嵌入一致图语法的依赖图*

李国东+, 张德富

(南京大学 计算机软件新技术国家重点实验室,江苏 南京 210093)

Dependency Graphs Embedding Confluent Graph Grammars

LI Guo-Dong+, ZHANG De-Fu

(State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China)

+ Corresponding author: Phn: +86-25-3596612, E-mail: magicflute@263.net

Received 2002-12-31; Accepted 2003-08-12

Li GD, Zhang DF. Dependency graphs embedding confluent graph grammars. *Journal of Software*, 2004, 15(7):956~968.

http://www.jos.org.cn/1000-9825/15/956.htm

Abstract: Graph grammars have been developed as an extension of the formal grammars on strings to grammars on graphs, and provide a mechanism in which transformations on graphs can be modeled in a mathematically precise way. In this paper, based on confluent graph grammars, the authors present a novel representation for data-flow graphs, control-flow graphs, combined control-data-graphs, bipartite graphs and hyperedge graphs. How to extract parallelism is specified automatically at different levels by graph rewriting, thus facilitating the design and implementation of parallel compilers and parallel languages.

Key words: graph grammar; graph rewriting; dependency graph; hypergraph; compiling

摘 要: 图语法将字符串上的形式文法扩充为图上的形式文法,提供一种能够使用精确的数学方法来模拟图变换的机制.提出了几种新的基于一致图语法的方法来表示控制流图、数据流图、控制数据流图、二分图和超图,并说明如何通过图重写来自动生成依赖图并挖掘并行性,从而协助并行编译器和并行语言的设计和实现.

关键词: 框架图语法;图重写;依赖图;超图;编译

中图法分类号: TP301 **文献标识码**: A

1 Introduction

Parallel computing has become a natural medium for solving a wide variety of computationally intensive problems in a very fast and efficient manner. Currently, more and more distributed software systems base their platforms on heterogeneous connected workstation clusters. However, the inherent complexity of many parallel

* **LI Guo-Dong** was born in 1975. He received the BS degree in computer science from Sun Yat-sen University, China, in 1997, and the MS degree in computer science from Nanjing University, China, in 2001. His research interests include parallel and distributed computing. He is a member of the IEEE Computer Society. **ZHANG De-Fu** was born in 1937. He is a professor and doctoral supervisor at Nanjing University. His current research areas include parallel processing and distributed systems.

computing applications makes the development using existing parallel programming paradigms both time-consuming and error-prone.

In parallel computing, dependency graphs are widely used to represent data and control dependencies among the computations. Two kinds of graphs are generally used: DAGs (directed acyclic graphs) and TIGs (task interaction graphs). Traditionally, the data and control dependencies among components in a program executing in parallel are described as DAGs, whose vertices and edges represent computations and communications/ dependencies respectively. TIGs are used to describe the computation and communication structures of another category of applications.

For DAGs, we generally use specific clustering algorithms and greedy algorithms to map and schedule tasks on multiprocessors; for TIGs, we often use clustering and partitioning approaches in the parallelization of calculations. The objective is to minimize the finish time of the last component, minimize overall communication costs, or maximize task throughput. Recently, researchers argue that standard graph model has significant shortcomings, and propose some new graph diagrams as an improvement. A bipartite graph model and a hypergraph model were proposed in Ref.[1]; two computational hypergraph models were proposed in Ref.[2]. For example, for parallelizating the computations through rowwise or columnwise decomposition of an m*m square matrix A with the same sparsity structure as the coefficient matrix[2], traditional graph model is suitable for the partitioning of nonsymmetric matrices. We need more powerful graph models to represent the dependency relations emerging in this problem, and also need a systematic and formal method to derive the significant characteristics of the graphs.

We need a general method being capable of expressing all these graphs in an efficient way. Traditionally, dependency graphs are represented by usual graphs that consist of the node set and edge set. The whole structures of these graphs must be maintained during the whole process they are involved. Because dependency graphs are used to express the intricate dependency constraints among up to thousands of nodes, some crucial properties should be deducted out by analyzing the structures of these graphs. However, since the traditional graphs only capture the connection relations between nodes and edges, it is extremely hard to get structural information concerning with different components in the graphs, not to mention to find the intrinsic properties which are not evident without applying some systematic methods. Moreover, parallel programming requires the development of visual environments and visual tools to facilitate the analysis and processing of these dependency graphs[3]. If we can figure out the semantic characteristics of dependency graphs and deduct some other properties by applying a formal deduction method, we will obtain much more useful information than what we can obtain by just analyzing the traditional node and edge connecting graphs. This is the motivation of our paper.

Graph grammars are motivated by considerations about pattern recognition, compiler construction, and data type specification, and then applied to the modeling of concurrent systems, massively parallel computer architectures, visual languages and many others[4]. They provide a mechanism in which local transformation on graphs can be modeled in a mathematically precise way.

Recall the formal language theory based on strings, which is a natural way for describing the complex situations at an intuitive level. For example, the following production rules are a compact recursive notation for specifying how to derive legal syntactic constructs of fraction. The numeral string 789 can be derived by a recursive application of the production rules.

⟨fraction⟩::= ⟨digit ⟩ | ⟨digit⟩⟨fraction⟩

Similarly, they are a natural generalization of the formal language theory based on strings and the theory of term rewriting based on graphs to facilitate the description and analysis of the dependency graphs that are crucial in compiler construction and parallel computing.

In this paper we present some graph-grammar-based representation for data-flow graphs, control-flow graphs,

control-/data-flow graphs, bipartite graphs and hypergraphs to facilitate the design of parallel compilers. The structure of this paper is as follows. In Section 2 we introduce some basic concepts and properties of the Node Replacement Graph Grammars and Hyperedge Replacement Graph Grammars. In section 3 we present our graph grammars for a variety of graphs. In sections 4 and 5 we give the discussion and the conclusion respectively.

2 Confluent Graph Grammars

In this section, we introduce the concepts and definitions of node replacement graph grammars and hyperedge replacement graph grammars given by other researchers[5~7].

A graph grammar consists of a finite set of productions. A production is a triple (M, D, E) where M and D are graphs and E is some embedding mechanism. When it is applied to a graph H, the occurrence of M is removed from H and replaced by an isomorphic copy of D, then the embedding mechanism E is used to attach D to the remainder H- of H. Node replacement where a node is replaced by a new subgraph which is connected to the remainder of the graph and edge/hyperedge replacement where a hyperedge is replaced by a new subhypergraph which is glued to the remainder of the hypergraph, are the two basic choices for rewriting a graph.

2.1 Node replacement graph grammars

When the mother graph consists of only one node, the node replacement is called NLC (node label controlled) mechanism. Formally, an NLC graph grammar is a system $G=(\Sigma, \Delta, P, C, S)$ where Σ-Δ and Δ are the alphabets of nonterminal and terminal node labels respectively; P is a finite set of NLC productions, C is a connection relation, and S is the initial graph. The graph language generated by G is $L(G) = \{H \in GR_\Delta \mid S =>^* H\}$ where GR_Δ is the set of the undirected graphs with node labels in Δ, => represents one rewriting step, and =>* represents a derivation, i.e. a sequence of rewriting steps. An extension of NLC called NCE (neighbored controlled embedding) is much more powerful, where each production in P is of the form X -> (D, C) such that X -> D is an NLC production and C is a connection relation "for D", i.e. $C \subseteq \Sigma \times V_D$ (where V_D is the set of nodes of D). We can further extend the NCE approach to the directed node-labeled graphs in which C consists of triples (μ, δ, d), where $d \in \{in, out\}$. (μ, δ, in) means that the embedding process should establish an edge to each node labeled δ in D from each node labeled μ that is an "in-neighbour" of the mother node m. Similarly, (μ, δ, out) means that the embedding process should establish an edge from each node labeled δ in D to each node labeled μ that is an "out-neighbour" of the mother node m. NLC-like graph grammars with directed edge together with edge labels are called edNLC grammars $G = (\Sigma, \Delta, \Gamma, P, C, S)$ where Γ is the set of edge labels[6].

Formally, let Σ and Γ be an alphabet of node labels and edge labels respectively. A graph over Σ and Γ is a tuple $H = (V, E, \lambda)$, where V is the finite set of nodes, $E \subseteq \{(v, \gamma, w) \mid v, w \in V, v \neq w, \gamma \in \Gamma\}$ is the set of edges, and $\lambda : V \to \Sigma$ is the node labeling function. For graph H, these components are denoted as V_H, E_H, and λ_H, respectively. For unlabeled nodes and/or edges, $\Sigma = \{\#\}$ and/or $\Gamma = \{*\}$. $GR_{\Sigma,\Gamma}$ denote the set of all graphs over Σ and Γ, and a subset of $GR_{\Sigma,\Gamma}$ is called a graph language. A production of an edNCE grammar has the form X -> (D, C) where X is a nonterminal node label, D is a graph, and C is a set of connection instructions. A graph with embedding over Σ and Γ is a pair (H, C) with $H \in GR_{\Sigma,\Gamma}$ and $C \subseteq \Sigma \times \Gamma \times \Gamma \times V_H \times \{in, out\}$. A connection instruction (α, β, γ, x, d) can be written as (α, β/γ, x, d), when $d = out$ it means that if there is a β-labeled edge from the mother node v for which (D, C) is substituted to a node w, then the embedding process will establish a γ-labeled edge from x to w. Similarly for '*in*' instead of '*out*'.

An edNCE grammar is confluent if the result of derivation does not depend on the order in which the productions are applied. That is, an edNCE grammar G is confluent, or a C-edNCE grammar, if for all productions X_1 -> (D_1, C_1) and X_2 -> (D_2, C_2) in P, all nodes $x_1 \in V_{D1}$ and $x2 \in V_{D2}$, and all edge labels α, $\beta \in \Gamma$, the following

equivalences hold: $\exists\beta \in \Gamma$: $(X_2,\alpha/\beta,x_1,out)\in C_1$ and $(\lambda_{D1}(x_1),\beta/\delta,x_2,in)\in C_2 \Leftrightarrow \exists\gamma\in\Gamma(X_1,\alpha/\gamma, x_2, in)\in C_2$ and $(\lambda_{D2}(x_1), \gamma/\delta,x_1,out)\in C_1$.

2.2 Hyperedge replacement graph grammars

In hyperedge replacement, after the hyperedge in structure D is removed, the replacing structure is glued into the original structure M by fusing each external node in D with the corresponding attachment node in M. Formally, a multi-pointed hypergraph over Σ is a tuple $G = (V, E, s, t, l, begin, end)$, where V is the finite set of nodes, E is the finite set of hyperedges, $s : E \to V^*$ is the source function, $t : E \to V^*$ is the target function, $l : E \to \Sigma$ is the labeling function such that $type(l(e)) = (|s(e)|, |t(e)|)$ for every $e \in E$, where $|s(e)|$ and $|t(e)|$ are the length of string $s(e)$ and $t(e)$ respectively, $begin \in V^*$ is the sequence of begin nodes, and $end \in V^*$ is the sequence of end nodes[7]. The components of G will also be denoted by V_G, E_G, S_G, T_G, L_G, $begin(G)$, and $end(G)$. G is said to be of $type(m, n)$, i.e. $type(g) = (m, n)$, if $|begin(G)| = m$ and $|end(G)| = n$. Similarly we will write $type(e)$ to denote $type(l(e))$. The set of hypergraphs over Σ will be denoted GR_Σ, and GR denotes the union of all GR_Σ. A (typed) graph language is a subset L of GR_Σ for some Σ such that all graphs in L have the same $type(m, n)$, and the type of L is denoted by $type(L) = (m, n)$.

A basic operation on hypergraphs is the substitution of a hypergraph for an edge. Let e be an edge of hypergraph G and H be a graph satisfying $type(H) = type(e) = (m, n)$. By removing e and adding H we get the hypergraph G', $G' = (V_G \cup V_H, (E_G - \{e\}) \cup E_H, s, t, l, begin(G), end(G))$, where $s(e) = s_G(e)$ for $e\ E_G - \{e\}$ and $s(e) = s_H(e)$ for $e\ E_H$, and similarly for t and l. The substitution of h for e in G, denoted by $G[e/H]$, is the hypergraph G'/R where $R = \{S_G(e)(i), begin(H)(i) \mid 1 \le i \le m\} \cup \{t_G(e)(i), end(H)(i) \mid 1 \le i \le n\}$. Furthermore, a replacement is a mapping $\Sigma \to GR$ such that $type(\varnothing(\sigma)) = type(\sigma)$ for every σ, it is extended to a mapping from $GR\Sigma$ to GR by defining, for $G \in GR$, $\varnothing(G)=G[e_1/\varnothing(l(e_1))]\ldots[e_k/\varnothing(l(e_k))]$ where $EG=\{e_1,\ldots,e_k\}$. It is well known that this definition does not depend on the order $e_1, \ldots, e_k$ in which the edges are replaced.

Hyperedge replacement grammars (or HR grammars) are context-free graph grammars that substitute graphs for edges. A HR grammar is a tuple $G = (N, T, P, S)$ where N is a typed alphabet of nonterminals, T is a typed alphabet of terminals (disjoint with N), P is a finite set of productions, and $S \in N$ is the initial nonterminal. Every production in P is the form $X \to H$ which $X \in N$, $H \in GR_{N\cup T}$, and $type(X) = type(H)$. Application of production $p = X \to H$ to a graph yields graph $g[e/H]$, i.e. $G \Rightarrow_p G'$. A sequence of the direct derivations $H_0 \Rightarrow \ldots \Rightarrow H_k$ is called a derivation of length k from H_0 to H_k and is denoted by $H \Rightarrow^* H_k$ or $H \Rightarrow^*_P H'$ or $H_0 \Rightarrow^k H_k$.

HR perseveres the confluence property. Let H be a hypergraph with distinct $e_1, e_2 \in E_H$ and let H_i be a hypergraph with $type(H_i) = type_H(e_i)$ for $i \in \{1, 2\}$, then $H[e_1/H_1][e_2/H_2] = H[e_2/H_2][e_1/H_1]$. The concepts of derivation trees as a convenient representation of derivations are similar to those of node replacement grammar.

3 Dependency Graphs

3.1 Related work

Over the past several decades, there has been growing interest in simultaneous exploitation of parallelism in sequential applications. For instance, because the HPF standard does not provide directives for expressing task parallelism at this time, researchers have independently proposed directives for expressing task parallelism in HPF programs.

- ✧ Fortran-M[8] is initially conceived as an extension to Fortran for expressing task parallelism. Recently, it has been integrated with HPF[9] in order to enable the simultaneous exploitation of task and data parallelism.
- ✧ The Parafrase compiler project[10] parallelizes Fortran applications for shared memory machines. It analyzes an input Fortran program and constructs a HTG (hierarchical task graph) representation for the program.

The HTG representation aims to capture parallelism information at all levels of granularity.

✧ Dhagat, *et al*.[11] have proposed a language, UC, that allows for the expression of task and data parallelism. The UC compiler generates code that executes the program in a task and data parallel manner.

✧ Shankar, *et al*.[12] have proposed the hierarchical MDG (macro dataflow graph) representation which is used for programs and is very similar to the HTG (hierarchical task graph). It abstracts much of the information of granularity for Fortran programs and is well-suited for our allocation and scheduling algorithms.

All these works use traditional graphs to represent the dependency graphs, and develop corresponding tools to facilitate the construction and modification of these graphs. Some visual tools have been also developed too. However, they suffer from the weaknesses in expressive capacity and automatic deduction which are inherent to the traditional graph method. In this paper, we present a new scheme of detecting and representing parallelism using graph grammar techniques. Our representation is very powerful and is able to capture dependency information at all levels of granularity in a variety of programs.

3.2 Control-Flow graphs

Task graphs have been used as a convenient abstraction of parallel computations and programs in virtually all areas of parallel processing and exploiting parallelism requires synchronization between control- and data-dependent tasks. CFGs (control-flow graphs) of structured programs (which only contain three kinds of control structure: sequence, condition and loop) are useful for data flow analysis. These control-flow graphs can be generated by the edNCE grammar $G_{c1} = (\{S, X, i, d, \#\}, \{i, d, \#\}, \{*\}, P, C)$, where P contains the productions given in Fig.1. Note that the unlabeled nodes/edges in the Figure have labels # and *, respectively.

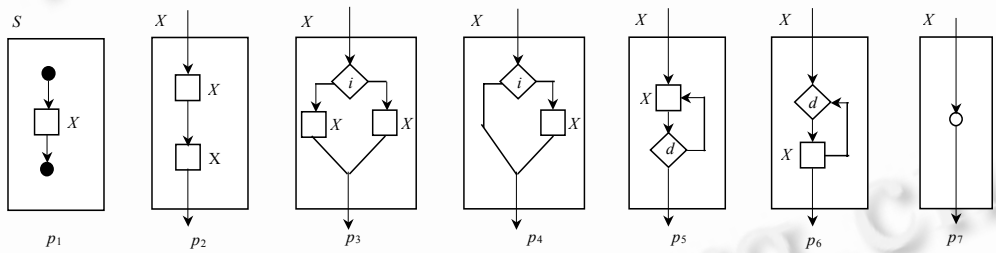

Fig.1 Productions of structured control flow graphs G_{c1}

A derivation example of $L(G_{c1})$ is shown in Fig.2, where the productions p_1, p_2, p_7, p_4, p_6 and p_7 are applied in sequence.

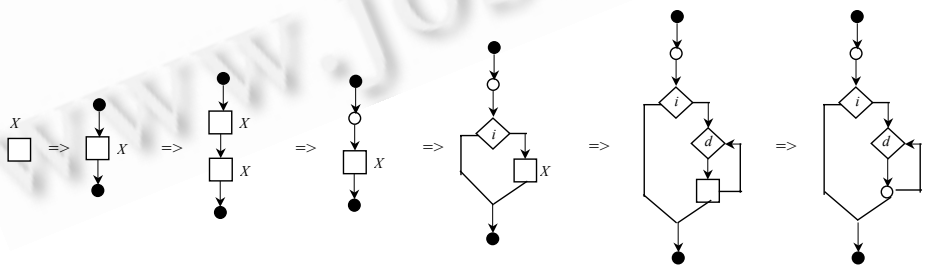

Fig.2 A derivation of graph grammar G_{c1}

It is easy to check that edNCE G_{c1} is confluent, thus the result of derivation does not depend on the order in which the productions are applied.

Similarly, the control-flow graphs can be generated by the HR grammar $G_{c2} = (\{C, D\}, \{c, d\}, P, C)$, where P contains the productions given in Fig.3. A derivation example of $L(G_{c2})$ is given in Fig.4.

As we have mentioned, HR grammars (including G_{c2}) are context-free graph grammars, and have the properties such as sequentialization and parallelization, associativity and confluence, so we can use the concepts of derivation trees and apply Leftmost Derivation to obtain the target graph language.

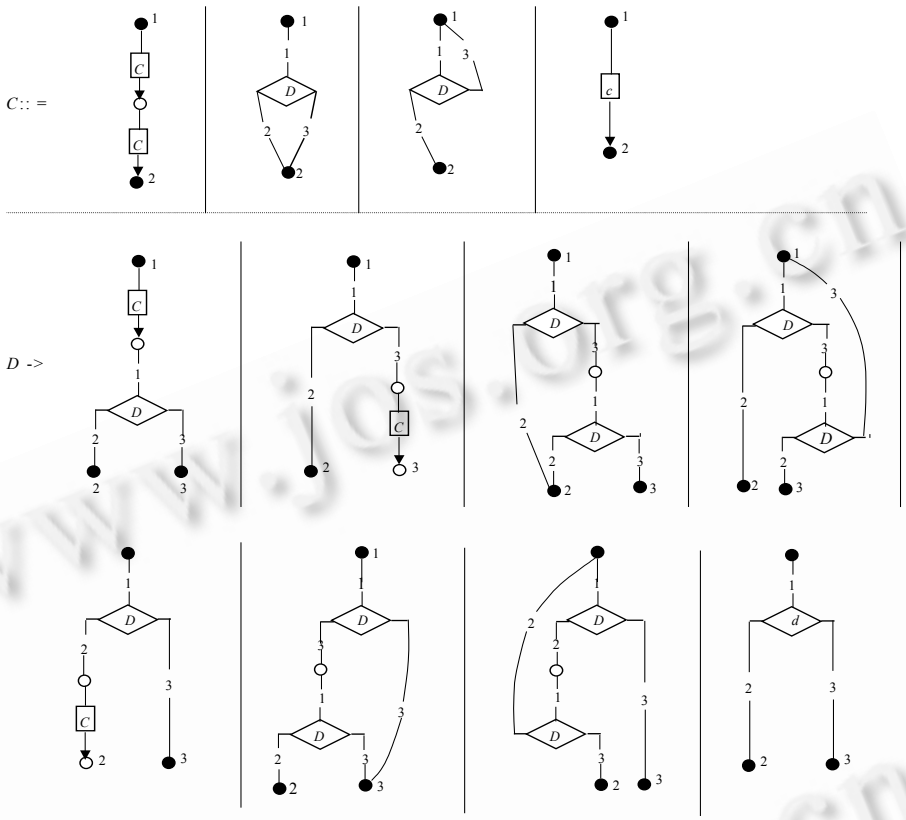

Fig.3 Productions of HR grammar G_{c2} for control-flow graphs

3.3 Data-Flow graphs

In this section we present a graph-grammar-based representation for DFG (data-flow graphs) to capture the data parallelism not described by control-flow graphs. A node y conflicts with node x if either x or y shares access to a common memory location and at least one of which is a "write" operation[13]. y is said to be data dependent on x if y accesses a data item modified (written) by x (denoted as $x\ \delta_d\ y$). The data dependence graph $DFG = (V_D, E_D)$ is defined as the directed graph such that V_D is the set of tasks and, for $x, y \in V_D$, $(x,y) \in E_D$ if $x\ \delta_d\ y$. There are two distinguished nodes called START and STOP in the DFG. START precedes all other nodes and STOP succeeds all other nodes. Edges in the DFG correspond to the precedence constraints that exist between tasks. The IN directive declares input variables for a node, while the OUT directive declares output variables. Input/output directives must be provided for all nodes in the program.

Given a set of n tasks (nodes) V_D, a set of input items (variables) I_D, and if the input/output directives are provided for all nodes $x \in V_D$, i.e. $IN(x) \in I_D$ and $OUT(x) \in I_D$, the DFG can be generated by an edNCE grammar $G_{c1} = (\{S, X, \#\}, \{*\}, I_D, P, C, S)$, where P contains a production called Node Production of x (NP_x) for each node $x \in V_D$. The right hand side of the NP_x contains a terminal node x_1 labeled # and an unterminal node x_2 labeled X with edges labeled m, $m \in OUT(x)$, between them, and its connection relations include (#, m/m, x_1, in) for each $m \in IN(x)$ and (#, m/m, x_2, in) for each $m \in I_D - OUT(x)$. In addition, P contains two productions for generating START node

and STOP node of the DFG.

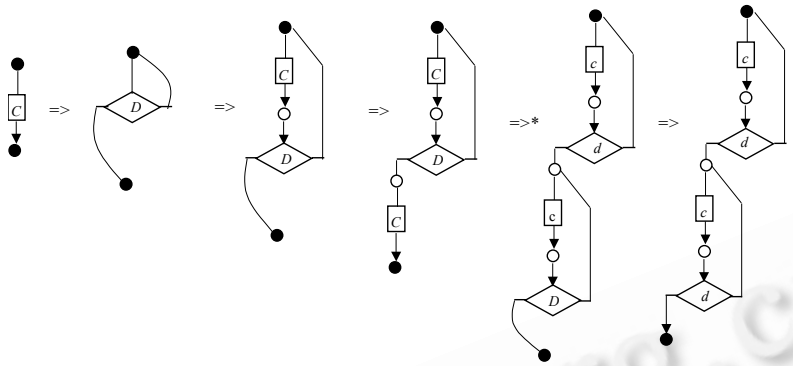

Fig.4 A derivation of HR grammar G_{c2} for control-flow graphs

For example, In Fig.5(a), four nodes v_1, v_2, v_3 and v_4 with the input/output directives are provided such that $IN(1)=\{a\}$, $OUT(1)=\{a, c\}$, $IN(2)=OUT(2)=\{a, b\}$, $IN(3)=\{b, c\}$, $OUT(3)=\{c\}$, $IN(4)=OUT(4)=\{a\}$. Note that the two black nodes are the START node and STOP node. These nodes can execute in an order of v_1->v_2->v_3->v_4 or v_2->v_1->v_4->v_3 or v_4->v_3->v_2->v_1, resulting in the DFGs shown in Fig.5(b). There are twenty other feasible orders. The corresponding edNCE grammar is $G_{d1}=(\{S, X, \#\}, \{*\}, \{a, b, c\}, P, C, S)$, where the productions in P are given in Fig.5(c). Note that the unlabeled nodes in the figure have label #, and $I_D=\{a, b, c\}$. For brevity sake, edges from the same source to the same target are combined into one edge whose label set is the union of the label sets of the original edges.

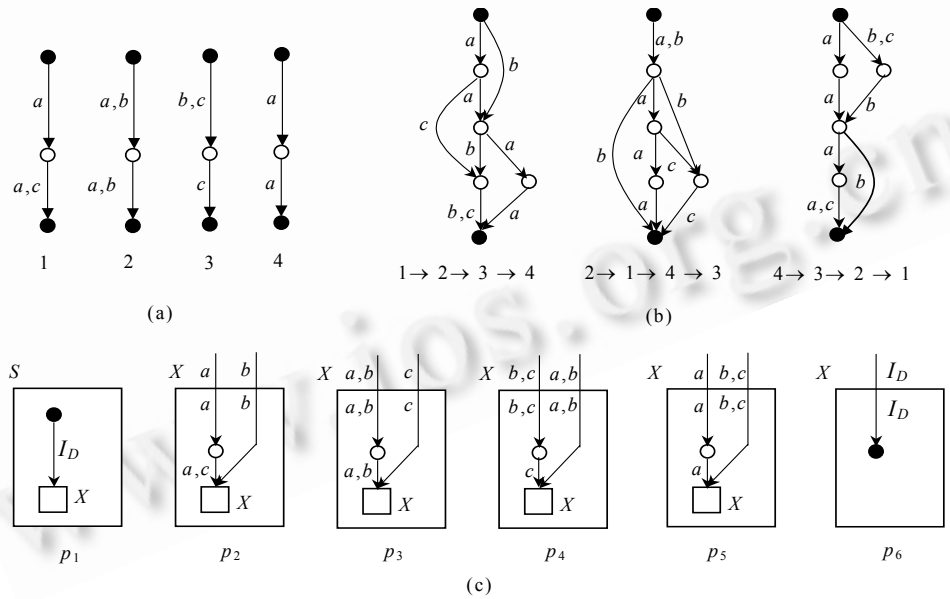

Fig.5 DFGs and the corresponding edNCE

A derivation example of $L(G_{d1})$ is shown in Fig.6 where p_1, p_2, p_3, p_4, p_5 and p_6, are applied in sequence, and the resulting CDF is exactly the one shown in Fig.5(b) with task order v_1->v_2->v_3->v_4.

Combing partial DFGs instead of incrementally adding a node in each step will accelerate the process of constructing the whole DFG. Intuitively, in order to combine two partial DFGs, we consider the list of variables written by the first DFG and compare it against the list of variables referred to in the second DFG. If we find out

that a variable referred to in the second DFG is written in the first DFG, we look for the node that modifies the variable list in the first DFG, and connect all nodes that refer to the variable in the second DFG to that node. If a variable is written in the first DFG and not written in the second DFG, the last node to write to the variable is connected to the STOP node of the second DFG. Similarly, if a variable is referred to in the second DFG but not written in the first DFG, we connect all nodes that refer to the variable in the second DFG to the START node of the first DFG. Finally, we free the STOP node of the first DFG and the START node of the second DFG. For example, in Fig.7, some of the three partial DFGs, G_1, G_2 and G_3, with $IN(G_1) = OUT(G_1) = \{a, b, c\}$, $IN(G_2) = \{a\}$, $OUT(G_2) = \{a, c\}$, $IN(G_3) = OUT(G_3) = \{a, b\}$, can be combined into a larger partial DFGs with respect to the order of G_1->G_2, G_2->G_3 or G_1->G_2->G_3.

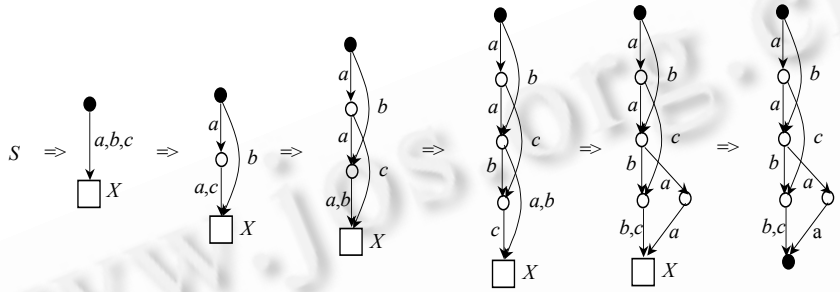

Fig.6 A derivation of G_{d1}

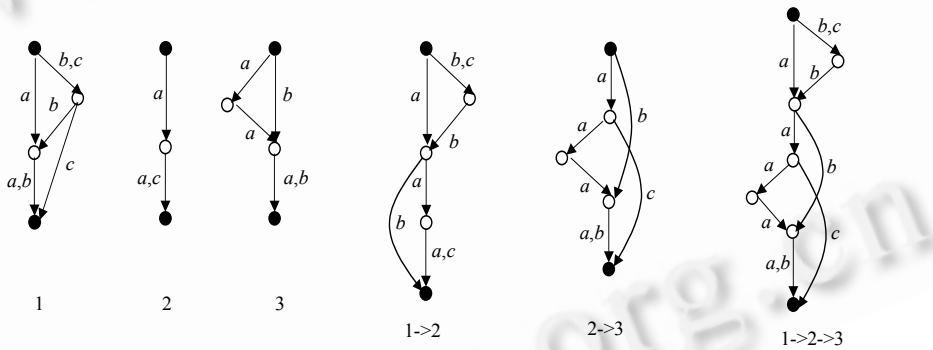

Fig.7 Data dependency graph

If we break each DFG into multiple partitions each of which has only one node, and construct a production for each node by the way we have mentioned, then applying these productions will also generate the same partial DCFs. However, this process will be time-consuming, especially when some partial DCFs occur frequently in the target DCF. Thus, we can construct a production for each partial DCF in the way similar to constructing a production for a single node. For instance, Fig.8 shows the three productions corresponding to the four partial DCFs in Fig.7.

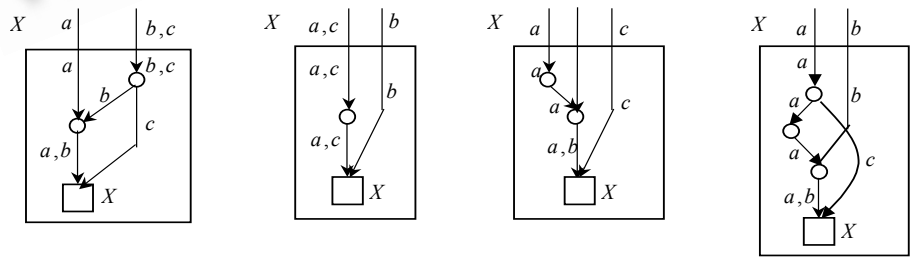

Fig.8 Productions of a partial DFG

3.4 CDG (control- and data-flow graphs)

To exploit available task and data parallelism for an application, we require a program representation that captures information about both control-flow and data-flow parallelism. During the last several years, many compiling techniques have focused on loop parallelism, e.g. vectorization, loop interchange, and loop blocking. However, nonloop/functional parallelism is more difficult to detect, package, schedule, or even express in a high-level language, and it has increasingly been the target of multithreading and instruction-level parallelism[14]. Thus, in this section we concentrate on functional parallelism instead of loop parallelization.

The parallelism extracted from the control flow graph can be embedded into the data dependence graph, thus the graph grammar for CFGs and DFGs should be combined. In Ref.[15] acyclic CFG is transformed to CDG (control dependence graph), which can be constructed from a postdominaotr tree, to facilitate the analysis and extraction of functional parallelism. We introduce the concept of CDG as follows.

Node y postdominates node x, denoted by $y\ \Delta_p\ x$, iff every path from x to STOP (not including x) contains y. Node y is control dependence on node x with label $x \to a$ ((x, a) is an arc in CFG), denoted by $x\ \delta_c\ y$, iff (1) $y \sim\Delta_p\ x$ and; (2) there exists a nonull path $P=\langle x,a,\ldots,y\rangle$ such that for any $z\ P$ (excluding x and y), $y\ \Delta_p\ z$. The control dependence graph CDF of a CFG is defined as the directed graph with labeled arcs, $CDF=(V,E)$ such that V is the same as that of CFG and $(x,y)\in CE$ with label $x\to a$ iff $x\ \delta_c\ y$ with label $x\to a$. CDG can be built form CFG using the postdominance tree[16].

An edNCE grammar G_{c3} can be used to generate CDGs. $G_{c3}=(\{S, X, \#\}, \{\#\}, \{*\}, P, C, S)$, where productions in P is given in Fig.9.

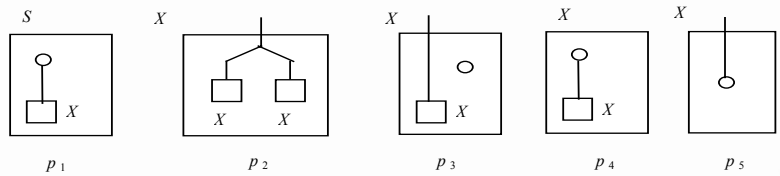

Fig.9 Productions of G_{c3}

A derivation example of $L(G_{c3})$ is given in Fig.10 where productions $p_1, p_4, p_4, p_2, p_3, \{p_3, p_3, p_3\}$ and $\{p_3, p_3\}$ are applied in sequence.

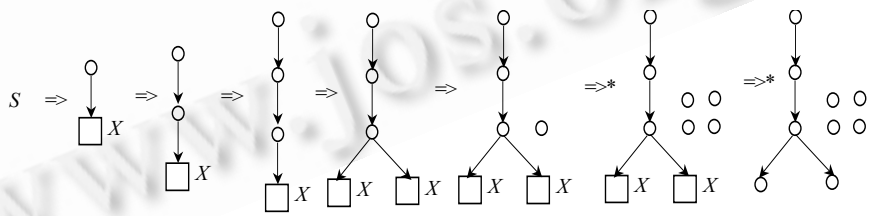

Fig.10 A derivation of G_{c3}

We can further design an edNCE grammar for generating a kind of graphs named Functional Control/Data Graphs (FCDG) which are more powerful than CDGs because a FCDG is capable of combining a CFG and a DFG into one single graph without loss of the information about data dependency and control dependence. Let G_{f1} be the edNCE grammar that is obtained from combining G_{c3} and G_{d1} above by adding most productions of the two grammars and making some minor modification, which is indicated in Fig.11. Productions p_1, p_2, p_3, p_4 account for building all ancestor-indexed trees (with edges from an ancestor to its offspring) such that there is an additional edge from each node to any of its ancestors, thus preserve the control-flow dependency relations. Productions $p_6, p_7,$

p_8 and p_9 correspond to four computation nodes (for instance, a procedure or a clause in the program language), which is used to derive the data-control dependency relations. Finally, production p_5 is used to generate the STOP node, note that STRAT node and STOP node are represented by a block circle in the figures.

A derivation example of $L(G_{f1})$ is given in Fig.12 where productions are p_1, p_4, p_6, p_4, p_7, p_2, p_8, $\{p_5, p_4\}$, p_9 and p_5 are applied in sequence. Note that the target graph may have multiple STOP nodes, whereas only one START node will exist in the final graph.

G_{f1}, G_{d1} and G_{c3} are all boundary NLC (NCE) graph grammars in which no two nodes with a nonterminal label are connected by an edge in the right hand side of productions and in the initial graph. An important feature of these grammars is: in a derivation, one obtains a sentential form with nodes x and y that are not connected by an edge, then whenever will happen later to those nodes in the derivation, no node descending from x will ever be later connected to a node in the derivation, thus ensuring that the grammar is confluent.

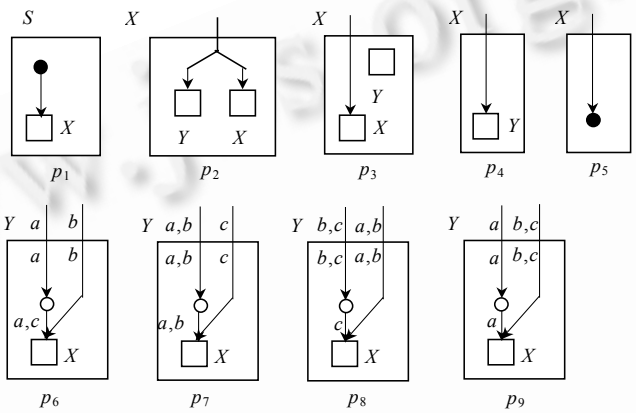

Fig.11 Productions of G_{c3}

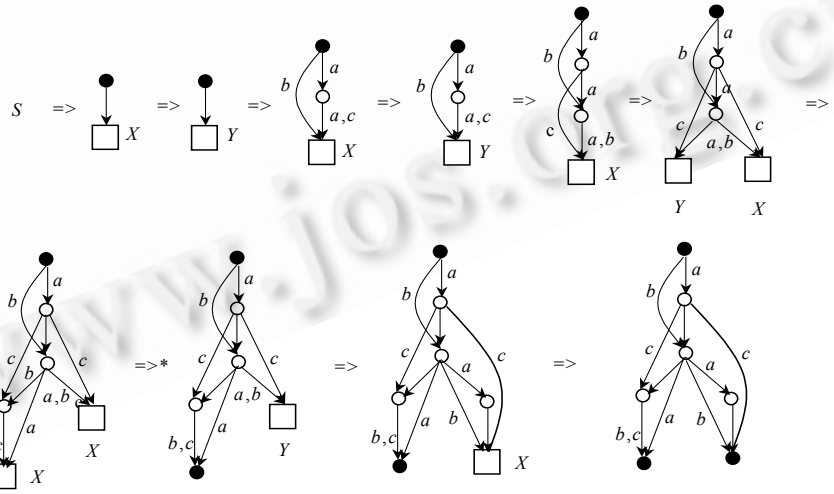

Fig.12 A derivation of G_{f1}

3.5 Advanced graph representation for dependency

For TIGs, the standard approaches using the traditional graphs have several significant shortcomings while partitioning the nodes into equally weighted sets[1]. In particular, they suffer from the limitations due to lack of the expressibility in the standard model. A bipartite graph model for describing matrix vector multiplication is proposed

in Ref.[1]. It is a special type of graph in which the vertices are divided into two disjoint subsets such that no edges connect two vertices in the same subset. It is most useful when the initial tasks are logically distinct from the final tasks, and has better representation capacity than the standard model does.

We just consider the complete bipartite graph here. The productions for general bipartite graphs can be obtained by similar methods. Let $K_{m,n}$ be the undirected complete bipartie graph on m and n nodes, i.e. the graph (V, E, λ) such that $V = \{u_1, \ldots, u_m, v_1, \ldots, v_n\}$, $E = \{(u_i, *, v_j), (v_j, *, u_i)\ 1 \le i \le m, 1 \le j \le n\}$, and $\lambda(x) = \#$ for every $x \in V$. The edNCE grammar $GK_{m,n}$ with the five productions shown in Fig.13 generates all the graph $K_{m,n}$ with $m,n \ge 1$. The connection relation C_1 of p_1 is empty, the connection relation C_2 of p_2, C_3 of p_3 and C_5 of P_5 are the same, i.e. $\{(\#, */*, X, in), (\#, */*, X, out)\}$, while C_4 of P_4 is $\{(\#, */*, Y, in), (\#, */*, Y, out), (\#, */*, \#, in), (\#, */*, \#, out)\}$.

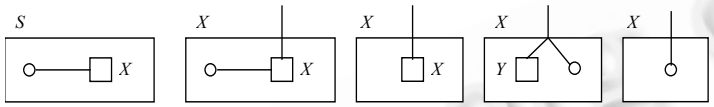

Fig.13 Productions of $GK_{m,n}$

A more elegant model is the hypergraph model. A hypergraph is a generalization of the graph in which edges (hyperedges) can have more than two vertices. It has broader applicability than the standard model since it correctly describes the minimal communication volume. It is more expressive than the standard model because it can encode problems in with unsymmetric dependencies. Figure 14 shows the two-way rowwise decomposition of a sample structurally nonsymmetric matrix on the left and the corresponding bipartioning of its associated graph depicted in Ref.[2], where the hyperedges crossing the processors represent the dependency between the nodes assigned to different processors. As we have discussed, hyperedge replacement graph grammars deal efficiently with hypergraphs, and choosing these grammars is a natural choice for achieving better expressibility and broader applicability.

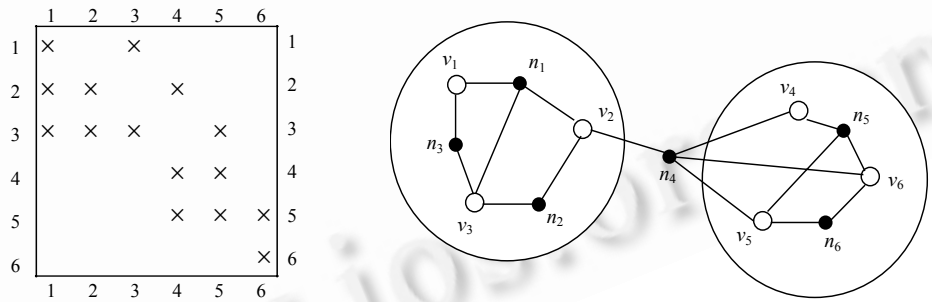

Fig.14 An example of hypergraph for matrix decomposition

4 Discussion

As we have mentioned, traditional dependency graphs cannot capture the whole structures of the dependency graphs. For expressing the intricate dependency constraints among up to thousands of nodes and deducting some crucial properties by analyzing the structures of these graphs, traditional graphs have intrinsic weakness because they cannot be used to obtain structural information concerning with different components in the graphs by applying some systematic methods. Furthermore, visual environments and visual tools may require another more complex method to facilitate the manipulation of dependency graphs. By introducing the graph grammar method, we can figure out the semantic characteristics of the dependency graphs and deduct some other properties by applying a formal deduction method, which is impossible by just analyzing the traditional node and edge connecting graphs.

In this paper we use graph grammars to represent all kinds of dependency graphs, thus providing a united model for these graphs. Graph grammars in general provide an intuitive description for the manipulation of graphs and graphical structures in various kinds of software and systems such as parallel computing systems relying heavily on parallelism identification. In most of these cases, grammars allow one to give a formal graphical specification which can be executed as far as corresponding graph grammar tools are available.

Formally, let Σ be an alphabet of node labels and Γ an alphabet of edge labels. A graph over Σ and Γ is a tuple $H = (V, E, \lambda)$, where V is the finite set of nodes, $E \subseteq \{(v, \gamma, w) \mid v, w \in V, v \neq w, \gamma \in \Gamma\}$ is the set of edges, and λ: V -> Σ is the node labeling function. Take node replacement as an example. The derivation process of C-edNCE is just like that of the general context-free string grammars, and derivation tree whose vertices labeled by production (i.e. root is labeled X -> (D, C) will correspond to the derivations starting from X. The resulting graphs can be yielded in a top-down fashion such as using the associated leftmost derivations with derivation trees.

Particularly, in this paper, after the derivation of graph grammars for extracting parallelism, theoretical results for graph grammars are useful for analysis, correctness and consistency proofs of such problem. At present, the so-called DPO (double-pushout) approach and SPO (single-pushout) approach are the most used algebraic approaches to formulate rewriting steps by gluing constructions. DPO is modeled indeed by two gluing diagrams in the category and total graph morphisms; while SPO defines a basic derivation step as a single pushout in the category of graphs and partial graph morphisms. Note that the algebraic approaches are characterized by the use of categorical notions for the very basic definitions of graph transformation rules, of match, and of rule applications. A detailed description of these two methods can be found in Ref.[4].

Compared with traditional models in which the information of each node and each edge must be stored, our graph grammar based method needs to store the productions only, and these productions can be used to generate all kinds of dependency graphs with various sizes while the structures of the graphs are still maintained.

5 Conclusions

In the domain of parallel and distributed computing, especially to the problem of parallelism extraction such as dependency detection and expression, there exists no satisfactory results based on the solid theoretical background such as graph grammar theory. Thus, in this paper, based on graph grammars, i.e. confluent edNCEs and HRs, we present some novel schemes to express data-flow graphs, control-flow graphs, combined control-data-flow graphs, bipartite graphs and hyperedge graphs. Due to the fact that graphs are a very natural way for explaining complex situations at an intuitive level, our results contribute to the design and implementation of parallel compilers or parallel languages in an easier and more efficient way.

References:

[1] Bruce H, Tamara GK. Graph partitioning models for parallel computing. Parallel Computing, 2000,26(12):1519~1534.

[2] Catalyurek VC, Aykanat C. Hypergraph-Partitioning based decomposition for parallel sparse-matrix vector multiplication. IEEE Trans. on Parallel and Distributed Systems, 1999,10(7):673~693.

[3] Zhang K, Ma W. Graphical assistance in parallel program development. In: Proc. of the 10th IEEE Int'l Symp. on Visual Languages. St. Louis, 1994. 168~170.

[4] Corradini A, Montanari U, Rossi F, Ehrig H, Heckel R, Loewe M. Algebraic approaches to graph transformation. In: Rozenberg G, ed. Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1. Foundations: World Scientific Publishing, 1997. 163~245.

[5] Nagl M. A tutorial and bibliographical survey of graph grammars. In: Claus V, Ehrig H, Rozenberg G, eds. Graph Grammars. Lecture Notes in Computer Science 73, Berlin: Springer-Verlag, 1980. 70~126.

[6] Engelfrief J, Rozenberg G. Node replacement graph grammars. In: Rozenberg G, ed. Handbook of Grpah Grammars and Computing by Graph Transformation, Vol.1. Foundations: World Scientific Publishing, 1997. 1~94.

[7] Drewes F, Kreowski HJ, Habel A. Hyperedge replacement graph grammars. In: Rozenberg G, ed. Handbook of Grpah Grammars and Computing by Graph Transformation, Vol.1. Foundations: World Scientific Publishing, 1997. 95~162.

[8] Foster I, Chandy KM. Fortran M: A language for modular parallel programming. Journal of Parallel and Distributed Computing, 1995,26(1):24~35.

[9] Foster I, Avalani B, Choudhary A, Xu M. A compilation system that integrates high performance Fortran and Fortran M. In: Proc. of the Scalable High Performance Computing Conf. Los Alamitos: IEEE Computer Society Press, 1994. 293~300.

[10] Girkar M, Polychronopoulos CD. Automatic extraction of functional parallelism from ordinary programs. IEEE Trans. on Parallel and Distributed Systems, 1992,3(2):166~178.

[11] Chandy KM, Manohar R, Massingill BL, Meiron DI. Integrating task and data parallelism with the group communication archetype. In: Proc. of the 9th Int'l Parallel Processing Symp. Santa Barbara: IEEE Computer Society Press, 1995. 724~733.

[12] Ramaswamy S, Sapatnekar S, Banerjee P. A framework for exploiting task and data parallelism on distributed memory multicomputers. IEEE Trans. on Parallel and Distributed Systems, 1997,8(11):1098~1116.

[13] Waite W, Goos G. Compiler Construction. Berlin: Springer-Verlag, 1996. 1~60.

[14] Gupta R, Pandeb S, Psarris K, Sarkar V. Compilation techniques for parallel systems. Parallel Computing, 1999,25(13):1741~1783.

[15] Girkar M, Constantine D. Extracting task-level parallelism. ACM Trans. on Programming Languages and Systems, 1995,17(4): 600~634.

[16] Ferrante J, Ottenstein Warren JD. The program dependence graph and its use in optimization. ACM Trans. on Programming Languages and Systems, 1987,9(3):319~349.

****

2005 年智能计算及其应用国际研讨会通知

(International Symposium on Intelligence Computation, ISICA'2005)

为了适应我国高技术计划"智能计算的理论与应用研究"的需要,推动我国深入开展智能计算的研究,加强国际交流与合作,在国家宇航学会、教育部等资助下,中国地质大学将举办一次高水平的智能计算国际研讨会(International Symposium on Intelligence Computation,简称 ISICA'2005),会议定于 2005 年 4 月 4 日~6 日在中国地质大学举行,会议将出版论文集并送检索机构检索,欲参加研讨会者请尽快与我们联系.

一、主要议题

神经网络、遗传算法、遗传程序设计、基因表达式程序设计等在科学数据分析中的应用、影像库的自动分析、航天器中的演化硬件、智能 GIS、多目标优化、数字和组合优化、模糊系统、动态优化、机器学习、人工免疫系统、生物信息学、蚁群方法、粒子群方法、演化数据挖掘

二、会议特色

会议邀请了演化计算方面各主要分支的国际知名专家来参加此次会议,包括 Hans-Paul Schwefel, (Faculty of Computer Science, Dortmund University), Kalyanmoy Deb (Indian Institute of Technology, India), David B.Fogel (Natural Selection Inc USA), Jason Lohn (NASA Ames Research Center, USA), Adiran Stoica (NASA JPL,USA), Tughrul Arslan (University of Edinburgh, UK), Garry Greenwood (Portland State University, USA), Zbigniew Michalewicz (Nutech Solutions, Inc. USA Poland), Bob Mckay (Australian Defence Force Academy, Australia), Hugo de Garis (Utah State University, USA)

三、联系方式

联系地址: 中国地质大学计算机科学与技术系 邮编: 430074 联系人: 蔡之华 李晖 李振华

联系电话: 027-67883713 电子信箱: jsjb@cug.edu.cn 或 zhcai@cug.edu.cn