The Refinement Relation of Graph-Based Generic Programs

Karl Lieberherr[1] and Boaz Patt-Shamir[1,2]

[1] College of Computer Science
Northeastern University
Boston, MA 02115, USA
lieber@ccs.neu.edu
http://www.ccs.neu.edu/home/lieber
[2] Dept. of Electrical Engineering–Systems
Tel Aviv University
Tel Aviv 69978, Israel
boaz@eng.tau.ac.il

Abstract. version 3, Sep. 3, 98
This paper studies a particular variant of Generic Programming, called Adaptive Programming (AP). We explain the approach taken by Adaptive Programming to attain the goals set for Generic Programming. Within the formalism of AP, we explore the important problem of refinement: given two generic programs, does one express a subset of the programs expressed by the other? We show that two natural definitions of refinement coincide, but the corresponding decision problem is computationally intractable (co-NP-complete). We proceed to define a more restricted notion of refinement, which arises frequently in the practice of AP, and give an efficient algorithm for deciding it.

1 Introduction

What is Generic Programming (GP)? The organizers of this Dagstuhl workshop view GP to have the following important characteristics:

- Expressing algorithms with minimal assumptions about data abstractions, and vice versa, thus making them as interoperable as possible.
- Lifting of a concrete algorithm to as a general level as possible without losing efficiency, i.e., the most abstract form such that when specialized back to the concrete case the result is just as efficient as the original algorithm.

GP is about polymorphism and we think that non-traditional kinds of polymorphism lead to particularly useful forms of Generic Programming. By non-traditional kinds of polymorphism we mean that parameterization is over larger entities than classes. In this paper we focus on parametrization with entire class graphs and we outline how Aspect-Oriented Programming and Adaptive Programming are forms of Generic Programming which attempt to satisfy the two characteristics mentioned above. We show the role of traversal strategies in Adaptive Programming by an analogy to Generic Programming and present new results about traversal strategies. We focus on the concept of graph refinement which is important when traversals are specialized. We show that the obvious definition of refinement leads to a co-NP-complete decision problem and we propose a refinement definition which is computationally tractable and useful for practical applications. The results are summarized in Table 1.

Relationship	Complexity	Symbol
path-set-refinement	co-NP-complete	$G_1 \leq_{\mathcal{N}} G_2$
expansion	co-NP-complete	$G_1 \preceq_{\mathcal{N}} G_2$
refinement	polynomial	$G_1 \sqsubseteq_{\mathcal{N}} G_2$

Table 1. Graph relationships for software evolution. $\mathcal{N}$ is a mapping of nodes of G_2 to nodes of G_1. $G_1 \leq_{\mathcal{N}} G_2$ if and only if $G_1 \preceq_{\mathcal{N}} G_2$. $G_1 \sqsubseteq_{\mathcal{N}} G_2$ implies $G_1 \leq_{\mathcal{N}} G_2$.

1.1 Aspect-oriented Programming (AOP)

Aspect-oriented programming [Kic96,KLM+97] is about controlling the tangling of cross-cutting issues. AOP allows to describe different issues separately making minimal assumptions between them. Many evolution problems come from tangled designs/programs. Code for a requirement is spread through many artifacts. In each artifact, code for different requirements is tangled together. Examples of tangling are: 1. Information structure is tangled with behavior. 2. Synchronization code is tangled with sequential code. 3. Code related to remote invocation is tangled with behavior.

AOP controls the tangling by splitting software into cooperating, loosely coupled building blocks which are woven together by a compiler called a weaver. See Fig. 1.

1.2 Adaptive Programming (AP)

Adaptive Programming [Lie92,Lie96] is a special case of AOP: AP = AOP with traversal strategies. One of the building blocks of the aspect-oriented program uses graphs which are referred to by traversal strategies. A traversal strategy defines traversals of graphs without referring to the details of the traversed graphs.

AP adds flexibility and simultaneously simplifies designs and programs. We make a connection between GP (as practiced in the STL community) and AP (see Table 2). In GP, algorithms are parameterized by iterators so that they can be used with several different data structures. In AP, algorithms are parameterized by traversal strategies so that they can be used with several different data structures. Traversal strategies can be viewed as a form of iterators which are more flexible than ordinary iterators. For details on the parameterization mechanism in AP, see [Lie96,ML98].

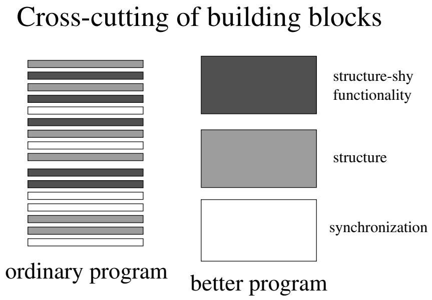

Fig. 1. AOP solution to control tangling

Kind	**Algorithms**	**Glue**	**Graphs**
GP(STL)	Algorithms	Iterators	Data Structures
AP	Adaptive Algorithms	Traversal Strategies	Class Graphs

Table 2. Correspondence between GP and AP

2 Traversal Strategies

Traversal strategies (also called succinct traversal specifications) are a key concept of AP. They were introduced in [LPS97,PXL95] together with efficient compilation algorithms. The purpose of a traversal strategy is to succinctly define a set of paths in a graph and as such it is a purely graph-theoretic concept. Since there are several works which demonstrate the usefulness of traversal strategies to programming [Lie96,PXL95,AL98,ML98] we are switching now to a mathematical presentation of the concepts underlying strategies without giving many connections to the practice of programming.

There are different forms of traversal strategies the most general of which are described in [LPS97]. In this paper we only consider a special case: positive strategy graphs. Positive strategy graphs express the path set only in a positive way without excluding nodes and edges. Positive strategies are defined in terms of graphs and interpreted in terms of expansions.

2.1 Definitions

A directed graph is a pair (V, E) where V is a finite set of *nodes*, and $E \subseteq V \times V$ is a set of *edges*. Given a directed graph $G = (V, E)$, a *path* is a sequence $p = \langle v_0 v_1 \ldots v_n \rangle$, where $v_i \in V$ for $0 \leq i \leq n$, and $(v_{i-1}, v_i) \in E$ for all $0 < i \leq n$.

We first define the notion of an **embedded strategy graph**.

Definition 1. *A graph $S = (V_1, E_S)$ with a distinguished source node s and a distinguished target node t is said to be an* embedded strategy graph of *a graph $G = (V_2, E)$ if $V_1 \subseteq V_2$.*

Intuitively, a strategy graph S is a sort of digest of the base graph G which highlights certain connections between nodes. In the applications, a strategy graph plays the role of a traversal specification and the base graph plays the role of defining the class structure of an object-oriented program. For example, a strategy graph could say: Traverse all C-objects which are contained in B-objects which are contained in A-objects. This would be summarized as a graph with three nodes A,B,C and an edge from A to B and an edge from B to C. In this paper, the base graphs are just graphs without the embellishments usually found in a class structure. (In [LPS97,PXL95] it is shown how to generalize the concept of a strategy graph for general class graphs used in object-oriented programs.) To complicate matters, strategy graphs can also play the role of class graphs. In this case refinement between strategy graphs means refinement between class graphs in the sense that we make the object structures more complex while preserving their essential shape.

A strategy graph S of a base graph G defines a path set as follows. We say that a path p is an *expansion* of a path p' if p' can be obtained by deleteing some elements from p. We define $PathSet_{st}(G, S)$ to be the set of all $s - t$ paths in G which are expansions of any $s - t$ path in S.

Unlike embedded strategies, general strategies allow the node sets of the graphs S and G to be disjoint by using a "name mapping" between them.

Next we define the concept of a strategy graph independent of a base graph.

Definition 2. *A* strategy graph *$\mathcal{S}$ is a triple $\mathcal{S} = (S, s, t)$, where $S = (C, D)$ is a directed graph, C is the set of* strategy-graph nodes, *D is the set of* strategy-graph edges, *and $s, t \in C$ are the* source *and* target *of $\mathcal{S}$, respectively.*

The connection between strategies and base graphs is done by a name map, defined as follows.

Definition 3. *Let $S = (C, D)$ be a graph of a strategy graph and let $G = (V, E)$ be a base graph. A* name map *for S and G is a function $\mathcal{N} : C \to V$. If p is a sequence of strategy-graph nodes, then $\mathcal{N}(p)$ is the sequence of base graph nodes obtained by applying $\mathcal{N}$ to each element of p.*

We next define expansion in the presence of a name map.

Definition 4. *Let V_1, V_2 be arbitrary sets, and let $\mathcal{N} : V_2 \to V_1$ be a function. We say that a sequence p_1 of elements of V_1 is an* expansion under $\mathcal{N}$ *of a sequence p_2 of elements of V_2 if $\mathcal{N}(p_2)$ is a subsequence of p_1, where $\mathcal{N}$ is applied to each element in the sequence.*

With this definition, we define the concept of a path set.

Definition 5. *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be directed graphs, let $\mathcal{N} : V_2 \to V_1$ be a function, and let $s, t \in V_2$. $PathSet_{st}(G_1, \mathcal{N}, G_2)$ is defined to be the set of all paths in G_1 which are expansions under $\mathcal{N}$ of any $s - t$ path in G_2.*

The identity of s and t is assumed to be fixed, and we shall omit subscripts henceforth.

Using the terminology above, if the name map is the identity function $\mathcal{I}$, then G_2 is an embedded strategy for G_1. Note, for example, that $PathSet(G, \mathcal{I}, G)$ is exactly the set of all $s - t$ paths in G. (Exercise for the reader: Prove that $PathSet(G, \mathcal{I}, G) = PathSet(G, \mathcal{I}, H)$, where H is the directed graph consisting of the single edge (s, t).)

We now turn to the first definition of the graph refinement relations. For the case of embedded strategy graphs, we say that a strategy graph G_1 is a path-set-refinement of strategy graph G_2 if for all base graphs G_3 for which G_1 and G_2 are strategies, $PathSet(G_3, G_1) \subseteq PathSet(G_3, G_2)$.

Example 1. Strategy graph G_2: Nodes A,B. Edges (A,B). Strategy graph G_1: Nodes A,B,X,Y. Edges (A,X), (X,B), (A,Y), (Y,B). Source A, Target B. Name map is the identity map. G_1 is a path-set-refinement of G_2.

In the presence of name maps, the situation is more complex: First, we need the following technical concept (see Figure 2).

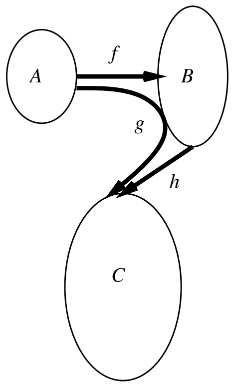

Fig. 2. Illustration for a function h extending g under f.

Definition 6. *Let A, B, C be sets, and let $f : A \to B, g : A \to C, h : B \to C$ be functions. We say that h* extends *g* under *f if for all $a \in A$ we have $h(f(a)) = g(a)$.*

Definition 7. *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be directed graphs, and let $\mathcal{N} : V_2 \to V_1$ be a function. We say that G_1 is a* path-set-refinement under *$\mathcal{N}$ of*

G_2, denoted $G_1 \leq_{\mathcal{N}} G_2$, if for all directed graphs $G_3 = (V_3, E_3)$ and functions $N_1 : V_1 \to V_3$ and $N_2 : V_2 \to V_3$ such that $\mathcal{N}_1$ extends $\mathcal{N}_2$ under $\mathcal{N}$, we have that $PathSet(G_3, \mathcal{N}_1, G_1) \subseteq PathSet(G_3, \mathcal{N}_2, G_2)$.

Note that if $G_1 \leq_{\mathcal{N}} G_2$, then usually G_2 is the "smaller" graph: intuitively, G_2 is less specified than G_1.

We now define another relation for graph refinement, called "expansion." This relation is more useful for exploring properties of graph refinement. For the case of embedded strategy graphs, we say that a strategy graph G_1 is an expansion of strategy graph G_2 if for any path p_1 (from s to t) in G_1 there exists a path p_2 (from s to t) in G_2 such that p_1 is an expansion of p_2. In example 1, $G1$ is an expansion of $G2$.

The general definition of expansion for positive strategies is:

Definition 8. *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be directed graphs, and let $\mathcal{N} : V_2 \to V_1$ be a function. We say that G_1 is an* expansion under $\mathcal{N}$ *of G_2, denoted $G_1 \preceq_{\mathcal{N}} G_2$, if for any path $p_1 \in PathSet(G_1, \mathcal{I}, G_1)$ there exists a path $p_2 \in PathSet(G_2, \mathcal{I}, G_2)$ such that p_1 is an expansion under $\mathcal{N}$ of p_2.*

We now prove equivalence of the notions of "path-set-refinement" and "expansion".

Theorem 1. *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be directed graphs, and let $\mathcal{N} : V_2 \to V_1$ be a function. Then $G_1 \preceq_{\mathcal{N}} G_2$ if and only if $G_1 \leq_{\mathcal{N}} G_2$.*

This theorem tells us that we can use the simpler definition of expansion instead of the more complex definition of path-set-refinement which involves quantification over general graphs.

Proof: Suppose first that $G_1 \preceq_{\mathcal{N}} G_2$. Let G_3 be any graph, and suppose that $\mathcal{N}_1, \mathcal{N}_2$ are as in Definition 7. Let $p_3 \in PathSet(G_3, \mathcal{N}_1, G_1)$. To show that $G_1 \leq_{\mathcal{N}} G_2$, it suffices to prove that $p_3 \in PathSet(G_3, \mathcal{N}_2, G_2)$. This can be seen as follows. By Definition 5, there exists a path $p_1 \in PathSet(G_1, \mathcal{I}, G_1)$ such that p_3 is an expansion of p_1 under $\mathcal{N}_1$, i.e., $\mathcal{N}_1(p_1)$ is a subsequence of p_3. Since $G_1 \preceq_{\mathcal{N}} G_2$, we have by Definition 8 that there exists a path $p_2 \in PathSet(G_2, \mathcal{I}, G_2)$ such that p_1 is an expansion of p_2 under $\mathcal{N}$, i.e., $\mathcal{N}(p_2)$ is a subsequence of p_1. It therefore follows that $\mathcal{N}_1(\mathcal{N}(p_2))$ is a subsequence of p_3, and since $\mathcal{N}_1$ extends $\mathcal{N}_2$ under $\mathcal{N}$, we get $p_3 \in PathSet(G_3, \mathcal{N}_2, G_2)$ as desired.

Next, suppose that $G_1 \leq_{\mathcal{N}} G_2$. Let $p_1 \in PathSet(G_1, \mathcal{I}, G_1)$. We need to prove that there exists a path $p_2 \in PathSet(G_2, \mathcal{I}, G_2)$ such that $\mathcal{N}(p_2)$ is a subsequence of p_1. This follows immediately from Definition 7, which says (by substituting G_2 for G_3, $\mathcal{I}$ for $\mathcal{N}_1$, and $\mathcal{N}$ for $\mathcal{N}_2$) that $PathSet(G_2, \mathcal{I}, G_2) \subseteq PathSet(G_1, \mathcal{N}, G_2)$. ∎

The following problem arises naturally in many applications of strategies.

Graph Path-set-refinement Problem (GPP)
Input: Digraphs $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ with $s_1, t_1 \in V_1$, and a function $\mathcal{N} : V_2 \to V_1$.
Question: Does $G_1 \leq_{\mathcal{N}} G_2$ hold true?

Unfortunately, it turns out that deciding GPP is hard. To prove that, we first consider a weakened version of GPP, defined as follows. Call an edge in a strategy *redundant* if its removal does not change the path sets defined by the strategy. For example, if there exists an edge from the source to the target, then all other edges are redundant, since all source-target paths are expansions of this edge anyway! More formally, an edge (u, v) in a strategy graph G is redundant if $G \leq_{\mathcal{I}} G - \{(u, v)\}$. We define the following decision problem.

Redundant Strategy Edge (RSE)
Input: A Digraph $G = (V, E)$ with source and target nodes $s, t \in V$, and a distinguished edge $(u, v) \in E$.
Question: Is the distinguished edge (u, v) redundant?

Theorem 2. *RSE is co-NP-complete.*

Proof: Consider the complement problem, namely, given G, s, t and (u, v) as above, whether $G - \{(u, v)\}$ is a strict path-set-refinement of G. Call this problem co-RSE. We prove the theorem by showing that co-RSE is NP-complete. We first give an NP algorithm for co-RSE:

1. Generate a sequence p of nodes in V.
2. If p is not a path in G, halt the computation.
3. If p is an expansion under $\mathcal{I}$ of a path in $G - \{(u, v)\}$, halt the computation.
4. Return "(u, v) is not redundant."

Next, we prove that co-RSE is NP-hard. This is done by reducing 3SAT [GJ79] to co-RSE. Fix an instance of 3SAT with m clauses $c_1, \ldots, c_m$ and n variables $x_1, \ldots, x_n$. That is, we are given a Boolean formula, where each clause c_i consists of three literals y_{i1}, y_{i2}, y_{i3}, and each literal is either a variable x_j or its negation $\neg x_j$. We transform the formula to an instance of co-RSE as follows (see Figure 3 for an example).

1. For each clause c_i, create four nodes labeled $d_i, z_{i1}, z_{i2}, z_{i3}$.
2. For each variable x_j, create three nodes labeled $u_j, w_j, \neg w_j$.
3. Create a source node s and a target node t. Below, we identify for convenience $s = d_0$ and $t = u_{n+1}$.
4. For $i = 1, \ldots, m$ and $k = 1, 2, 3$, create edges (z_{ik}, d_i) and (d_{i-1}, z_{ik}).
5. For $i = 1, \ldots, n$, create edges $(u_i, w_i), (u_i, \neg w_i)$ and $(w_i, u_{i+1}), (\neg w_i, u_{i+1})$.
6. For $i = 1, \ldots, m$ and $k = 1, 2, 3$, connect the node representing the literal y_{ik} and its corresponding w node. Specifically, if $y_{ik} = x_j$, create the edge (z_{ik}, w_j), and if $y_{ik} = \neg x_j$, create the edge $(z_{ik}, \neg w_j)$.
7. Create an edge (d_m, u_1).
8. The co-RSE instance is given by G as defined above, where the distinguished edge is (d_m, u_1).

We now need to show that the instance of co-RSE constructed by the transformation is a YES instance if and only if the Boolean formula is satisfiable. First, we define a one-to-one correspondence between truth assignments and $s-t$ paths

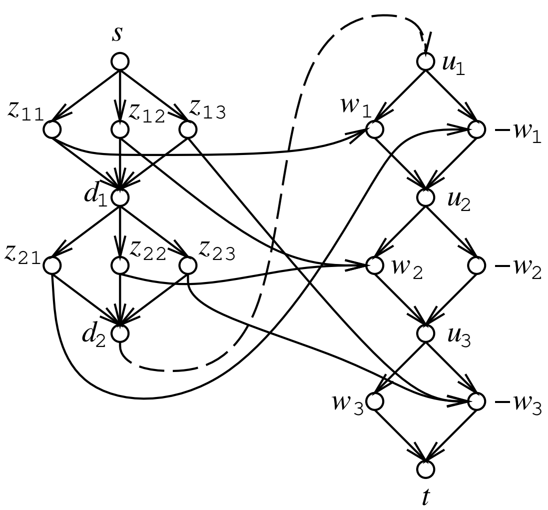

Fig. 3. An example of the reduction: the Boolean formula is $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$. The dashed arrow connecting d_2 and u_1 represents the distinguished edge.

containing the distinguished edge (d_m, u_1): each such path visits exactly one of $w_j, \neg w_j$ for $1 \leq j \leq n$; if w_j is in the path, we will have FALSE assigned to x_j in the corresponding truth assignment, and if $\neg w_j$ is in the path, we assign TRUE to x_j.

Note that any $s - t$ path in $G - \{(d_m, u_1)\}$ (i.e., a path not using the distinguished edge) must use one of the edges created at Step 6 of the transformation.

To complete the proof, observe that there exists a path which is not an expansion of a path in $G - \{(u, v)\}$ if and only if there exists a path passing through the distinguished edge which does not contain both endpoints of any of the edges created in Step 6. This in turn holds (using the corresponding truth assignment) if and only if there is a literal with value TRUE in each clause: the literal are connected to their FALSE values. It follows that the co-RSE instance is a YES instance if and only if the Boolean formula is satisfiable. ∎

A direct implication of Theorem 2 is that the problem of finding a strategy with minimal representation for is hard. With regard to the main point of this paper, we have the following easy corollary.

Corollary 1. *GPP is co-NP-Complete.*

This theorem tells us that when we build tools for AP we cannot use the general definition of expansion since it would result in a slow design tool for large applications.

Proof: By reduction from RSE: given an instance $(G, s, t, (u, v))$ of RSE, define an instance of GPP by $G_1 = G - \{(u, v)\}$, $G_2 = G$, and ask whether $G_1 \leq_{\mathcal{I}} G_2$. ∎

3 The Refinement Relation

In this section we define a more stringent version of the graph path-set-refinement relation, called the *refinement* relation. We argue that this relation is central to software engineering practices. Then we show that the path-set-refinement relation is a generalization of the refinement relation, and we give an efficient algorithm for deciding the refinement relation.

In this section we invoke a mathematical pattern called the *Tractable Specialization Pattern (TSP)* which has several applications in computer science. TSP is defined as follows: Given is a decision problem which has a high complexity but which we need to solve for practical purposes. We define a more strict version of the decision problem for which we can solve the decision problem more efficiently. The goal of the restricted version is that it does not disallow too many of the inputs which are occurring in practice. Fig. 3 shows two applications of TSP. The first is to graph properties in this paper and the second to context-free grammar properties in language theory [HU79]. The second column in Fig. 3 shows a decision problem with its complexity. The third column shows a stricter form of the decision problem with the hopefully lower complexity.

Area	**Decision Problem**	**Stricter**
Graphs	path-set-refinement (co-NP-complete)	refinement (polynomial)
Grammars	ambiguous (undecidable)	LL(1) (polynomial)

Table 3. Applications of the Tractable Specialization Pattern

We first consider the case of embedded strategy graphs. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be directed graphs with V_2 a subset of V_1. We say that G_1 is a refinement of G_2, denoted $G_1 \sqsubseteq G_2$, if for all $u, v \in V_2$ we have that $(u, v) \in E_2$ if and only if there exists a path in G_1 between $\mathcal{N}(u)$ and $\mathcal{N}(v)$ which does not use in its interior a node in V_2.

Example 2. Strategy graph G_2: Nodes A,B,C. Edges (A,B),(B,C). Strategy graph G_1: Nodes A,B,C. Edges (A,C), (C,B), (B,C)). Source A, Target C. Name map is identity map. G_1 is not a refinement of G_2. For the edge from A to B in G_2 there is no path in G_1 from A to B which does not go through C. However, strategy graph G_3: Nodes A,B,C,X. Edges (A,X),(X,B),(B,C) is a refinement of G_2.

The intuition behind the graph refinement relation is that we are allowed to replace an edge with a more complex graph using new nodes. In example 2, we replace the edge (A,B) by the graph (A,X),(X,B), where X is a new node and not one of A,B or C. We first need the following technical concepts to define graph refinement in the presence of a name map.

Definition 9. *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be directed graphs, and let $\mathcal{N} : V_2 \to V_1$ be a function. Given a path p, let first(p) and last(p) denote its* *first and last nodes, respectively. A path p_1 in G_1 (not necessarily an $s-t$ path) is* pure *if $\mathit{first}(p) = \mathcal{N}(u)$ and $\mathit{last}(p) = \mathcal{N}(v)$ for some $u, v \in V_2$, and none of the internal nodes of p is the image of a node in V_2.*

We define refinements as strategies whose pure-path connectivity is the same as the edge-connectivity in the super-strategy. Formally, we have the following definition.

Definition 10. *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be directed graphs, and let $\mathcal{N} : V_2 \to V_1$ be a function. We say that G_1 is a* refinement of G_2 under $\mathcal{N}$*, denoted $G_1 \sqsubseteq_{\mathcal{N}} G_2$, if for all $u, v \in V_2$ we have that $(u, v) \in E_2$ if and only if there exists a pure path in G_1 between $\mathcal{N}(u)$ and $\mathcal{N}(v)$.*

To justify Definition 10, we remark that the notion of strategies is particularly useful in evolution of software, where it is often the case that some crude concepts (and objects) are refined in the course of development. In such scenarios, refining an edge to a more complex structure is usually done with the aid of a refinement. It is important to check whether such an evolution leads to modifying the connectivity structure of the strategy, which is the question of deciding the refinement relation.

The following theorem states that the refinement relation is a subrelation of the path-set-refinement relation.

Theorem 3. *If $G_1 \sqsubseteq_{\mathcal{N}} G_2$, then $G_1 \leq_{\mathcal{N}} G_2$.*

Proof: Suppose that $G_1 \sqsubseteq_{\mathcal{N}} G_2$. By Theorem 1, it is sufficient to prove that $G_1 \preceq_{\mathcal{N}} G_2$. Let p any $s-t$ path in G_2. Decompose $p = p_0 p_1 \cdots p_n$, where each p_i is a pure path (this is possible since s, t are in G_2). By definition, we have that the sequence $s = \mathit{first}(p_0), \mathit{first}(p_1), \ldots, \mathit{first}(p_n), \mathit{last}(p_n) = t$ is a path in G_2. It follows that any $s-t$ path in G_1 is an expansion of an $s-t$ path in G_2, and hence $G_1 \leq_{\mathcal{N}} G_2$. ∎

We now give an algorithm to decide whether $G_1 \sqsubseteq_{\mathcal{N}} G_2$. This is done by a simple graph search algorithm defined as follows.

First Targets Search (FTS)
Input: Digraph $G = (V, E)$ a node $v_0 \in V$, and a set of *targets* $T \subseteq V$.
Output: A subset $T' \subseteq T$ of the targets, such that for each $t \in T'$, there exists a path in G from v_0 to t which does not contain any other target.

It is easy to implement FTS (using BFS or DFS) in linear time, assuming that we can test in constant time whether a node is a target. In Figure 4 we give a BFS-based algorithm, using a FIFO queue Q.

Running FTS can detect superfluous or missing connectivity in G_1, when compared to G_2 under $\mathcal{N}$. The algorithm for deciding the refinement relation proceeds as follows.

Input: $G_1 = (V_1, E_1), G_2 = (V_2, E_2), \mathcal{N} : V_2 \to V_1$.

1. Let $T \subseteq V_1$ be the image of V_2 under $\mathcal{N}$, i.e., $T = \{\mathcal{N}(v) \mid v \in V_2\}$.

```
PROCEDURE FTS(G, v0, T)
  mark v0 visited
  insert v0 to tail of Q
  while Q ≠ ∅
      remove u from head of Q
      if u ∈ T then add u to output set       //...and don't explore this path further
      else for_all v such that (u, v) ∈ E
          if v is not marked visited then
              mark v visited
              insert v to tail of Q
          end_if
      end_if
  end_while
```

Fig. 4. *Algorithm for First Targets Search*

2. For each node $v \in V_2$:
   (a) Perform FTS from $\mathcal{N}(v)$ with targets T in G_1. Let the resulting set be T_v, i.e., $T_v = \text{FTS}(G_1, \mathcal{N}(v), T)$.
   (b) If there exists $u \in V_2$ such that $(v, u) \notin E_2$ and $\mathcal{N}(u) \in T_v$, or $(v, u) \in E$ and $\mathcal{N}(u) \notin T_v$, return "$G_1 \not\sqsubseteq_{\mathcal{N}} G_2$" and halt.
3. Return "$G_1 \sqsubseteq_{\mathcal{N}} G_2$."

The running time of the algorithm is $O(|E_1| \cdot |V_2|)$.

4 Conclusions

We have discussed how Generic Programming through parameterization of programs with entire graph structures (as opposed to only single classes) leads to more flexible programs. We introduced graph theory which is needed to better apply and understand this new form of generic programming, called Adaptive Programming which is itself a special case of the more general Aspect-Oriented Programming.

We introduced the concept of refinement between graphs which has the following applications: It can be applied to check efficiently whether one traversal is a subtraversal of another. It can be applied to check whether one class graph is a generalization of another class graph so that the containment relationships are preserved.

5 Acknowledgements

We would like to thank Mira Mezini, Joshua Marshall, Doug Orleans, and Johan Ovlinger for the ideas they contributed to the paper. This work has been partially supported by the Defense Advanced Projects Agency (DARPA), and Rome Laboratory, under agreement number F30602-96-2-0239. The views and conclusions herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

http://www.ccs.neu.edu/research/demeter/biblio/graph-refine.html contains further information about this paper.

References

[AL98] Dean Allemang and Karl J. Lieberherr. Softening Dependencies between Interfaces. Technical Report NU-CCS-98-07, College of Computer Science, Northeastern University, Boston, MA, August 1998.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability.* Freeman, 1979.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[Kic96] Gregor Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28A(4), December 1996.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, pages 220–242. Springer Verlag, 1997.

[Lie92] Karl J. Lieberherr. Component enhancement: An adaptive reusability mechanism for groups of collaborating classes. In J. van Leeuwen, editor, *Information Processing '92, 12th World Computer Congress*, pages 179–185, Madrid, Spain, 1992. Elsevier.

[Lie96] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X, entire book at www.ccs.neu.edu/research/demeter.

[LPS97] Karl J. Lieberherr and Boaz Patt-Shamir. Traversals of Object Structures: Specification and Efficient Implementation. Technical Report NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA, Sep. 1997.

[ML98] Mira Mezini and Karl Lieberherr. Adaptive plug-and-play components for evolutionary software development. Technical Report NU-CCS-98-3, Northeastern University, April 1998. To appear in OOPSLA '98.

[PXL95] Jens Palsberg, Cun Xiao, and Karl Lieberherr. Efficient implementation of adaptive software. *ACM Transactions on Programming Languages and Systems*, 17(2):264–292, March 1995.