An Intermediate Representation for Integrating Reverse Engineering Analyses

Rainer Koschke

University of Stuttgart
Breitwiesenstr. 20-22
D-70565 Stuttgart, Germany
koschke@informatik.uni-stuttgart.de

Jean-François Girard, Martin Würthner

Fraunhofer Inst. for Experimental Software Eng.
Sauerwiesen 6
D-67661 Kaiserslautern, Germany
{girard,wuerthne}@iese.fhg.de

Abstract

Intermediate representations (IR) are a key issue both for compilers as well as for reverse engineering tools to enable efficient analyses. Research in the field of compilers has proposed many sophisticated IRs that can be used in the domain of reverse engineering, especially in the case of deep analyses, but reverse engineering has also its own requirements for intermediate representations not covered by traditional compiler technology This paper discusses requirements of IRs for reverse engineering. It shows then how most of these requirements can be met by extending and integrating existing IRs. These extensions include a generalized AST and a mechanism supporting multiple views on programs. Moreover, the paper shows how these views can efficiently be implemented.

Keywords: reverse engineering, program representation, views

1. Introduction

As reported at WCRE 96 [Ruga96], one step toward a research infrastructure accelerating the progress of reverse engineering is the creation of an interoperable intermediate representation. This paper contributes to this goal by presenting a list of requirements for such a representation and proposing a candidate technology.

In the field of compiler technology, the importance of a suitable intermediate representation **(IR)** has long been recognized. The design of an intermediate representation is a key factor for the efficiency of analyses for code optimization and has been extensively studied. This experience should be reused by reverse engineering analyses which originally stem from optimizing compiler technology.

However, this compiler technology does not address all reverse engineering concerns, since reverse engineering has also additional demands on intermediate representations, for example, the need to interact with users during analysis and capture their feedbacks. Furthermore, as Kazman, et al. point out in a paper at this conference [Kazm98], an intermediate representation for reverse engineering must support different levels of abstraction - from the code-structure level up to the architectural level - to be suitable for all phases of reverse engineering.

Related Research

Many IRs have been proposed (DIANA [Goos81], CCG[Kinl94], IRIS [iri], F(p) [Cimi91]) but most of them are language-specific or do not take advantage of more recent advances in optimizing compiler technology which could provide the performance needed to analyze large industrial systems. Examples of such recent advances include sparse representations like SSA [Wegm91], GSA [Ball90], and Dependence Flow Graphs [Ping91], which support more efficient analyses.

In reverse engineering, Kinloch and Munro [Kinl94] use the combined C graph to identify reusable components. In [Cimi95], Cimitile et al. combine this graph with AST in order to support specification-driven slicing. In both cases, the representations are language-specific. Canfora and Cimitile propose F(p) [Cimi91] which is largely language-independent but is not designed to optimize operations of any particular analyses.

A related topic is the data exchange format among tools potentially using different IRs. Klint and Verhoef propose the Annotated Term Format as an externalized representation as part of the Toolbus architecture [Klin98] that supports tool integration.

Paper outline

This paper draws a list of requirements of IRs for reverse engineering (Section 2), discusses the drawbacks and advantages of existing IRs (Section 3) and proposes a new IR which meets most of these requirements (Section 4). In addition, an implementation for the view mechanism is suggested (Section 5) and the use of the IR is illustrated with usage scenarios taken from Kazman et al. [Kazm98] and from our own work on abstract data type recovery (Section 6).

2. IR Requirements for Reverse Engineering

Tasks in reverse engineering can involve analyses ranging from very fine-grained, e.g., checking which function pointers can be used to call a routine, to rather coarse-grained, e.g., providing an overview of the main components of the system and what means of communication are used among them. An intermediate representation should support all these tasks. Generally, an IR for reverse engineering supporting deep analyses will have the same requirements as traditional IRs for compilers. But an IR for reverse engineering also has further requirements, in particular to support shallow analyses and to allow for higher abstractions and user annotations.

This section discusses requirements for an IR based on our experience in the field of architectural recovery [Gira97]. However, we believe that these requirements for an IR are typical for many reverse engineering tasks. The requirements can be divided in those that are shared by IRs for reverse engineering and IRs for compilers and additional ones that go beyond those for compilers.

2.1. Common requirements

(R1) The IR should be programming language independent; that is it should abstract away from language specific syntax, so analysis can be applied onto different languages of the same family (e.g., procedural) without major modifications.

(R2) The semantics of the IR must be well-defined and it must exactly describe the constructs of the modeled programming languages; this is necessary for an exact analysis.

(R3) Traversals of the IR should be efficient; this is necessary because analyses usually imply traversing the IR at least once, in an iterative algorithm even many times.

(R4) The IR should be constructed efficiently. This property is a necessary condition for an overall efficient analysis. It is required to handle large systems in a reasonable time.

(R5) Likewise, the IR should be linear in size to the length of the source code. This property is particularly important for global analyses of large programs.

(R6) The IR should allow efficient control and data flow analysis.

2.2. Reverse Engineering Specific Requirements

While many requirements of an IR are shared between reverse engineering tools and compilers, others are specific to the reverse engineering context. The most important are

the following:

(R7) The IR should preserve a mapping to the original source code; this enables an analysis to feed back information to the user in terms of the original program. This requirement is in conflict with (R1). The strategy used to resolve this conflict is discussed below.

(R8) The IR should be able to represent a system made of several programs. In this context, it should distinguish two instances of the same variable (same name, declared in the same file) when this file was linked to two different programs. This is necessary, for example, while investigating the interaction between these programs.

(R9) While compilers need to know all tiny details of a program, for many maintenance tasks a rough overview is sufficient; thus the IR should support different levels of granularity from fine-grained to coarse-grained.

(RIO) When used in an interactive reverse engineering environment, the IR should capture information provided by the user in addition to facts that are directly derivable from source code as needed by a compiler. This information takes the form of annotations (free comments in natural language), attributes (structured data), or assertions (predicates that the user guarantees to hold during runtime).

(RI 1) It should be possible to save the IR. As opposed to compilers, reverse engineering often involves user interventions and computational expensive analyses: so these results should be preserved.

(R12) IRs for compilers consist only of programming language constructs; in a reverse engineering environment IRs must also capture higher level abstractions. For example, cliche recognition may detect an abstract concept *stack* in the source code consisting of a set of routines and a type; the concept and its connection to the programming language constructs that make up this concept should be represented by the IR. To be useful in practice, it should be possible to add new abstract concepts to the IR without invalidating all previous system analyses.

(R13) Not only does the IR have to have the ability to specify higher concepts, it also must provide means to express any relationships between these concepts. A part-ofrelationship to describe hierarchical relationships is one example, a *communication* relationship between two subsystems in an architectural description is another. The IR should also allow for attributing these relationships.

(R14) In a multi-user reverse engineering environment, we have the need to allow each user to have his or her own view of the system. Similarly, the IR should capture the

results of analyses that compute alternatives which can also be regarded as different views.

2.3. Requirements Discussion

Requirements about the efficiency of construction and traversal of an IR $(R3,R4,R5)$ are important for compilers. They are even more important for reverse engineering, because, as opposed to compilers that usually analyze individual modules and generate code for them separately, reverse engineering generally involves the analysis of the system as a whole. Therefore, scalability becomes a key issue. This is even more true in an interactive environment.

If source code of different programming languages has to be analyzed, the intermediate representation must be general enough to cover these different languages (RI). Otherwise, i.e., in the case of different representations for different languages, specific analyses would have to be written for each language. This would imply duplication of effort and major maintenance problems.

CBMS: An Example of Reverse Engineering Needs

Code base management systems **(CBMS) are** reverse engineering environments that store, retrieve, analyze, and visualize information on software artifacts. CBMS offer a query language to the contents of the database and are open in the sense that tools can be plugged to gather information or to add information to the database. The contents of the database is the source code, its change history, results of previous analyses, further user documentation, and other artifacts. At the core of the database, the source code is represented in a way that allows efficient analyses.

IRs usually need much more disk space than the source code. With an efficient way to construct the IR, regenerating the IR from the source code each time becomes an appealing alternative.This way, the regenerated IR is always consistent with the source code and suitable for an interactive CBMS which allows code modifications. This is an important factor in a CBMS that administrates a change history . instead of saving the IR for each revision of the source code, the source code of the subject revision can be checked out and the IR can quickly be generated for it.

One could argue that revisions should be a concept supported by the IR. This way, all revisions of the system would be available at the same time, which would ease analyses that compare different revisions, a frequent maintenance activity. On the other hand, as a consequence, the IR would contain information that is only relevant to former revisions. This and the need for each piece of information in the IR (at least of the not directly derivable part) to maintain its own revision information would cause explosive growth of the IR. That is why we consider administration of revisions a task of the CBMS. For each revision

of the source code, the CBMS should save the corresponding non-derivable part of the IR (or only the delta of revisions) and retrieve it when it is necessary to compare different revisions. For this comparison, it is sufficient for the IR to support different views (R14).

3. Proposed Intermediate Representations

There are many proposals of intermediate representations in the compiler literature. Due to lack of space, this paper discusses only those which form the basis for our new intermediate representation. A reader interested in other existing IRs could consult [Aho86, Ball90]. This section presents abstract syntax trees, gated single assignment form, and entity-relationship graphs. It then reports which of these representations fulfill the requirements presented in the previous sections.

3.1. Abstract Syntax Trees

For the semantic analysis, compilers often construct an abstract syntax tree **(AST)** whose nodes are programming language constructs and whose edges express the hierarchical relation between these constructs. The structure of an AST is basically a simplification of the underlying grammar of the programming language, e.g., by generalization or by suppressing chain rules. Attributed ASTs have simple attributes, such as the level of scope, and further edges, such as links from each use of an identifier to its declaration. Edges that make up the tree structure of the AST are called **syntactic edges** and the additional edges are called **semantic edges.** The semantic edges extend the AST from a tree to a general directed (possibly cyclic) graph.

Usually, each programming language has its own specific AST structure because an AST is more or less a one-to-one representation of the source code (R7). However, this structure can be generalized so that it can be used to represent programs of different languages (RI). How this can be done without inflating the IR with separate constructs for each individual programming language and yet still fulfilling $(R1)$ is discussed in Section 4.

Following the syntactic edges of an AST allows for an efficient top-down traversal (R3). Adding semantic edges to an **AST** allows for alternative traversal strategies. For example, in subtrees for expressions, a parent edge to climb the tree bottom-up is useful. The AST can efficiently be built (R4) and is linear in the size of the program (R5). ASTs can easily be stored $(R11)$, but since they can quickly be generated directly from the source code, this is often not done.

The ASTs of separate compilation units can be linked together to form one system AST if a global analysis is required. This can also be done for the source code of sys-

terns consisting of several programs. In this case, we only have to keep apart different main programs and different incarnations of globally declared variables (R8).

Beyond syntactic and semantic edges and simple attributes in an attributed AST that are directly derived from the source code, each node could also be annotated by the user (R10).

Structured control flow is represented in ASTs by syntactic edges and the type of nodes; e.g., the syntactic edges for the *then* and else parts of an *if* node point to the next statement to be executed; once either the *then* or the else branch has been executed, execution continues at the next statement after the *if* node. Additional semantic edges for *gotos* that point to the corresponding label, may denote unstructured control flow. Restricted data flow information is only implicitly available by assignment and expression nodes. However, for a variable at a certain point in the program, there is no information available where its value was last set.

Since ASTs are a one-to-one representation of the source code, they offer only one, very detailed level of granularity. They neither offer means to express more abstract concepts than programming language constructs nor do they provide additional relationships between these abstract concepts. They do not offer different views either.

3.2. Gated Single Assignment Form

The Gated Single Assignment (GSA) form was introduced by Ballance, Maccabe, and Ottenstein [Ball90]. It is based on the Static Single Assignment (SSA) form.

SSA was introduced by Cytron et al. [Cytr9 1] to speed up data flow analyses by providing explicit links between variable references and the definitions that can reach them. This is enabled by the addition of +-functions that merge multiple definitions converging at places where different control flow paths merge, thereby creating a unique definition for each use of a variable.

GSA extends SSA by using different types of gating functions instead of φ-functions according to the nature of the control flow join: γ-, μ- and η-functions are used at the join point after an *if* branch statement, at the head of a loop, and at the exit of a loop, respectively. Each gating function contains a predicate specifying the condition under which the node is reached by a specific branch of control. GSA form allows for efficient predicated data flow analyses.

Strictly speaking, SSA (GSA, respectively) is not an intermediate representations on its own. It is rather a means of extending an existing intermediate representation to represent definitions and uses. Often, control flow graphs are extended with SSA information [Wegm91]. A more compact representation in conjunction with SSA is the Value Graph [Alpe98]. However, the Value Graph does not com-

pletely represent the original source program. In Section 4, we will show how ASTs can be enhanced by GSA.

The GSA is language independent and its semantics is well-defined. An intermediate representation with GSA offers an efficient traversal by def-use links. While a pure AST allows only for top-down traversal, an AST enhanced by GSA offers the possibility of following links of a definition to its uses and vice versa. GSA allows sparse traversal and thereby enables efficient data flow analysis.

SSA can be built in linear time with respect to the number of edges [Sree94]. SSA can then be converted into GSA. Peng Tu developed an efficient way to construct GSA directly in almost linear time for structured programs [Tu95]. For certain kinds of n nested loops, the space needed for SSA may be in the range of $O(n^2)$. However, measurements for typical programs indicate that SSA is linear to the program size [Cytr9 1].

3.3. Entity-Relationship Graph

ASTs and GSA form provide very fine-grained information. A more global picture of the system can be represented by an Entity-Relationship Graph (ERG). An ERG is basically a general entity relationship model to represent knowledge on a given program. The entities of an ERG are programming language concepts of interest, such as functions, types, and variables, but also more abstract concepts, such as abstract data types, components, or subsystems. The entities are represented as nodes in an ERG and relationships among these concepts are represented as edges. Examples of relationships range from those that can directly be derived from source code, such as function calls, to more abstract relationships, such as the communication between a client and a server.

A Resource Flow Graph (RFG) is an example of a concrete instantiation of the ERG. It consists of functions, types, and variables together with call, data, and type relationships among them that can directly be derived from source code. RFGs are used in approaches to detect subsystems [Müll90] and abstract data types [Gira97]. In both applications, higher abstractions are added to those directly derived from source code: the concepts of subsystems and abstract data types.

In the context of our work, we also added nodes and edges for components (tasks, filters, client/server, etc.) and connectors (pipes, remote procedure calls, shared memory, etc.) to describe the architecture of a system.

3.4. Summary

Table 1 reports to what extent each of the representations presented so far fulfills the requirements presented in Section 2. In this table, a fulfilled requirement is marked

with a "+" and an unfulfilled requirement is marked with a "-." A "~" means that the requirement is not completely fulfilled. Some of the entries for GSA are blank, because the question whether these requirements are fulfilled can only be answered in the light of a real intermediate representation that is enhanced by GSA. The additional IRs in Table 1 will be explained in Section 4.

Table 1. Properties of intermediate representations

Req.	efficient												abstract	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
AST	-	+	+	+	+	-	+	+	-	+	+	-	-	-
GSA	+	+	+	~	~	+								
ERG	i	i	+	+	+	-	-	+	+	+	+	+	+	-
GAST/GSA	+	+	+	~	~	+	+	+	-	+	+	-	.	.
ERGV	+	+	+	+	+	-	-	+	+	+	+	+	+	+
IIR	+	+	+	-	-	+	+	+	+	+	+	+	+	+

None of the intermediate representations discussed so far fulfills all needed requirements. This is in part due to R9 which requires to support fine-grained as well as coarse-grained analyses: an AST extended with GSA form is intended for fine-grained analyses only and ERGs support only coarse-grained analyses. An obvious solution is to integrate all these different intermediate representations to get the advantages of each of them (N.B. this integration also inherits the disadvantage of GSA form in the case of R4 and R5). Yet, this integration alone does not meet R14 (support for different views).

4. Integrated intermediate representation IIR

The proposed intermediate representation **(IIR)** is an integration of existing intermediate representations consisting of two levels: The lower level is a Generalized Abstract Syntax Tree **(GAST)** extended with GSA to allow fcr efficient data flow analyses and other fine-grained analyses. The higher level is an ERG that allows coarse-grained analyses and supports adding higher concepts. The AST is general in the sense that it can uniformly represent programs written in different languages. To fulfill the yet unfulfilled requirement (R14), we extended the ERG to support different views.
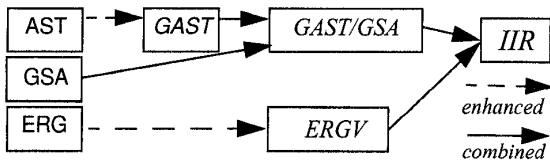

Figure 1. Concept flow towards IIR

Figure 1 provides art overall view on how the intermediate representations discussed in Section 3 were extended and combined to IIR.

The lower level of the intermediate representation is constructed in two steps. First the GAST is generated directly from source code by a language-specific frontend, and then the GSA form is computed (see Figure 2). Based on this information, the basic ERG is derived. It consists solely of entities and relationships explicitly present in the source code.
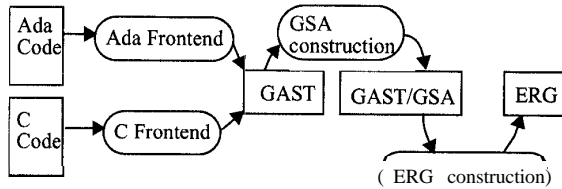

Figure 2. Construction of intermediate representation.

Higher abstractions, attributes, annotations, and assertions are then added to the higher level (ERG) by the users or by further analyses operating on the IIR. Note that information is only added to the ERG part. Thus the GAST/GSA part can always be m-generated; is it supposed to represent all the source code and nothing but the source code.

The two levels are really kept distinct, i.e., a node in the ERG that has a direct correspondence in the GAST appears twice: in the GAST and in the ERG. However, the two nodes are linked in both directions. This allows for switching from an ERG node to the corresponding GAST node to get detailed data flow information and back from a GAST node to an ERG node to get user annotations, for example, and still saves analyses that are only based on one part of the intermediate representation from loading the other part. This is an important factor for coarse-grained global analyses solely based on the ERG since ASTs need a lot of space. By adding a mediator layer in-between the two levels instead of having immediate accesses between the two levels, it is possible to add a loading on-demand capability: only when a GAST node is needed, the GAST of its compilation unit is loaded into memory.

The rest of this section discusses the general design ideas of the GAST and ERG implementation.

4.1. GAST with GSA

The GAST is designed for representing all constructs of the programming language C and for a subset of Ada, namely its sequential and procedural part. To keep the number of distinct constructs in the intermediate representation small in order to save analyses from distinguishing many different cases, one design principle was to provide simple constructs that can be combined to represent any of

the languages constructs [Würt96]. This can be illustrated with the example of *while* loops. Instead of having an explicit *while* loop with an associated condition, a *while* loop can be represented by using a general *loop,* an *if-then-else,* and an *exit* statement to quit the loop as illustrated by the two equivalent programs in Figure 3 (a) and (b).

```
                              loop
while P  loop                   if P then
   A;                             A;
end loop;                       else
                                  exit;
                                end if;
                              end  loop;
```

(a)   **(b)**

Figure 3. Equivalent loops.

(a) → syntactical edges

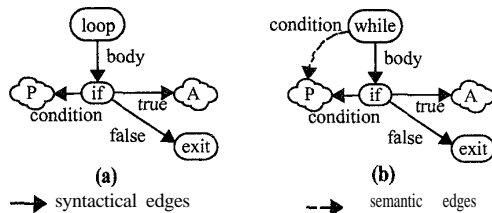**(b)** --→ semantic edges

Figure 4. GASTs for loops.

The graph representation of the loop statement in Figure 3b is shown in Figure 4a. This mapping has the disadvantage that the organization of the original source code gets lost. To avoid this, a *while* node class is derived from the general loop node class with an additional semantic attribute that points to the condition of the *while* loop (Figure 4b). Now, since *while* is a *loop* and the additional *condition* attribute is only semantic, the analysis sees the loop in Figure 4a but the source code is still appropriately represented by the graph in Figure 4b. The same design principle is applied to capture the semantics of similar yet different constructs in C and Ada, such as the *for*-loop. In Ada, start and end value of a *for*-loop are mandatory. Both expressions are evaluated only once. The step value is 1 implicitly. In C, all these parts are optional and can be any expression (which includes assignments in C). The initialization is executed once, the expressions of the condition and the step are executed for each iteration. Despite these differences, both loops can be represented in a uniform manner by means of the intermediate representation. In the case of Ada, the implicit code of the for loop is added to the intermediate representation. Figure 5 illustrates this (the intermediate form is written out as an equivalent Ada program for readability).

In the case of C, the same structure can be chosen (but E2 has to be executed in each iteration and is therefore moved into *the* body). A *continue* within the body S of the

original&r-loop is represented by a *continue* node (derived from *goto* statements) whose target is the label c in the code in Figure 5.

To preserve the original source, the loop is actually represented by a for-loop node (derived from the general loop construct) in the intermediate form. Similar to the example of the while-loop above, a *for*-loop node has three additional semantic attributes for the respective expressions of the loop: start, end, and step expression. Semantic edges denote only the source code organization of the *for*-loop; the actual flow of execution is explicitly and separately represented. Otherwise we would have to distinguish between an Ada and a C for-loop node since the two languages have different interpretations of a *for*-loop.

```
     Ada   for-loop                    C   for-loop
for I in El   ..E2 loop S; end loop;   for (El  ; E2; E3)  S;

Intermediate    form:              Intermediate    form:
I := El; bound := E2;              El;
loop                               loop
   if I  <= bound then                if E2 then
      S;                                 S;
      I := I + 1;                        <<label c>>E3;
   else                               else
      exit;                              exit;
   end if;                            end if;
end loop;                          end  loop;
```

Figure 5. Unified loop representation.

C has many idiosyncrasies, such as conditional values, assignment expressions, post increment operators, unrestricted gotos, etc. Nevertheless, all of them are represented by the GAST following the above design principles. The interested reader is referred to [Rohr98] for details.

The GAST has been implemented in Ada95, extensively using its object-oriented facilities. Altogether there are about 170 different GAST node types, many of which are considered abstract.

4.2. ERG with Views

For coarse-grained analyses, we use the ERG as described in Section 3.3. However, we added views to the ERG to represent different users' perspectives as well as alternative results of different analyses. Views can also be used to pass only the relevant part of a global graph to an analysis in a uniform and efficient way.

A view is a subgraph of the ERG, i.e., a graph consisting of the subset of the nodes and the edges such that both target and source of any edge in the view belong to the view. Views are a convenient modeling means in different contexts as shown by the following sections.

Views to represent different aspects of the source code.

Call graphs, type graphs, and variable reference graphs are common concepts in reverse engineering. Table 2

shows what these graphs model. Each of them is a view of entities and relationships directly derivable from source code. There are analyses that are interested only in one of these graphs. A dominator analysis, for example, would compute the dominator tree for a given call graph. Likewise in code visualization, a user might want to see only the types to understand class hierarchies. However, in a CBMS, in which we have a set of different tools for different purposes, we are generally interested in all these graphs. Instead of representing these graphs individually, we keep all the information that we extract from source code together in one graph and represent these subgraphs as different views. All the nodes and edges in these views can be extracted from the general graph immediately. This not only increases efficiency but also avoids change anomalies caused by otherwise redundant information. A routine, for example, is part of both the call graph and the variable reference graph and is still contained only once in the general graph.

Table 2. Common graphs in reverse engineering.

Graph	Nodes	Edges
call graph	routines	call relationship
type graph	types	part-of, is-a, or subtype relationships
variable reference graph	routines, variables	set and use relationships

Views as a uniform input and output of analyses.
Often, analyses should operate on part of the graph only. For example, a user may want to see the dominance tree just for a subsystem. It is obvious that a reusable algorithm must be parameterized by the relevant nodes and edges. Views are a means to express which nodes and edges should be considered by the analysis and can act as parameter to each analysis that operates on the ERG. The analysis then can iterate over the nodes and edges of the given view. The result of an analysis can also be represented as a view. This way, we can keep track which information was added by which analysis. Analyses that use views and analyses that produce views can be plugged together.

Views to represent intermediate results. If analyses are computational expensive, intermediate results should be kept if more than one later analysis needs them. This is particularly true in situations when alternative analyses are all built on the results of a previous analysis. Views can be used to save intermediate results. Analyses can then be combined in the form of a network as illustrated in Figure 6.
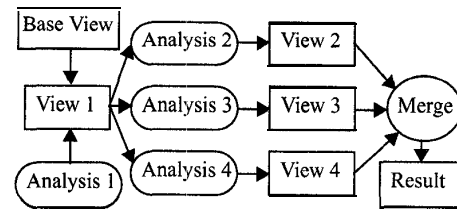

Figure 6. A network of analyses connected by views.

Views to represent alternative results. If different analyses are used that yield alternative results for the same problem, one has to save all the results and still has to be able to distinguish them. This can be achieved by representing the results as different views. Likewise, if users have to validate results of the same analysis, different users can have different opinions. All their decisions should be represented distinctly; this can be achieved by different views. By computing the difference of alternative views, similarities and differences can be revealed.

5. Technical Details

Although the efficient implementation of graph data structures has been discussed extensively in the literature, this section gives insights into the implementation we have chosen for the ERG with views because the addition of views poses additional restrictions on the implementation.

The basic assumptions of our implementation are as follows:

- The average number of outgoing edges of a node is relatively small, thus the graph is very sparse.
- Nodes and edges are rarely added and even less rarely removed. Fast access is a more important factor than efficient addition or removal.
- The number of distinct views is relatively small.

5.1. ERG representation

The ERG is a very sparse graph and is therefore implemented by adjacency lists. Each node has a list of its successors and a list of its predecessors. This redundant information is an optimization because we need fast access not only to the successors but also to the predecessors of a node. The node type is an abstract class of which concrete entities, such as variables and routines, with additional attributes can be derived. This way the ERG is easy to extend.

Edges are first order objects, i.e., they can have their own attributes; that is why they cannot be implicitly represented as simple links between nodes. In analogy to ERG

nodes, the abstract edge class is extended by concrete derived classes with additional attributes. Each edge has at least two attributes: its source and target.

Nodes and edges are saved in two separate dynamic tables. A node table is necessary because we can have nodes that are not successor or predecessor to any other node and thus cannot be attached to the rest of the graph. The edge table is needed because of the view implementation strategy discussed below.

Figure 7 shows the representation of two nodes a and b and an edge from a to b in the view v. The meaning of the view information is discussed in the next section.
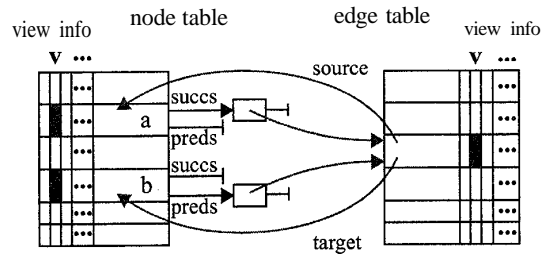

Figure 7. ERG implementation.

5.2. View implementation

Views are conceptually sets of nodes and edges and they could also be represented as such. That is, we could implement them on top of the ERG representation, hence the ERG would not know about views. However, this causes inconsistencies when a node is removed that is designated by a view. That is why views are an integral part of the ERG in our implementation: It allows for removing a node from all views when this node is removed from the ERG.

Sets. Within the ERG there are several ways to implement views. An explicit set representation, e.g., using a linked list, has the disadvantage that the time complexity for checking whether a given entity is in a view is $O(n_V)$ for nodes and $O(e_V)$ for edges (let n_V be the number of nodes and e_V the number of edges in the view). This could be improved by using a sorted tree or a hash list but these representations have an additional space overhead and still do not achieve a constant lookup time.

Bitvectors. Views could be implemented as bitvectors. A bit in this vector indicates whether an edge or node is present in a view. This optimizes the access time to the view information for each entity to O(1). The needed space is $S_V(n, e) = (n + e) \times 1$ [bits] where n is the num-

ber of nodes and e is the number of edges in the ERG. Note that the space for bitvectors depends on the total number of nodes and edges of the ERG, whereas for sets, the space depends only on the view. For views that are not very sparse, bitvectors use less space than explicit representations and offer more efficient data access.

The disadvantage of bitvectors is that each bitvector has to be adjusted when entities are added to or removed from the ERG to allow for an efficient implementation of the union of two views. This makes adding or removing nodes very expensive operations.

Integrated view information. The implementation we chose is based on the idea of bitvectors. However, instead of keeping the view information separate, we integrate the view information with the node and edge tables. A table entry is two-fold: the attributes of the node or edge and its view information given as an array of bits A where $A(i) = 1$ indicates that the respective node or edge is in view i. This achieves the same space and time efficiency as the bitvector implementation but adding or removing nodes or edges entails only changing one entry of the node or edge table. The prize we pay for these advantages is that we restrict the number of possible views. However, we expect a relatively small number of views and the re-use of intermediate views.

Compared to the set implementation of views, iteration over all nodes of a view is less efficient. It is always O(n) where n is the number of nodes *in* the graph whereas for views as sets, iteration depends only on the number of nodes *in* the view. On the other hand, traversals through an ERG (that is, starting with a given node and following all edges in the view) is more efficient when the view information is stored in the table (or in a bitvector) because the test whether a node or edge is in the view takes only O(1) time.

Another advantage of this implementation strategy is that the union of views can be implemented efficiently: The view identifier for view i is represented by a bit mask M_i whose bit i is set (all other bits are 0). To check whether node n is in view i, we simply have to test $B(n)$ AND M_i for a non-zero value, where $B(n)$ is the bit array of node n. The same test can be used to check whether a node is in the union of two views by using the bitwise OR of their view identifiers instead of M_i.

6. Examples of the application of IIR

To illustrate the usefulness of our intermediate representation, and in particular of the concepts of views, this section shows how the IIR would support Kazman's model of architecture recovery and how IIR was used in our own ADT recovery work.

6.1. IIR meets horseshoe

Kazman et al. [Kazm98] present a framework that can accommodate analysis and transformation processes in architecture recovery. This framework is called the horseshoe model and it consists of four different levels:

- source level: source code in textual representation
- code structure level: the AST enriched with control and data flow information
- function level: relationships among functions, data, and files, providing a global system overview
- architectural level: architectural elements, e.g., connectors and components

Two out of these four levels directly correspond to levels in the IIR: The code structure level is represented in the GAST and the function level is represented in the ERG.

The source level of the IIR corresponds to the source files of the system (possibly revision controlled or stored in a CBMS). They are not explicitly stored as part of the IIR, but, instead, each element in the GAST and in the ERG contains a mapping to its position in the source code.

In contrast to the horseshoe model, IIR does not distinguish between the function level and the architectural level but represents both in the ERG because the latter can accommodate both levels using the same mechanism while allowing explicit representations of the mappings between them.

Code structure level. Large systems often consist of components written in different programming languages. To overcome this diversity present at the source code level, these elements should be uniformly represented at the code structure level. This is what the generalized abstract syntax tree representation, GAST, was designed for. Likewise, data and control flow information are an integral part of GAST (the construction of the GSA is still ongoing work).

Functional and architectural levels. As mentioned by Kazman et al., an intermediate representation for architectural reengineering must be suitable for representing concepts at all different levels of abstraction. Among these levels, the IR must allow for drawing explicit and seamless mappings. IIR supports both of these needs and offers additional benefits by virtue of its view mechanism that can help represent, compare, and manipulate multiple levels of abstraction, complementary analyses, and different perspectives associated with various tasks.

Furthermore, the view mechanism can be used to capture alternatives caused by the diverging opinions of experts on debatable cases. E.g., in the context of abstract data type detection, there are cases where it is not clear whether a given routine belongs to an abstract data type.

The view mechanism allows to compare these alternatives, revealing the differences and commonalities.

By capturing specific versions of a system into views, the evolution of the system can be monitored and analyzed.

Finally, in a top-down/bottom-up architecture recovery process, two more kinds of views are important: We want both the expected architecture and the architecture as built be specified in order to draw a mapping between them.

Extensible abstractions,, The ERG allows for adding new concepts to the system abstraction that go beyond the code level elements. In fact, we have added connectors (pipes, RPC, shared memory, etc.) and components (client/server, filter, tasks, etc.) to our implementation of the ERG. This allows for a static view of the architecture. Static and temporal features, as Kazman et al. request, could easily be added as attributes of ERG nodes and edges.

Complementary specifications. A complete architectural description requires a specification of constraints and dynamic aspects, too. IIR does not have immediate expressive means for such specifications. Specific specification languages [Alle94] are better suited to describe these aspects on top of IIR. E.g., static constraints, such as whether two components may interact with each other, can then be checked by investigating the actual communication channels in the IIR. Likewise., path expressions extracted from the IIR could be checked against protocol specifications of connectors. Many of these verifications need human-computer interaction since they are generally undecideable. This enforces the need for an intermediate representation such as IIR that captures human assertions.

6.2. Experience with ADT recovery using IIR

In the context of our work of architectural recovery, the IIR is used to detect atomic architecture components, namely abstract data types (ADT) and state encapsulations [Gira97], and subsystems;. The use of views facilitated the management of different user perspectives as well as the manipulation of the results of different analyses.

Multiple analyses have been proposed to recognize ADTs. When analyzing a system, the user can select some of them according to his experience and the system's characteristics. The result of each selected analysis is stored in a view that is presented to the user for validation. The user can indicate that he/she wants to exclude some entities from ADTs or that certain entities must appear together. Once again, each of these inputs are represented in a distinct view. Then, these views can be combined appropriately to form the input of the next iteration of the selected analysis.

7. Conclusions

In this paper, we introduced the requirements of intermediate representations (IR) for reverse engineering. We presented how they resemble those of IRs for optimizing compilers and how they differ. We then gave an overview of some existent IRs (AST, gated single assignment form, and entity relationship graphs) and rated them with respect to these requirements. We then showed how most of these requirements can be met by integrating and extending existent IRs. These extensions include a generalized AST and a mechanism supporting multiple views on programs. After giving insights on efficient implementation of the proposed IR, we illustrated its usefulness in the context of our work on ADT recovery and by discussing how it can support the horseshoe model proposed by Kazman et al.

Acknowledgments

We would like to thank Jürgen Rohrbach and Thomas Eisenbarth for their contributions and ideas on the intermediate representation. We would also like to thank Erhard Plödereder for his support of this project and his insightful inputs to this research.

References

[Aho86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, techniques and tools.* Addison Wesley, 1986.

[Alle94] R. Allen, D. Garlan. Formalizing Architectural Connection. In *Proc.* of the Int. Conf. on Software Engineering, IEEE Computer Society Press, 1994.

[Alpe98] B. Alpern, M. Wegman, and F. Zadeck. Detecting equality of variables in programs. In *ACM Symposium on Principles of Programming Languages,* 15, pages 1-l 1, 1998.

[Ball90] R. Ballance, A. Maccabe, and K. Ottenstein. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *ACMSIGPLAN Conference on Programming Language Design and Implementation,* pages 257-27 1, June 1990.

[Cimi91] A. Cimitile and U. De Carlini. Reverse Engineering: Algorithms for Program Graph Production. *Journal of Software Practice and Experience,* 21(5):519–537, May 1991.

[Cimi95] Aniello Cimitile, Andrea De Lucia, and Malcolm Munro. Qualifying reusable functions using symbolic execution. In *Working Conference on Reverse Engineering,* pages 178-187, Toronto, Canada, 1995. IEEE Press.

[Cytr91] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependency graph. *ACM Transactions on Programming Languages and Systems,* 13(4), Oct. 1991.

[Gira97] J.F Girard, R. Koschke, and G. Schied. Comparison of abstract data type and abstract state encapsulation detection techniques for architectural understanding. In *Working Conference on Reverse Engineering,* pages 66-75, Amsterdam, The Netherland, Oct. 1997.

[Goos81] G. Goos and W. A. Wulf. Diana Reference Manual. Technical Report CMU-CS-8 1 - 101, Department of Computer Science, Carnegie-Mellon University, 198 1.

[iri] The IRIS toolset. http://laser.cs.umass.edu/tools/iris.html.

[Kazm98] R. Kazman, S. Woods, and S. Carrière. Requirements for Integrating Software Architecture and Reengineering Models: CORUM II. In *Working Conference on Reverse Engineering,* Hawaii, Oct 1998.

[Kinl94] D. A. Kinloch and M. Munro. Understanding C Programs Using the Combined C Graph Representation. In *International Conference on Software Maintenance,* pages 172-180, 1994.

[Klin98] P. Klint and C. Verhoef. Evolutionary software engineering: A component-based approach. In R.N. Horspool, editor, *IFIP WG 2.4 Working Conference: Systems Implementation 2000: Languages, Methods and Tools,* pages 1-l 8. Chapman & Hall, 1998.

[Müll90] H. A. Müller and J. S. Uhl. Composing subsystem structures using (k,2)-partite graphs. In *International Conference on Software Maintenance,* pages 12–19. IEEE, IEEE Computer Society Press, 1990.

[Ping91] K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence Flow Graphs: An Algebraic Approach to Program Dependencies. In *ACM Symposium on Principles of Programming Languages,* pages 67-78, Jan 1991.

[Rohr98] J. Rohrbach. Erweiterung und Generierung einer Zwischendarstellung fir C-Programme. Studienarbeit 1662, University of Stuttgart, Institute of Computer Science, Jan 1998.

[Ruga96] S. Rugaber and L.M. Wills. Creating a research infrastructure for reengineering. In *Working Conference on Reverse Engineering,* pages 98-102, Monterey, 1996.

[Sree94] V.C. Sreedhar and G.R. Gao. Computing $-nodes in linear time using DJ-graphs. ACAPS Technical Memo 75, McGill University, School of Computer Science, Jan 1994.

[Tu95] Peng Tu. *Automatic array privatization and demand-driven symbolic analysis.* Ph.D. thesis, University of Illinois at Urbana-Champaign, 1995.

[Wegm91] M. Wegman and F. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems,* 2(13): 18 1-l 2 1, April 1991.

[Würt96] M. Wtirthner. Entwurf und Implementierung einer Zwischendarstellung für die Analyse von Ada-Programmen. Studienarbeit 1567, University of Stuttgart, Institute of Computer Science, Oct 96.