From Proceedings of[1]

The 5th International Conference on Software Engineering and Knowledge Engineering

June 14-15, 1993
San Fransico Bay, uSA

Formalism for Hierarchical Organization and Flexible Abstraction of Program Knowledge

Javed I. Khan and Isao Miyamoto
Software Engineering Research Laboratory
University of Hawaii at Manoa
javed@wiliki.eng.hawaii.edu

Abstract

This paper describes a formalism for program **knowledge structure** which can support *hierarchical* representation of conventional programs for the purpose of *flexible* program abstraction. The resulting abstraction system can automatically extract program knowledge and formulate concise and abstract description of the program by organizing its instruction and data spaces as well as their inter-relationships. Unlike most other abstraction systems, this formalism allows flexibility in the selection of abstraction level, without enforcing any pre-engineered abstraction format.

1 Introduction

The notion of hierarchy has been used as a method for knowledge organization from the ancient time in the "*reverse engineering*" of nature. In Veda, the classification of matters, in biology, the taxonomy of flora and fauna, in geology, the classification of rocks are just few examples which show how the hierarchical organization of accumulated concepts have helped the human learning and gradual understanding of natural processes. It is no wonder, in the *reverse engineering* of software codes, a similar attempt to formulate the knowledge and concepts associated with software processes in a hierarchical organization, is expected to facilitate understanding of computer programs. Specially when, hierarchies have been used extensively in various sophistications in the forward engineering [3].

Organization of the overall knowledge space associated with the code is critical in creating and prudently guiding *expectation* (to know what to learn, which is a precondition for any learning) in the context of computer programs re-learning. In fact, such organization is prerequisite for both the plan as well as code knowledge driven approaches.

In this paper we would like to present a symmetrical function and data hierarchical decomposition based formalism for the organization of program knowledge through abstraction. It focuses on the information that can be obtained or derived from code. Our approach has two important distinction from most of the previous approaches in program abstraction [1,3,5]. First, we would like to emphasize, not only on the function space but also on the understanding and organization of data space as well. Secondly, the level of abstraction is not pre-engineered into the system, rather we allow the human expert to flexibly select appropriate abstraction level.

Sections 2 and 3 respectively describe the program model which provides an intuitive latitude bound of the program information space, and the resulting program *knowledge structure*. Finally, section 4 provides the scheme which extracts the program knowledge from source code and organizes them into the *knowledge structure* in the form of concept hierarchies leading to flexible abstraction.

2 Program Information Space

Our program model is conceptually rooted into the very von Neumann architecture, where, the aggregate information flow through a processing element is distinguished into two principal flows: *data stream* and *instruction stream*. As a direct consequence, all conventional programs, irrespective of their sophistication, may be viewed as consisting of two principal entities;(i) *function* and (ii) *data item.*

A *program* P thus, can be viewed as a pair of principal entities $< F(P), D(P) >$, where F(P) is a finite set which includes all the instructions (or functions) stated in the program. And D(P) is the set of all the data items stated in P and accessed by the members of F(P).

Two entities can be inter related in at most three different manners. For our case; these are relationships between (iii) two functions, (iv) two data items, and (v) a function and a data item. Any program knowledge belongs to either of these five *concepts*. Each of these concepts is defined by its associated attribute (or "information") which is generally either extracted from the source code or deduced by the reverse engineering process. Below we formally define the concept space of a program P:

Definition 1: A *function concept* in program P is defined as an associated set $<f_i,I_f>$ where, F(P) is the set of all the functions(that which affects the program execution through instruction stream) of program P, and $f_i \in F(P)$. Each of the concepts in F(P) is bounded by a set of attributes A_F. I_f refers to function information or the attribute values of A_F in f_i.

Definition 2: Analogously, a *data item concept* in program P is defined as an associated set $<d_i,I_d>$ where, D(P) is the set of all the data objects (that which affects or is being affected by program execution through data stream) of program P, and $d_i \in D(P)$. Each of the concepts in D(P) is bounded by a set of attributes A_D. I_d refers to the data information or the attribute values of A_D in d_i.

as mentioned, in the folds of P, there lies three relational concept space generated from the product of the principal entities of P.

Definition 3: The concept of all the possible relations between two *functions* is a member of the set $\Sigma(P)$ of program P, and is specified as $< f_i, f_j, I_\Sigma >$. Each member of $\Sigma(P)$ has a corresponding $< f_i, f_j > \in F(P) \times F(P)$. I_Σ refers to the attribute values of the attribute set A_Σ that defines the concept.

Definition 4: The concept of all the possible relations between two *data items* is a member of the set $\Phi(P)$ of program P, and is specified as $< d_i, d_j, I_\Phi >$. Each member of $\Phi(P)$ has a corresponding $< d_i, d_j > \in D(P) \times D(P)$. I_Φ refers to the attribute values of the attribute set A_Φ that defines the concept.

Definition 5: The concept of all the possible relations between a *function* and a *data item* is a member of the set $\Psi(P)$ of program P, and is specified as $< f_i, d_j, I_\Psi >$, Each member in $\Psi(P)$ has a corresponding $< f_i, d_j > \in F(P) \times D(P)$. I_Ψ refers to the attribute values of the attribute set A_Ψ that defines the concept.

The total concept space associated with a program is defined as:

$$P = \{F, D, \Sigma, \Phi, \Psi\}$$

Similarly the total information space associated with a program P is: $I_P = \{I_F, I_D, I_\Sigma, I_\Phi, I_\Psi\}$

Any knowledge associated with code based program understanding belongs to either of the five sub-spaces of P. Below we will state that in the form of theorem. The proof is evident from the definitions of the concept classes.

Theorem 1*: Any information, which is explicit in the code, or can be sufficiently deduced from the code, belongs to the five information spaces of P.*

Both the auto-relation concepts Σ, and Φ can be further subdivided into temporal and spatial relations. For instance, Σ can be divided into two major components $\Sigma(P) = \{\Sigma_t(P), \Sigma_s(P)\}$ where, $\Sigma_t(P)$ refers to the dependency related to the time sequence of accessing the instruction stream between the function objects. On the other hand $\Sigma_s(P)$ refers to the dependency related to composition of function entities. As an example, control flow information lies in $\Sigma_t(P)$. In a non-self-recursive program explicit syntactic control information C $\subseteq F(P) \times F(P) - \{< p, p > | \, p \in F(P)\}$. For a relation $< u, v > \in C$, u represents the predecessor function and v represents the successor function in a control path. Generally the *difference space* $\{F(P) \times F(P) - \{< v, v > | \, v \in F(P)\}\} - C \neq \emptyset$. The relation ship between a subroutine and its constituent instructions is compositional and is an example of $\Sigma_s(P)$. The concepts regarding the relations among data items can be similarly classified as $\Phi(P) = \{\Phi_t(P), \Phi_s(P)\}$ where, the temporal relation $\Phi_t(P)$ refers to concept among the modify/write/read dependencies among the data items. And $\Phi_s(P)$ refers to the spatial type subsumption relation as found among structure/union, record/field etc. Use/def dependency type dependency on the other hand belongs to $\Psi(P)$. Only a part of the total concept space is generally *explicit* in a program code. For example, the *control flow* between two consecutive statements is generally explicit, but the *control flow* between any two statements may not be explicit and obvious. The objective of various dependency analysis in software engineering can be broadly categorized as the derivation of these implicit informations from the explicit components.

3 GHPM Knowledge Structure

On the basis of above program model, the following semantic network based formalism called Generic Hierarchical Program Model (GHPM) has been defined using the meta language MERA [2] to represent and process program knowledge. The semantic network is built up of entity-nodes and relation- edges such that, $GHPM_{entity} = \{F, D\}$, and $GHPM_{relation} = \{\Sigma, \Phi, \Psi\}$. The information associated with the program concepts is defined and stored as *attribute: value* pair in the defining slots associated with the entity and relation objects. Fig-3 shows the definition of the formalism.

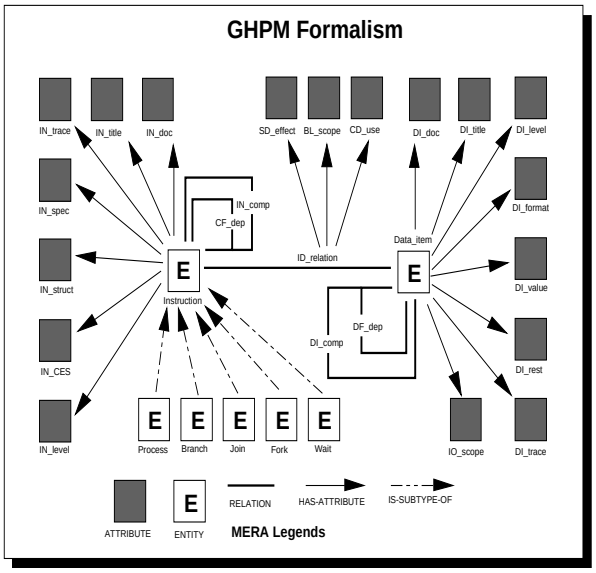

Fig-3

GHPM uses two relations to attain space efficiency while representing I_Σ. The spatial and temporal relations are respectively named as "IN_comp" and "CF_dep". Petri-net like "FORK" and "WAIT" are used to represent concurrent paths in "CF_dep". Symmetrical to functional relation, GHPM uses two different relations "DI_comp" and "DF_dep" to represents the spatial and temporal part of I_Φ.

GHPM maintains three attributes of I_Ψ or relation "ID_relation". These are, (i) the data dependency among the data items and functional blocks (SD_effect). (ii) the role of a data item in a function (CD_use={C_IN, C_PT, NULL}), and (iii) the block relative scope boundaries of the data items (BL_scope={INTERNAL,EXTERNAL}).

4 Abstraction by Hierarchies

A monolithic source code (shortly, we will show the natural extension to non-monolithic multi-module programs) provides a base model at the abstraction level, which is generally representative of the level of the programming language. The base model consists of the function and data entities. Our system accepts this base model as input and gradually constructs abstract or

composite data and function entities in a bottom up fashion and organizes them into two hierarchies named as (i) *Hierarchical Data Model* (HDM), and (ii) *Hierarchical Function Model* (HDM).

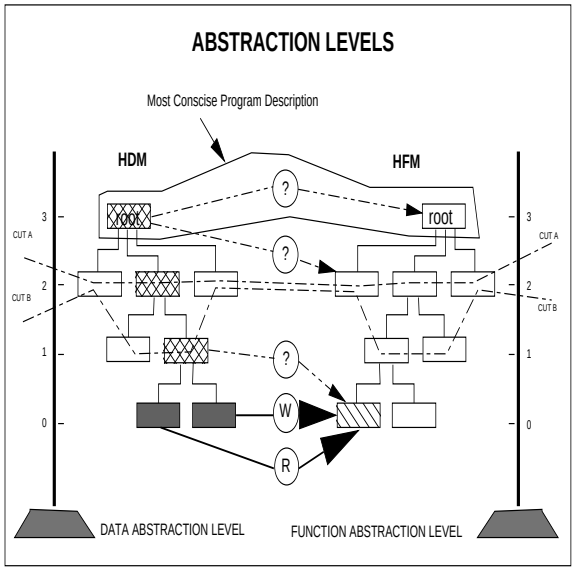

Fig-2

Figure-2 shows the process for an arbitrary computer program P which is made up of 5 statements and 4 data items. In this figure the left and the right trees respectively denote hypothetical configurations of the HDM and HFM of P. The leaves nodes of the hierarchies denotes the concrete (those which are not abstract) entities of P and other nodes are generated abstract or composite entities. Let its statements are $\{f_1(\delta_1), f_2(\delta_2), f_3(\delta_3) f_4(\delta_4), f_5(\delta_5)\}$, where, δ_i refers to the data set associated with each of the statements. Then $\bigcup_{i=1}^{5} \delta_i = D(P)$. And let $D(P) = \{d_i \mid 1 \le i \le 5\}$. The atomic (base) entities receive attribute values directly from the corresponding atomic statements present in the source code. The abstract entities (such as the "?" marked relations) generate their slot values during the abstraction process through a set of appropriate grammars. The abstraction process can be viewed as the systematic compression of the sets F(G) and D(G) at each level by re-organizing or rediscovering implicit spatial relations in $\Sigma_s(P)$ and $\Phi_s(P)$. Any complete cut across the hierarchies forms a complete description of the program. The average height of the cut intuitively provides a measure of abstraction of that particular representation. (For example cut A is more abstract than cut B). The hierarchical representation makes it possible to flexibly transcend between different abstraction levels by gradually moving down and up the hierarchies. At each level, the abstraction grammar provides the deduction ability of the inter-relations and entity attributes. At the root of the two hierarchies, the concepts in $\Sigma_{root}(P)$ and $\Phi_{root}(P)$ becomes empty. Thus, the specification of the two root entities and their interrelations, $\{F_{root}, D_{root}, \Psi_{root}\}$ becomes the ultimate abstract description of the program P.

Multi-module (non monolithic) programs can be viewed as partially abstracted program, where, each of the sub-routine is a priori-abstracted module in HDM. Thus, multi-module programs only provide an advanced starting point where some of abstraction already has been done by the programmer. The function and data hierarchies are generated using a technique based on the theory of proper decomposition [3]. The details of the prototype algorithm and specification generation rules can be found in [4].

5 Complexity

Below, we present an informal estimate of space complexity of the representation formalism. Rigorous estimate will require the analysis of detail algorithms which is beyond the scope of this paper.

Theorem 2: *For a input problem with M basis functions and N basis data items the total space requirement is O(MN).*

Proof: The base model will require M function blocks and N data items and at most MN relation entities. The abstract hierarchical model will have at most 2M function blocks, 2N data blocks and thus at most 4MN relations. Thus, the full abstraction space is 2M+2N+4MN. Thus, the space requirement is O(MN).

The target system is intended to generate program abstract from a large (1000-2500 line) but not necessarily complex programs in a reasonable time and space. A proto-type abstraction system called *Hierarchical Program Abstraction System* (HPAS) for COBOL has already been developed and is currently under testing with real sized programs.

6 References

[1] V. R. Basili, S. K. Abd-El-Hafiz, G. Caldiera, Towards Automated Support for Extraction of Reusable Components, Proceeding of the IEEE Software Maintenance Conference, Sorento, 1991.

[2] MERA: Meta language for Software Engineering, Proc. Of the 4th Intl. Conf. on Software Engineering & Knowledge Engineering, June, 1992, Capri, Italy, pp495-502.

[3] J. E. Hartman, Automatic Control Understanding for Natural Programs, Ph.D. Dissertation, Univ. Of Texas at Austin, May 1991.

[4] Javed I. Khan, Hierarchical Program Abstraction System Design Documents, Technical Report 92-30/92-31, Software Engineering Research Lab, Univ. Of Hawaii, Sept, 92.

[5] M. Weiser, Program Slicing, IEEE Tran. On Software Engineering, v. SE-10, no.4, July 1984, pp352-357.