, Feb. 1993.