Published in the Proceedings of the
5th International Conference on Software Maintenance
ICSM '94, Victoria, Canada, 1994

DESIGN EXTRACTION BY ADIABATIC MULTI-PERSPECTIVE ABSTRACTION

Javed I. Khan

Department of Electrical Engineering
University of Hawaii at Manoa
Holmes 493, 2540 Dole Street
Honolulu, HI-96822
javed@wiliki.eng.hawaii.edu
(808)-956-7249
(808)-941-1399(FAX)

ABSTRACT

Design extraction of an unfamiliar system is a complex cognitive task. This paper presents an approach that can help human expert in exploring a large and complex code information space of an unfamiliar software. It provides her/him a platform to access the code information with **flexible**, fine and delicate control over volume and composition of the accessed information sub-space. The proposed approach integrates two forms of abstraction. First, it helps to comprehend **complexity** of the code information space by allowing explorer to investigate the system from numerous (combinatorial) coherent perspectives. In the second level, it helps to overcome **scale** of the information space by allowing explorer to compress or expand any composition of its sub-spaces. This new approach, named as **adiabatic multi-perspective** (AMP) approach to program abstraction, is founded on a **symmetrical dual hierarchical** (SDH) organization of code information space and a novel formalism for **abstract dependency analysis** (ADA), which is also one of the first formalism to perform complete program dependency analysis on abstract program models.

Design Extraction by Adiabatic Multi-Perspective Abstraction

Javed I. Khan[1]

Department of Electrical Engineering
University of Hawaii at Manoa, USA
javed@wiliki.eng.hawaii.edu

Abstract

Design extraction of an unfamiliar system is a complex cognitive task. This paper presents an approach that can help human expert in exploring a large and complex code information space of an unfamiliar software. It provides her/him a platform to access the code information with ***flexible****, fine and delicate control over volume and composition of the accessed information sub-space. The proposed approach integrates two forms of abstraction. First, it helps to comprehend* ***complexity*** *of the code information space by allowing explorer to investigate the system from numerous (combinatorial) coherent perspectives. In the second level, it helps to overcome* ***scale*** *of the information space by allowing explorer to compress or expand any composition of its sub-spaces. This new approach, named as* ***adiabatic multi-perspective*** *(AMP) approach to program abstraction, is founded on a* ***symmetrical dual hierarchical*** *(SDH) organization of code information space and a novel formalism for* ***abstract dependency analysis*** *(ADA), which is also one of the first formalism to perform complete program dependency analysis on abstract program models.*

1. Introduction

Design extraction of an unfamiliar system is a challenging cognitive task and lies at the fringe of current research in Knowledge Engineering. Such exploration, particularly in the case of software engineering, quite often becomes a formidable mental exercise, because of the **scale** and **complexity** of the unknown system.

Let us first consider the difficulty originating from the complexity. Classical reverse engineering uses various diagrams such as control-flow graph, petri-nets, call graph, slice [10,11,15,19], etc., to visualize a software system. In isolation, none of these views is sufficient to encapture the complexity of the overall system. However, collectively each helps at least in dividing the overall complexity of the underlying system. The ability to visualize from such multiple perspectives plays a key role in understanding complex systems. Cognitivically, it is a divide and conquer strategy to win over complexity. Therefore, a clean mechanism for abstraction in the sense of multi-perspective visualization of the target information space may be considered an essential part of any effective design extraction platform. However, one of the critical cognitive tasks performed by a human expert during the process of design extraction is to correlate the concepts across the perspectives. Therefore, the effectiveness of the multi-perspective visualization also depends on the **mutual coherency** across the perspectives.

A second level of difficulty in design comprehension arises from **scale**. Sheer scale can easily make even a relatively simple system (or a singly isolated perspective of a complex system) incomprehensible. In forward engineering, scale affects mostly quantitatively. Because, the target system is generally a machine. But, in reverse engineering, the purpose is to produce effective process perception in human expert. As a consequence, the size of the visual and mental information can seriously affect the quality and efficiency of human understanding. Therefore, a design extraction tool for real scale software systems should also provide abstraction mechanism in the sense of volumetric **reducibility** of the perspectives.

In essence, the effectiveness of a design extraction platform will critically depend on the **flexibility** and ease with which design information can be abstracted both in the sense of generating coherent perspectives and their volumetric control.

In the past years, researchers from diverse application objectives have proposed a significant number of approaches, which in various ways address the two essential aspects of design extraction. Such as Miyamoto et. al. [10,11] proposed about a dozen views (perspectives) to extend multi-perspective visualization beyond the classical few. However, in this and many other similar attempts [2,16,17], no serious consideration was made towards volumetric reducibility of these views. On the other hand, Basili et. al [1] proposed formal validation based approaches to create concise specification of a program mainly to facilitate debugging and testing during forward engineering. Howden [5] used a formal comment based approach to express a program in *condition:action* format. However, most of these formal approaches (also known as logical abstraction), compress a few highly sophisticated but rigid perspectives, mostly at a pre-engineerined abstraction level. In essence they lack flexibility, which is very important from design extraction point of view.

1 Part of this work has been conducted at and supported by the **Software Engineering and Research Laboratory (SERL)**, Information and Computer Science Department, UH.

More recently, a number of researchers proposed models based on function hierarchy [3,4,12,18,20]. Such as CIA by Chen, Nishimoto and Ramamoorthy [3], provides a formalism to understand macro model of program file and subroutine structures for C programs. A similar hierarchical approach RIGI, by Muller, Tilley et. al.[18] specifically targets large scale software systems. However, most of such hierarchical approaches are rather function-centric. They only provide some reducibility on some variant of function views, and do not organize or abstract data space. While, it is well known that in an unfamiliar code, bulk of the unfamiliar text is generally due to data items. Thus, in design extraction, the understanding inside the data space and their relation to function space is at least equally critical if not more. Most of these approaches provide no abstraction in data space.

An integrated platform, that could provide general as well as flexible reducibility over a wider range of constructible program views remained illusive because of two reasons. Firstly, due to the lack of similar decomposable organization of the data space of a program. And more importantly, because of the lack of any suitable formalism that could abstract the general dependency relating program and data [9]. In fact, many of the hierarchical approaches, such as RIGI[18], which claim handling of large scale software systems, actually manage only macro-aspects, a narrow range in overall scale spectrum. This range generally has only program-files and function-modules, with fairly straightforward data dependency involving mostly data-files. The lack of suitable abstract dependency analysis formalism restricts such approaches from widening the scale spectrum to finer entities involving instructions or even instruction segments, which are quite often connected by more complex data dependencies.

In this paper, I present a program information abstraction approach, which attempts to fill up this missing piece in reverse engineering research. The proposed formalism facilitates human exploration inside an unknown software system by employing divide and conquer strategy simultaneously over the complexity as well as on the scale of the target system. The approach is principally based on (i) organization of both function and data information space in the form of **two symmetric dual hierarchies (SDH)**, and (ii) the formalism for **abstract dependency analysis (ADA)**. The SDH organization allows efficient storage and access to any part or whole of the program information space with flexible reducibility. The ADA formalism connects the entities of the two hierarchies irrespective of their level of abstraction. This paper presents the combined approach and shows how it provides an opportunity to flexibly generate various program perspectives at pliable dimensions.

In the rest of this paper, this integrated approach will be referred to as **adiabatic multi-perspective[2] (AMP)** approach to abstraction. A platform which realizes the approach is named as **program information abstraction system (PIAS)**. Following section first outlines the central notion of program information abstraction as used in this work. Next, section 2 and 3 present the SDH organization. Section 4 and 5 present the ADA formalism. Finally, section 6 demonstrates application of this system in accessing, exploring and analyzing various program aspects with flexible abstraction.

2. Program abstraction

There exists a significant amount of work related to program abstraction. Surprisingly, it is very hard to find any clarification of this important concept in related publications. Therefore, I will briefly pause to outline the very notion of program abstraction as used in the approach.

Definition (program abstraction): ***Program abstraction*** *is a process by which a program is presented in a* ***concise*** *form, that facilitates program understanding by* ***emphasizing*** *the* ***significant*** *information of the target program.*

In this model, **information** refers to the code knowledge that can be obtained or deduced solely from the information present at the source code (target software). It is also assumed that sufficient contextual knowledge is available about the source code language(s). However, no expectation is made about the availability of programmer's domain knowledge on the application.

Compaction and **selection** are the two basic means of reducing information [8]. *Compaction* process uses all the components to generate the synthetic concise abstract. On the other hand, *selection* process presents only some specific components and hides the rest to generate the concise abstract.

The relative significance among various components of information is dependent on the intended use of the generated abstract. Therefore, instead of assuming or imposing any fixed model of relative significance on the various components of information, AMP approach is to provide a flexible interface so that the human user can flexibly specify/change the preference model, and generate corresponding abstract.

3. Information organization

The objective of SDH organization is to arrange the total information space in such a way, so that any component or a logical composition of it can be accessed, processed and abstracted (by compaction and selection) efficiently with convenience and flexibility.

The Components of Program Information: In AMP approach, a code (P) is viewed as a collection of two sets of principal entities; (i) *instruction* I(P), and (ii) *data* D(P). Considering spatial and temporal distinctions, two entities can be inter-related in at most five different manners. For this case these are; (iii) auto-relation between two instruction entities $\Sigma_s(P)$ in space, (iv) auto-relation between two instruction entities $\Sigma_t(P)$ in time, (v) auto-relation between two data entities $\Phi_s(P)$ in space, (vi) auto-relation between

2 The term *adiabatic* refers to the volumetric compressibility/expandability of any aspect of the program information space with minimum loss of dependency information.

two data entities Φ_t(P) in time, and finally (vii) inter relation between an instruction and a data entity Ψ(P). Thus a program information space is defined as:

$$P = \{I, D, \Sigma_t, \Sigma_s, \Phi_t, \Phi_t, \Psi\}$$

A complete entity relation model describing this seven component code information space has been presented in [7].

Information Structure: The above seven component information space is organized in the form of two symmetrical hierarchies involving the two principal entities (instruction, and data). A single monolithic source code (the natural extension to multi-module programs will be discussed shortly) provides a base-model. PIAS accepts the base-model as input and gradually constructs abstract data and function entities in a bottom up fashion and organizes them into two three-dimensional pyramidal structures named as (i) *Data Cube* (D-CUBE), and (ii) *Instruction Cube* (I-CUBE), and (iii) their inter-relations (LINK). These three components are together called as SDH organization.

Fig-1 shows the D-CUBE and I-CUBE for a simple target program P (**test.p**) that is made up of 7 statements (**s1, s2, s3, s4, s5, s6, s7**) denoted as set I(P) and 6 data items (**x, y, max, sum, stdin, stdout**) denoted as set D(P). These two hierarchies are structurally and conceptually symmetrical.

In Fig-1, the relations along the vertical dimension of I-CUBE represent the spatial auto-relations between the instructions and the horizontal dimension represents the temporal auto-relations between the instructions. The depth dimension represents the instruction entity attributes. Symmetrically, D-CUBE contains three other symmetrical dimensions. The seventh dimension is the inter-relation (LINK) among the two pyramids.

The leaf nodes of both the pyramids denote the concrete entities of P (black sided in the figure). All other nodes are abstract entities. Any abstract entity is the summary (or abstract description) of all its children nodes. Similarly, any relation between them is the summary of all the relations between their children.

As evident by now, the key computational components to this approach are: (i) how to construct I-CUBE and D-CUBE, and most importantly (ii) how to compute abstract LINKs. Section 4 briefly explains the hierarchy generation mechanism. Section 5 presents the novel formalism for LINK computation, which is in fact, is the heart of AMP. However, before that, below I show how the above organization helps us in flexible abstraction.

Information Compaction: Let us define the following two concepts:

Definition (cut): *A **cut** is defined as a set of instruction and data entities, where each of the entities in I-CUBE and D-CUBE is singly-included, i.e. included only once either as itself, or as any one of its spatial-ancestors, or as all of its spatial-children. A **complete cut** is one, which singly-includes all the leaf entities of both the pyramids.*

Definition (thread): *The entities in a cut set, together with inter relations among these entities, form a **thread**. A **complete thread** is generated from a complete cut. A thread T is thus $T = \{I(T), D(T), \Sigma_t(T), \Phi_t(T), \Psi(T)\}$, where $I(T) \subseteq I(P)$, $D(T) \subseteq D(P)$.*

The number of entities at each spatial level is less than that of the level immediately below it, both in I-CUBE and D-CUBE. This reduction generates the opportunity of information compaction at each abstraction level of program representation.

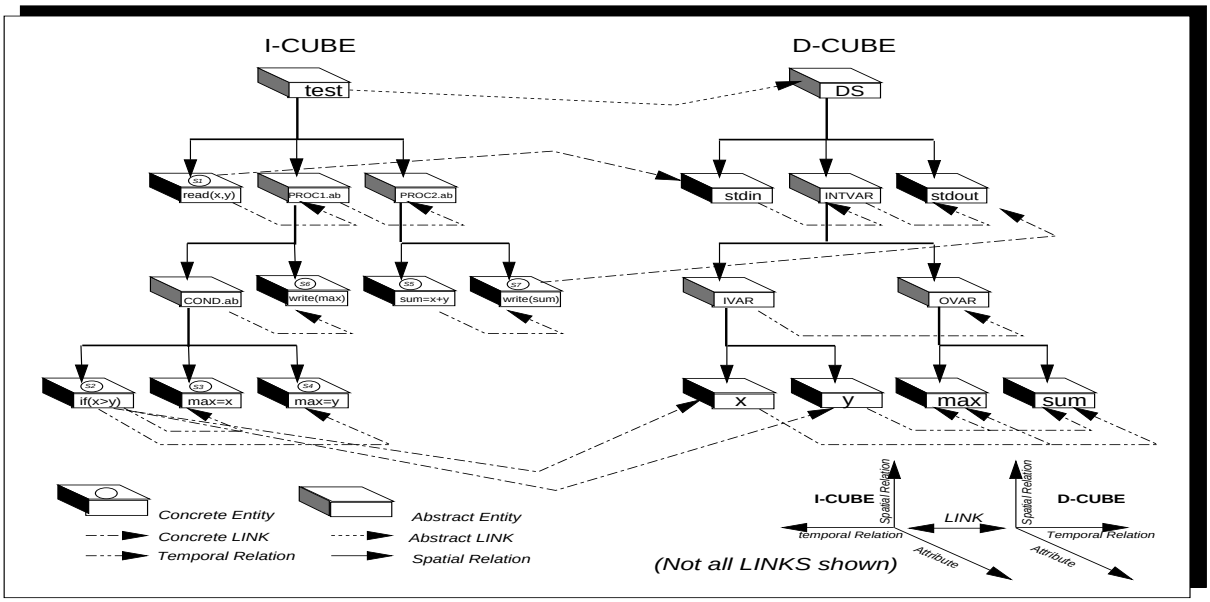

Fig-1 SDH Information Structure

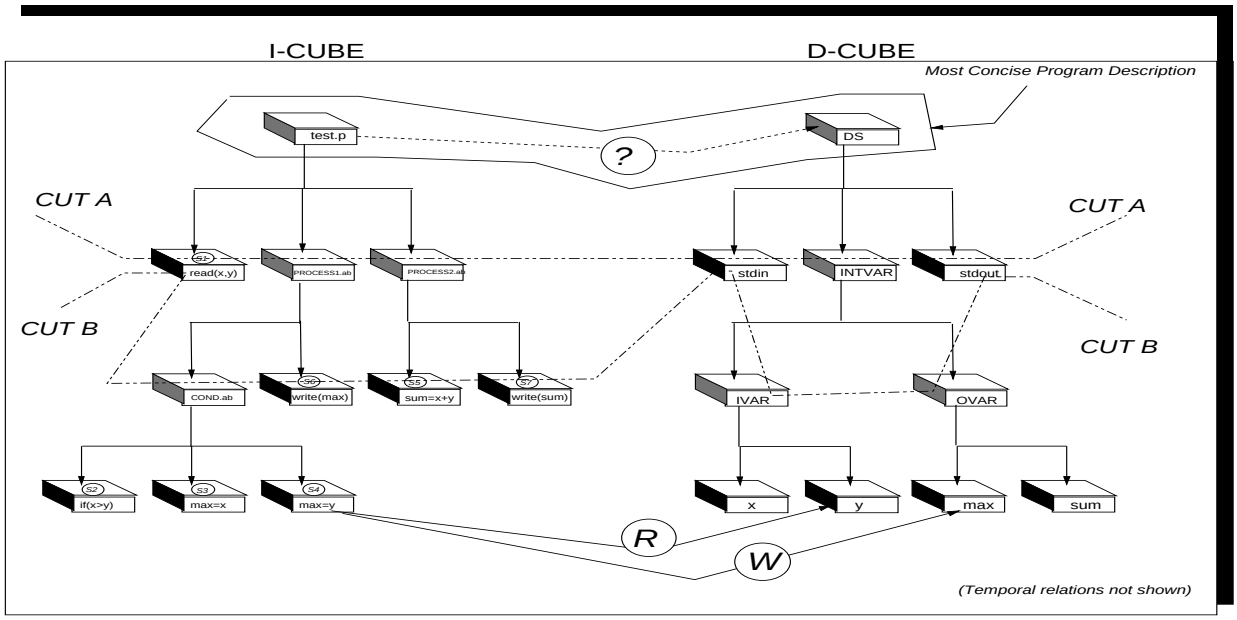

Fig-2 Abstract Representations

Each of the complete threads across the hierarchies forms one complete representation of the program. The average height of the cut provides an intuitive measure of compactness of that particular representation. (For example, in Fig-2 cut A is more abstract than cut B). The pyramidal organization of SDH makes it possible to transcend between a multitude of adiabatic representations by traversing up or down in the spatial hierarchies, allowing flexible control over the pattern of compaction. For example, when, more functional abstraction but detailed data structural information is desirable, the user selects instruction entities from the upper levels of I-CUBE and data entities from the lower levels of D-CUBE. A symmetric inverse drift generates a representation of the same program with more data abstraction and less instruction abstraction. At the highest level of the hierarchies, the description of the two root entities and their relation (a thread through the roots), becomes the maximally abstract representation of the program P. In general, a combinatorial number of adiabatic representations can be formed by varying the pattern of the thread.

Definition (maximally abstract thread): *For a specified set of singly-included entities, the highest (in the hierarchies) complete thread that distinctly contains the entities of the specified set is called a* ***maximally abstract thread****.*

Information Selection: Perspectives are generated by selection on the total information space. Any projection of the 7-dimensional SDH on any of its sub-spaces, represents a generic perspective. Different variants of sub-space perspectives can be obtained by additional selection on the basis of (i) attributes, (ii) attribute value, and (iii) the path of traversal.

For example a classical *call-diagram* is a special case of SDH projection on $I - \Sigma_s$ sub-space, where only instruction entities 'referenced in code' are selected. Similarly, a *conceptual process model* (CPM) [16] can be constructed by selecting the $I(T) - D(T) - \Psi(T)$ projection of a thread **T**. The volume of the CPM can be varied with arbitrary flexibility by varying the thread pattern and its average height. The constituent objects and relations (either concrete or abstract) in all the abstract perspectives are projections from the same underlying SDH. Therefore, all the perspectives remain intrinsically coherent. As explained earlier, such coherency is critical for effective design perception.

Non-Monolithic Program: *Multi-module* (non monolithic) programs can be viewed as a partially abstracted program, where, each of the sub-routines is a priori-abstracted module in I-CUBE. A subroutine is an abstraction performed by the programmer for the ease of forward engineering. Therefore, the base-model of a multi-module program includes not only base instruction entities but also some entities in the upper level of abstraction. PIAS treats an instruction, program segment, or a collection of programs as structurally equivalent and as instances of instruction entity.

Uniform System View: In a computer system, a particular application program is a part of larger system program. On the other hand, the atomic statements and data items of higher level languages themselves are abstract concepts relative to their machine code decomposition. Thus, any application program, subject to our understanding, is only a cross-section (cut) of the overall abstraction hierarchies that are both upward and downward expandable. The abstraction machine of PIAS is intrinsically level independent and uniform in this hierarchical space. Depending on user requirement and availability of resources, it is technically equipped to construct and manage the hierarchies and their inter-relations.

Notably, most abstraction approaches [3,18], for handling large systems only manage the macro aspects, and are function-centric. Thus, these operate only in the upper region

of the I-CUBE, which itself may be only a narrow band of the entire scale spectrum. In contrast to all these previously proposed approaches, ADA formalism seamlessly integrates the macro and micro aspects of large software systems.

4. Hierarchy generation

4.1 I-CUBE generation

The temporal dependency set: The SDH organization of instruction hierarchy is obtained by recurrently clustering sets of instructions with similar temporal dependencies together and arranging the clusters into spatial relation hierarchy. The temporal relation between two instruction objects $\Sigma_t(I_1, I_2)$ has 4 possible enumerations as shown in Table-1.

Seeds	Semantics
>>	*Forward sequential execution dependency*
<<	*Reverse sequential execution dependency*
\| | *Exclusive execution dependency.* |
\|\| | *Concurrent execution dependency.* |

Table-1 Enumerations of Σ_t

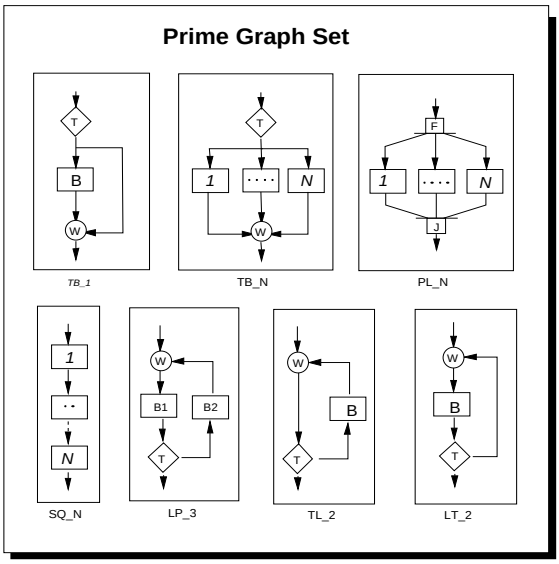

Fig-5 Prime Graphs

The Algorithm: The decomposition algorithm is analogous to prime number factorization. The *temporal instruction dependency graph*, generated from the base model and data dependency analysis, is factorized into a set of prime graphs. A prime graph is one that can not be factorized further. Each detected prime is replaced by an abstract node in the original graph. The procedure is repeated, until it reduces to one single prime graph. If a structured code is decomposed, it results in a set of basic prime graphs shown in Fig-5. (F represents FORK, J represents JOIN and W represents WAIT, and T represents test and BRANCH of Fig-3). Decomposition of non-structured code simply results in additional larger primes. This technique can decompose parallel programs, detect concurrency, breakdown a long sequence of control flow into independent decomposed sub-sequences, (resulting in PL_N prime of Fig-5), and remove redundant control links.

4.2 D-CUBE generation

The SDH organization of data hierarchy is obtained in a similar way by clustering sets of data items with similar temporal dependencies together and arranging the clusters into spatial relation hierarchy. However, there is a small difference. Almost all of temporal dependency Φ_t is generally implicit in a procedural program and is contextual to an instruction segment. Thus, the possible enumeration set of Φ_t is much more complex (than Σ_t). Section-5 presents the formalism for reasoning with Φ_t in more detail.

SDH data space parsing technique use a procedure where first the data model is constructed according to the static structural hierarchy stated in the source code or derivable from the base model. All parents are put below a root node representing the universal data space. For each user declared parents (context), the involved data items are classified into four groups, (i) only read, (ii) only modified, (iii) other references, and (iv) other locals but not referenced (to identify unused variables). However, the static hierarchy explicitly stated through the data declaration segment always overrides the dynamic analysis. Fig-1 is an example of SDH organization generated by parsing the code **test.p** of Fig-6.

5. Abstract dependency analysis

The traditionally bi-variate (with only *read* and *write*) notion of dependency is inadequate to express abstract dependency notions which may arise between abstract entities. In the general case, it is intractable to find a theoretically sound formalism that can express, manage and process arbitrary abstract dependencies (such as, sorting, increment, etc.). However, AMP approach incorporates a formalism that is sufficient to support program dependency analysis on the abstracted models. The following sub-sections present the new dependency set and its grammar.

5.1 Ψ abstraction

5.1.1 The Enumeration set

The Ψ dependency refers to the way an abstract data is affected by an abstract instruction or conversely an abstract instruction is affected by an abstract data. ADA uses three fundamental notions of dependency, namely: *read*, *write* and *read and then write* as the seed concepts for the probable value assignment of Ψ. A set of 15 dependency concepts, based on the three seed concepts, compounded with the notions of *may be*, *or*, and *no effect* constitutes the new dependency set. The set of 15 members is *complete* in a relation field. During abstraction process, the defined transformations combine these input values to synthesize a new output value, which is also always be in the field. Table-2 provides the enumeration values and their semantics.

Enumeration	Semantics
R	*Read in the block.*
W	*Modified in the block.*
R.W	*Read and then modified in the block.*
R⊕W	*Read, or modified in the block.*
R⊕R.W	*Only read, or read and then modified in the block.*
W⊕R.W	*Only modified, or read and then modified.*
R⊕W⊕R.W	*Read, or modified, or read and then modified.*
N	*No dependency.*
R⊕N	*May be read.*
W⊕N	*May be modified.*
R.W⊕N	*May be read and then modified.*
R⊕W⊕N	*May be read, or modified.*
R⊕R.W⊕N	*May be only read, or read and then modified.*
W⊕R.W⊕N	*May be only modified, or read and then modified.*
R⊕W⊕R.W⊕N	*May be read, or modified, or read and then modified.*

Table-2 Enumerations of Ψ

5.1.2 Abstraction in instruction space

Now the transformation rules for the abstraction of Ψ dependency in instruction space are presented. The abstract dependency $\Psi(I,A)$ between instruction **I** and data **A**, when **I** is composed of $\mathbf{I_1}$ and $\mathbf{I_2}$ is given by the 12 production rules given in (1). The function space is clustered on the basis of temporal relations Σ_t (Table-1).

$\Psi(I_1,A).\Sigma_t(I_1,I_2).\Psi(I_2,A)$		$\Psi(I,A)$
X»X	→	X
N»X	→	X
X»N	→	X
W»R	→	W
R»W	→	R.W
R»R.W	→	R.W
W»R.W	→	W
X|Y	→	X⊕Y
X»(Y⊕Z)	→	X.Y⊕X.Z
(X⊕Y)»Z	→	X.Z⊕Y.Z
X⊕X	→	X
X||Y	→	X>>Y⊕Y>>X

….(1)

An abstract node composed of more than two entities incrementally combines the possible paths. Exclusive parallel branches originate from the branch statements. Only one of the conditionally exclusive parallel paths is executed in any single execution of the program. On the other hand, all the mutually parallel paths are executed in any execution of the program. For, example, a loop of TL_2 type (in Fig-5) has two exclusive parallel sequences. Thus, its abstract dependency to a data-item D is,

$\Psi(TL\_2,D) = [\Psi(T,D)] \mid [\Psi(T,D) \gg \Psi(BODY,D) \gg \Psi(T,D)]$

Definition (isolevel): *If an abstract attribute is computable only from the attribute values available, at the maximally abstract thread containing the involved objects, then the computation is* ***isolevel.***

As evident, the production set for $\Psi(I,A)$ is isolevel. Isolevel characteristic is critical for any abstraction grammar. It eliminates the expensive deep tracking to derive upper level abstract information. It also characterizes the reusability of the information generated through previous abstraction.

5.1.3 Abstraction in data space

Similarly, the following 6 production rules generate $\Psi(I,A)$, the abstract (new) dependency relations_during abstraction in data space, when data item A is composed of $\mathbf{A_1}$ and $\mathbf{A_2}$. As evident, this grammar is also isolevel. Here the symbol ^ indicates that both the arguments are children of **A**.

$\Psi(I,A_1)\wedge\Psi(I,A_2)$		$\Psi(I,A)$
X^Y	→	Y^X
N^X	→	X
X^X	→	X
R^W	→	W⊕RW
R^RW	→	W⊕RW
W^RW	→	W⊕RW

..(2)

Fig-6 illustrates an example of Ψ dependency abstraction. The I-CUBE and D-CUBE of this program **test.p** is shown in Fig-1. The left table shows the concrete LINKs obtained from source code, and the right table shows the abstract LINKS computed through the proposed formalism. The rows and columns of this table have been computed respectively using the grammar sets (1) and (2). This link table summarizes the abstract relations between the entities of I-CUBE and D-CUBE.

5.2 Φ_t abstraction

Generally Φ_t is the most implicit information set of a procedural program. Similar to Ψ relations, the traditional scheme is inadequate for abstract entities. Now the ADA formalism for Φ_t is presented.

test.p

S1: read(x,y)
S2: if(x>y)
S3: max:=x
S4: else max:=y
S5: sum:=x+y
S6: write(max)
S7: write(sum)

Base model LINKS

	X	Y		max	sum			stdin	stdot	
s1	W	W		N	N			R	N	
s2	R	R		N	N			N	N	
s3	R	N		W	N			N	N	
s4	N	R		W	N			N	N	
s6	N	N		R	N			N	W	
S5	R	R		N	W			N	N	
S7	N	N		N	R			N	W	

CUBE LINKS

	X	Y	IVAR	max	sum	OVAR	INT-VAR	stdin	stdot	DS
s1	W	W	W	N	N	N	W	R	N	RW+W
s2	R	R	R	N	N	N	R	N	N	R
s3	R	N	R	W	N	W	RW+W	N	N	RW+W
s4	N	R	R	W	N	W	RW+W	N	N	RW+W
COND.ab	R	R	R	W	N	W	RW+W	N	N	RW+W
s6	N	N	N	R	N	R	R	N	W	RW+W
PROCESS1.ab	R	R	R	W	N	W	RW+W	N	W	RW+W
S5	R	R	R	N	W	W	RW+W	N	N	RW+W
S7	N	N	N	N	R	R	R	N	W	RW+W
PROCESS2.ab	R	R	R	N	W	W	RW+W	N	W	RW+W
test.p	W	W	W	W	W	W	W	R	W	RW+W

Fig-6 Links of test.p

Seed Enumerations	Semantics
NF	*No information flows between A and B*
FF	*Information flows from A to B*
RF	*Information flows from B to A*
BF	*Informations flows in both directions between A and B.*

Table-3 Enumerations of $\Phi_t(A,B,I)$

5.2.1 The enumeration set

Unlike, the temporal auto-relation between instructions, the temporal auto-relation between data items is contextual. It is always computed in the context of an instruction entity, either concrete or abstract.

Definition (information flow): *If the value of data entity A before the execution of instruction I (A_{beg}), affects the value of data entity B after the execution (B_{end}), then we say that there is a* ***flow of information*** *from A to B in the context of instruction I.*

The context can be varied from as narrow as a single base model instruction to the entire program by selecting the root entity of I-CUBE. Table-3 lists the possible seed events of $\Psi(A,B,I)$ denoting the flow from the data item A to B in the context of instruction I. An abstract relation may contain any combination of the above seed events (because of the various execution paths) compounded with operator $\oplus$ resulting in 31 possible enumerations. This enumeration set is also *complete*.

5.2.2 Computation grammar

Given two data entities A and B, and a context instruction I, the dependency $\Phi_t(A,B,I)$ can be computed from the component sequences of $\Psi(I,A)$ and $\Psi(I,B)$. Production set (3) is used to compute it. Compound $\Psi(I,A)$ results in corresponding compound $\Psi(A,B,I)$ values by product expansion.

An enumeration computed by this grammar provides the possible directions of information flow for which the necessary condition exists in the overall dependency. Thus, it only indicates possibility of the associated event, rather than confirming the event. The grammar is isolevel. If required, sufficient conditions can be derived through deep tracking only in the hierarchy below the context instruction.

$\Psi(I,A).\Psi(I,B)$		$\Phi_t(A,B,I)$
$[X]_A.[N]_B$	$\rightarrow$	NF
$[N]_A.[X]_B$	$\rightarrow$	NF
$[R]_A.[R]_B$	$\rightarrow$	NF
$[R]_A.[W]_B$	$\rightarrow$	FF
$[R]_A.[RW]_B$	$\rightarrow$	FF
$[W]_A.[R]_B$	$\rightarrow$	RF
$[W]_A.[W]_B$	$\rightarrow$	NF
$[W]_A.[RW]_B$	$\rightarrow$	RF
$[RW]_A.[R]_B$	$\rightarrow$	RF
$[RW]_A.[W]_B$	$\rightarrow$	FF
$[RW]_A.[RW]_B$	$\rightarrow$	FF

….(3)

Below the abstract information flow analysis is illustrated through an example. Fig-7(a) and Fig-7(b) shows the source code, I-CUBE, D-CUBE and LINK tables of a program **noname.p**. Fig-8 shows the data-flow profile between the connected (those which have LINKs from context) data objects in the context of entire program **noname.c**(Fig-8(a)), abstract module **TEST.ab** (Fig-8(b)), and abstract module **BODY.ab** (Fig-8(c)) as computed using the above grammar. The forward flow *F* of Fig-8(a) indicates the net information-flow effect from **FILE-Y** to **FILE-X.** On the other hand, abstract flow *B+N* of Fig-8(b) indicates that there exists two possible outcomes of **TEST.ab,** one of which results in bi-directional transfer and other results in no transfer of information between **A** and **B**. As evident, these

profiles summarize (and also conform to) the fact that **noname.p** is a program which reads data from **FILE-Y**, sorts them and writes the sorted data to **FILE-X**. If all the data are already in sorted order, then there will be no transfer of information between internal variables A and B.

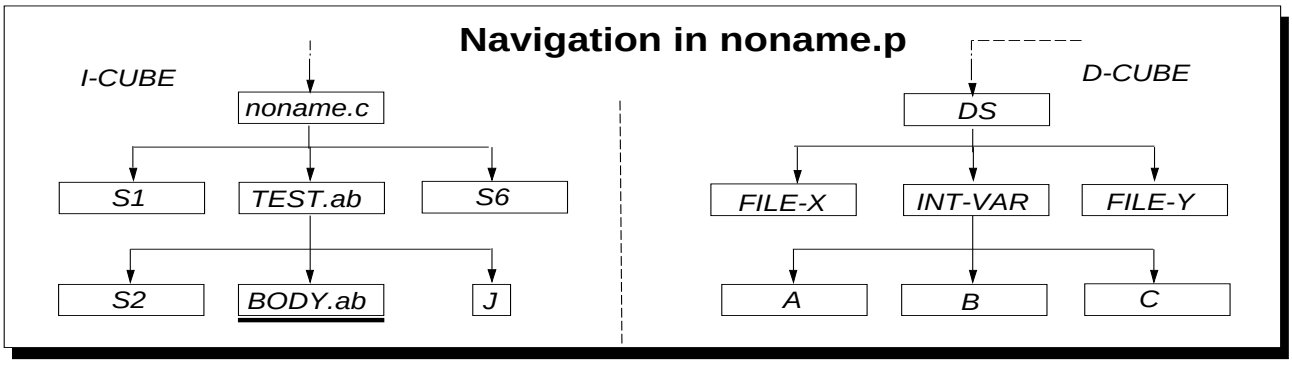

Fig-7(a) Navigation in noname.p

noname.p

S1: Read(File-X, A,B)
S2: If(A > B)
S3: C:=A
S4: A:=B
S5: B:=C
S6: Write(FILE-Y, A,B)

Base Model LINKS

	A	B	C		FILE-X	FILE-Y	
S1	W	W				R	
S2	R	R					
S3	R	N	W				
S4	W	R					
S5		W	R				
S6	R	R			W		

CUBE LINKS

	A	B	C	INT-VAR	FILE-X	FILE-Y	DS
S1	W	W	N	W	N	R	RW+W
S2	R	R	N	R	N	N	R
S3	R	N	W	W+RW	N	N	W+RW
S4	W	R	N	W+RW	N	N	W+RW
S5	N	W	R	W+RW	N	N	W+RW
BODY.ab	RW	RW	W	W+RW	N	N	W+RW
TEST.ab	R+RW	R+RW	W+N	RW+R	N	N	RW+R
S6	R	R	N	R	W	N	RW+W
noname.p	W	W	W+N	W	W	R	RW+W

Fig-7(b) Links of noname.p

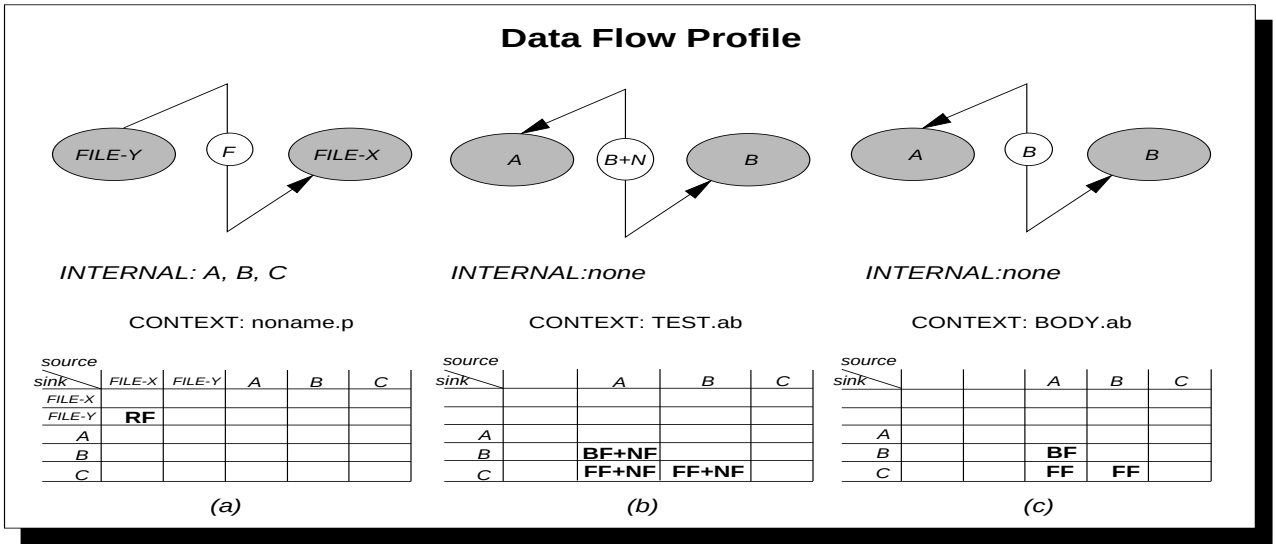

Fig-8 Profile

5.3 Program dependency analysis

This section shows the abstract analysis of specifically two fundamental program dependency forms; *data flow dependency* and *control flow dependency* using ADA formalism. Podgruski and Clarke [14] have formally shown that these two are fundamental pair of program dependencies. Other abstraction systems require complete traversal of concrete program models, when such dependencies are to be analyzed.

5.3.1 Data flow dependency analysis

Informally, if the outcome of statement **A** is dependent on the outcome of statement **B** (by a set of variable assignments), than statement **A** is data flow dependent on statement **B**. Below a formal definition is given in the context of PIAS information organization.

Definition (Data Flow Dependency): *Let* $G = \{I, D, \Sigma_t, \Phi_t, \Psi\}$, *be a thread and let* $u, v \in I(G)$. *Instruction u is directly data flow dependent on v iff there is a walk vWu in G such that* $(DEF(v) \cup USE(u) - DEF(W)) \neq \emptyset$. *u is data flow dependent on v iff there is a sequence* $v_1, v_2, \ldots v_n$ *of vertices,* $n \geq 2$, *such that* $u = v_1$ *and* $v = v_n$, *and* v_i *is directly data flow dependent on* v_{i+1} *for i=1,2,...n-1. A data entity* $d \in DEF(u)$ *if* $\Psi(u, d) = W \mid RW$, *and* $d \in USE(u)$ *if* $\Psi(u, d) = R \mid RW$.

This generalized definition reduces to the formal definition of data flow dependency provided by [14] when only concrete entities are considered. As evident, analysis requires only the Ψ information present in the thread, without any deep tracking. Thus, it is isolevel. The analysis at higher abstraction level requires traversal of shorter paths. As a result, not only the computational efficiency of the analysis increases at higher abstraction level, but also the perceptual efficiency of the human subject in visualizing the dependency increases.

5.3.2 Control flow dependency analysis

Informally, if the outcome of statement $\mathbf{S_1}$ decides the executability (or the number of times it may be executed) of statement $\mathbf{S_2}$, then statement $\mathbf{S_2}$ is control dependant on statement $\mathbf{S_1}$.

Definition (Control Flow Dependency): *Let* $G = \{I, D, \Sigma_t, \Phi_t, \Psi\}$, *be a thread and let* $u, v \in I(G)$. *Instruction v is control flow dependent on u iff, u and v are not direct child or parent to each other, and there exists a sequence* $v_1, v_2, \ldots v_n$ *of vertices,* $n \geq 2$, *such that* $u = v_1$, $v = v_n$ *and either of three relations exists:* $\Sigma_t(v_i, v_{i+1}) = \gg$, $\Sigma_s(v_i, v_{i+1}) = \uparrow$, *or* $\Sigma_s(v_{i+1}, v_i) = \uparrow$ *for i=1,2,...n-1.*

The symbol $\uparrow$ indicates the child-parent relations in spatial dimension. As stated earlier in section 5, during the organization of I-CUBE from the base, the abstraction (or grouping of statements) is performed only on the basis of temporal auto-dependency. No control sequence is altered during the SDH organization, rather all redundant dependencies are removed, implicit concurrencies are detected and the instructions are categorized. As a result, control dependency analysis is more straightforward in I-CUBE. If there is a control dependency between two entities, then there must also be temporal auto-dependency between any of their parents in the spatial hierarchy. Thus, control dependency analysis requires computation only in and above the spatial levels of the given entities. The number of entities at higher level is smaller than the number of entities at the lower level. Thus, the control dependency analysis is also equivalent to isolevel computation.

6. Applications

Adiabatic Navigation: The SDH organization allows the software engineer to visually access the very micro detail of any piece of information and to swiftly navigate to other, however, without necessarily being forced to lose the broader context of the macro structure of the software. Due, to SDH hierarchical organization, it takes only logarithmic time (in terms of the number of system components) to access any piece of information. To maintain the macro-view, user can flexibly select immaterial segments and compress details. The expressiveness of the ADA formalism apparently conserves the net dependency information (or adiabaticity) despite the compaction.

Flexible Visualization: A large set of program visuals can be constructed on the basis of the SDH organization and ADA formalism with improved clarity and efficiency to facilitate design extraction. The projections of the seven component SDH space, on any of its sub-spaces define generic perspectives. Various specialized program perspectives can be formed by further filtration of the generic perspectives. The dimension of any such visual that is deducible from the seven components of the program information space can be further controlled through compaction. Note that such reduction of dimension is different from zooming. In cases of overcrowded visual, control of visual dimension through zooming provides little help in contrast to the conceptual abstraction. Usual zooming means gaining micro-view at the cost of macro-view.

Efficient Analytic Tools: Dependency analysis forms the heart of numerous software tools used in applications ranging from testing, impact analysis, debugging, maintenance, code optimization, to parallelization and computer security [3,10,14]. With the capability of abstract dependency analysis, PIAS can perform various software engineering tasks readily and almost effortlessly with increased computational as well as perceptual efficiency. For example, in *dead code elimination* [3], any program segment that remains disconnected from the principal hierarchy of I-CUBE represents the dead code. Similar advantages can be demonstrated in a number of other applications such as *subsystem porting, complexity estimation, program layering, smart-recompilation, operators-fault detection, complexity matrix estimation, impact analysis, etc.* [3,14,18].

7. Conclusion

This paper presents the adiabatic multi-perspective (AMP) approach to program abstraction, which can help the human expert in exploring the large and complex information space of an unfamiliar software. To the knowledge of the author, this paper introduces one of the first formalism to perform efficient isolevel dependency analysis at abstract level.

Because of this new capability, a wide variety of coherent perspectives with **general** adiabatic compressibility can be generated by sub-space projection, which includes all those program views (those can be deduced solely from code information) cited in previous abstraction systems. PIAS provides a seamless **uniform analytic platform** for comprehension of software design both at macro level as well as micro level. The upper (how big a system) and lower limits (how detailed a system) in the abstraction scale is only constrained by storage capacity and speed. As shown, the isolevel property of the grammars ensures computational efficiency of the entire abstraction process. As a future direction, currently, I am investigating a new abstract dependency language with more expressive power. Finally, the author would like to thank members of SERL and specially Dr. Miyamoto for generous intelletual guidance in pursuing critical directions in this research.

8. Reference

[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] J. Browne, D. B. Johnson, "FAST: A Second Generation Program Analysis System", *Proceedings of 2nd International Conference on Software Engineering*,1977, pp. 142-148.

[3] Y. F. Chen, M. Y. Nishimoto, C. V. Ramamoorthy, "The C Information Abstraction System", *IEEE Transaction on Software Engineering*, v.16, no.3, March 1990, pp. 325-334.

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

[5] W. E. Howdens & S. Pak, "Abstraction and Problem Dependency in Reverse Engineering", *Fujitsu Workshop on Program Maintenance*, Tokyo, Japan, Sept, 92.

[6] S. Horowitz, T. Reps & D. Binkley, "Inter procedural Slicing Using Dependence Graphs", *Technical Report 756*, Univ. Of Wisconsin at Madison, March 1988.

[7] Javed I. Khan & I. Miyamoto, "Formalism for Hierarchical Organization and Flexible Abstraction of Program Knowledge", *Proceedings of the 5th International Conf. on Software Engineering and Knowledge Engineering*, SEKE93, San-Fransisco, June, 1993, pp. 301-303.

[8] Javed I. Khan & I. Miyamoto, "Integrating Abstraction Flexibility with Diverse program Perspectives", *Proceedings of the 17th Annual International Computer Software and Applications Conference*, COMPSAC'93, Phoenix, November 1993, pp. 186-192

[9] M. A. Linton, "Implementing Relational Views of Programs", *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium*, May 1984.

[10] I. Miyamoto, et. al, "A Multiple-View, Modelling-Based Prototyping Environment", *Technical Report*, ICS Dept, University of Hawaii, 1987.

[11] I. Miyamoto, "A Prototyping Tool for Graphical Software Engineering Tools", *ACM Software Engineering Notes*, 12(4), 1987.

[12] J. Q. Ning, "A Knowledge Based Approach to Automatic Program Analysis". *Technical Report*, Ph.D. thesis, University of Illinois, U-C, 1989.

[13] N. Pennington, "Stimulus structures and Mental Representation in Expert Comprehension of Computer Programs", *J. of Cognitive Psychology*, v. 19, 1987, pp. 295-341.

[14] A. Podgruski, & L. A. Clarke, "A Formal model of Program Dependances and Its Implication for Software Testing, Debugging, and Maintenance", *IEEE Transactions on Software Engineering*, Vo.6, no.9, Sept. 1990, pp. 965-979.

[15] "A Case Study In Reverse Engineering", *Technical Report 91-44*, Software Engineering Research Lab, Univ. Of Hawaii, Sept, 92.

[16] K. Takeda, D. Chin, & I. Miyamoto, "MERA: Meta Language for Software Engineering", *Proc. Of the 4th Intl. Conf. on Software Engineering & Knowledge Engineering*, June, 1992, Capri, Italy, pp. 495-502.

[17] W. Teitelman, and L. Masinter, "The Interlisp Programming Environment", *Computer*, v.14, no.4, April, 1981, pp. 25-34.

[18] R. S. Tilley, H. A. Muller, M. J. Whitney, K. Wong, "Domain-Retargetable Reverse Engineering", *Proceedings of Conference on Software Maintenance*, CSM'93, Montreal, Sept., 1993, pp. 142-151.

[19] M. Weiser, "Program Slicing", *IEEE Tran. On Software Engineering*, v. SE-10, no.4, July 1984, pp. 352-357.

[20] L. M. Wills, "Automatic Program Recognition by Graph Parsing", *Technical Report*, no 1358, MIT Artificial Intelligence Lab, 1992.