

A New Approach to Compiling Adaptive Programs*

Jens Palsberg[†]

Boaz Patt-Shamir[‡]

Karl Lieberherr[‡]

March 6, 1996

Abstract

An adaptive program can be understood as an object-oriented program where the class graph is a parameter, and hence the class graph may be changed without changing the program. The problem of compiling an adaptive program and a class graph into an object-oriented program was studied by Palsberg, Xiao, and Lieberherr in 1995. Their compiler is efficient but works only in special cases. In this paper we present and prove the correctness of a compiler that handles the general case. The compiler first computes a finite-state automaton and then uses it to generate efficient code.

1 Introduction

Object orientation has demonstrated that properties such as encapsulation, inheritance, late binding, etc. are useful in the discipline of software engineering. However, object-oriented languages suffer from a certain inherent “rigidity” which makes software re-use sometimes awkward and laborious. This property can be intuitively explained as follows. The key feature of most object-oriented languages is that the description of actions (usually called “methods” in this context) is attached to the description of types (“classes”). While this characteristic property is useful in many cases, it has been observed (see, e.g., [2]) that changes in the structure of data (i.e., class definitions) may necessitate re-writing large portions of the action code (i.e., method definitions), even if essentially the underlying algorithm remains the same.

Let us illustrate this point with a simple example. Consider the following two scenarios. In one scenario, we are given a data structure named *company* which describes fully a commercial firm, and our task is to write a function *sumSalary* which computes the total sum of salaries on *company*'s payroll. In the second scenario, we are given a data structure called *airplane* which describes the current state of an airplane, and our task is to write a function *sumWeight* which finds the current total cargo weight. Naturally, the *company* and *airplane* structures are different, and it seems that there is no escape from

*A preliminary version of this paper appears in the Proceedings of the European Symposium on Programming, April 1996 (Springer-Verlag Lecture Notes on Computer Science).

[†]MIT Laboratory for Computer Science, NE43-340, 545 Technology Square, Cambridge, MA 02139, USA. Email: `palsberg@theory.lcs.mit.edu`.

[‡]Northeastern University, College of Computer Science, 161 Cullinane Hall, Boston, MA 02115-9959, USA. Email: `{boaz,lieber}@ccs.neu.edu`.

writing each of the two functions *sumSalary* and *sumWeight* from scratch. However, after a second thought (or perhaps after writing dozens of functions...) one sees that *sumSalary* and *sumWeight* are doing essentially the same thing. Loosely speaking, the algorithm for both *sumSalary* and *sumWeight* is as follows: “given an object, scan all its sub-objects of a certain kind, and apply a (commutative and associative) combining operation to these subobjects to obtain the final result.” The difference between the code for *sumSalary* and *sumWeight* is due solely to the difference in the specific structure of the input, and not to differences in the underlying algorithms. Moreover, the detailed description of the data structures (which contributes most of the complexity in the code in the examples above) is, in fact, readily provided to the programmer! It would be desirable to enable programmers to specify a generic algorithm which could be automatically tailored to fit the application at hand according to a given description of the structure of the application.

Scenarios such as the one sketched above (which are quite common in the practice of software development) constitute the main motivation for *adaptive programs* [7, 10, 6]. Informally (a formal description is given in Section 2), an adaptive program is a program where the complete description of its data structures is a parameter. Employing the idea of object orientation, actions are associated with types, and in adaptive programs this means that *action code is associated with partially-specified data structures*. Of course, an adaptive program cannot be executed. To get an executable program, an adaptive program has to be *specialized*, in the sense of partial evaluation [5], with a complete description of the actual data structures to be used.

Let us outline the way adaptive programs can be used in our example from above. The basic concepts of adaptive programs will be informally introduced as we proceed. Adaptive programs consist of *traversal specifications* and *code wrappers*. Traversal specifications select objects according to their classes, and code wrappers associate actions with the selected objects. For example, a traversal specification of the form $[A, B]$ is interpreted as “all objects of class B which are subobjects of an object of class A .” With the proper code wrapper attached to class B , the interpretation of the adaptive program could be “for each object of class B contained in the class A object, add its value to a *sum* variable.”

In Figure 1 we give a complete description of the adaptive program informally sketched above, which illustrates the concise nature of this language. The advantage of adaptive programs is that they adapt automatically to changes in the class structure. For each particular application, we just need to provide a class graph that describes it and a renaming which maps adaptive-program identifiers to class names from the graph. A class graph is a labeled directed graph which expresses the “has-a” and “is-a” relations among the classes. The combination of an adaptive program and a class graph contains all the details required for execution: the operations that are to be applied to objects are fully defined, and the desired traversals (in the *company* example, finding all subobjects of type *salary*) can be automatically generated from the class graph.

Adaptive programming is related to functional programming with iterators and folders. Instead of writing a traversal specification, one might first use a traversal routine to extract a list of the relevant objects, and then do a fold on that list. The advantage of adaptive programming is that the traversal routine is succinctly specified and automatically generated from the traversal specification. This is particularly convenient when we want to change the class graph but not the traversal specification. An

OPERATION void add(counter& total)		
TRAVERSE		traversal specification
[Container, Item]		meaning: find all Item subobjects of Container
WRAPPER Item		behavior (C++ code)
(@ total = total + value; @)		
RENAME	company scenario	RENAME
add => sumSalary,		airplane scenario
Container => Company,		add => sumWeight,
Item => Salary		Container => Airplane,
		Item => Weight

Figure 1: Top: an adaptive program. Bottom: renamings for two scenarios.

advantage of typed functional programming is that iterators and folders can be defined at a meta-level as type-dependent functions. This, however, requires the set of types to be smaller than the untyped set of class graphs that we use in this paper. If the advantages of adaptive and functional programming were to be combined, a useful first step would be to define a typed universe of class graphs, where the types provide more information than, say, meta-classes. We leave such developments to future work.

Syntax for traversal specifications, etc. can easily be added to an existing object-oriented language. See [6] for numerous examples of adaptive programming in an extension of C++.

Systems which support adaptive programming have been available since 1991, and are being successfully used at Northeastern University, Xerox PARC, and other places [1]. The core of the compiler provided by these tools was presented and proved correct in [10]. The current compiler, despite being quite useful in many practical cases, is not general in the sense that there are certain combinations of adaptive programs and class graphs which the compiler rejects. If a program and a class graph cannot be compiled, then the program has to be rewritten (as discussed in [10]). This defeats the original motivation of adaptive programs, namely the automation of adaptiveness.

In this paper, we present a new compiler which is applicable to combinations of adaptive program and class graph which could not be dealt with by the old compiler. Informally, the main idea is as follows. While the old compilation algorithm uses the class graph directly to generate traversal code, the new compiler uses the class graph to construct a finite automaton which is used to generate the traversal code. The concept of intermediate automaton enables us to apply standard minimization techniques to ensure that the size of the traversal code is optimal. We give two variants of the basic compiler, which differ in the way the automaton is computed. One variant is *general*, i.e., it works for any combination of adaptive program and class graph; unfortunately, this variant may require exponential running time to compute. The second variant works for an important subclass of the adaptive programs (including programs which could not be compiled by the old compiler), and for these programs it generates efficient code in *polynomial time*.

We prove the correctness of the compiler with respect to the original semantics for adaptive programs, as described in [10, 6]. Our proof consists of two stages. First, we define a variant of the original semantics, and prove that it is equivalent to the original one. Then we show how to construct automata which implement the new semantics. Informally, the purpose of defining the new semantics

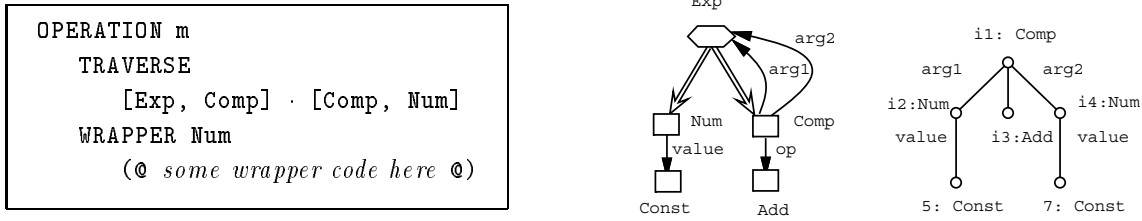


Figure 2: Left: an adaptive program. Middle: a class graph. Right: an object graph.

is to help us to deal with the subclass relation in the construction.

The remainder of this paper is organized as follows. In Section 2 we define basic notions and recall the original semantics of adaptive programs. In Section 3 we give a new semantics of adaptive programs and prove that it is equivalent to the old one, and in Section 4 we give a compilation algorithm for adaptive programs and prove it correct with respect to the new semantics.

2 The Semantics of Adaptive Programs

In this section we recall from [10] the definitions of class graphs, object graphs, paths, traversal specifications, wrappers, and the semantics of adaptive programs. We also define the semantics of an object-oriented target language. The target language is slightly different from the one used in [10].

As a running example throughout the paper we consider the adaptive program, the class graph, and the object graph in Figure 2.

2.1 Class and Object Graphs

A labeled directed graph is a triple (V, E, L) where V is a set of nodes, L is a set of labels, and E is a set of edges where $E \subseteq V \times L \times V$. If $(u, l, v) \in E$, then u is the source, l is the label, and v is the target of (u, l, v) . We will write (u, l, v) as $u \xrightarrow{l} v$.

In this paper we will be interested in special kinds of graphs, called class graphs and object graphs, defined as follows. (The present definitions are akin to those of [8] and [9].)

Fix a set **ClassName**, the set of class names which can be used in class graphs. The predicate **Abstract** is true for names of abstract classes, and it is false otherwise. If a class is not abstract, it is said to be *concrete*.

A *class graph* is a finite labeled directed graph, intended to represent the (static) class structure of a program. Formally, it is defined as follows. The node set is a subset of **ClassName**. Each edge is labeled by an element of $L = N \cup \{\diamond\}$, where $\diamond \notin N$. We assume that (N, \leq) is a totally ordered set of labels. If $l \in N$, then the edge $u \xrightarrow{l} v$ indicates that the class u has an instance variable with name l and with type v . Such an edge is called a *construction* edge. We require that the construction edges outgoing from a node are labeled with distinct labels.

The edges $u \xrightarrow{\diamond} v$ are called *subclass* edges; they represent the fact that v is a subclass of u . In a

class graph, only abstract classes have outgoing subclass edges. A class graph is *flat* if for every node u where $\mathbf{Abstract}(u)$, all outgoing edges are subclass edges. Following [10] we henceforth assume that all class graphs are flat.

For example, in the class graph of Figure 2, **Exp** is an abstract class (indicated by a hexagon), and the other four classes are concrete (indicated by rectangles). The edges from **Exp** to **Num** and **Comp** are subclass edges (indicated by double arrows), and the other four edges are construction edges (indicated by regular arrows). Clearly, the class graph is flat.

An *object graph* is a finite labeled directed graph, intended to represent a run-time object structure. Formally, it is defined as follows. Each node represents an object, and the function **Class** maps each node to “its class,” that is, the name of a concrete class. Each edge is labeled by an element of N . The edge $u \xrightarrow{l} v$ indicates that the object represented by u has a component object represented by v . For each node u and each label $l \in N$, there is at most one outgoing edge from u with label l . For example, the object graph in Figure 2 contains six nodes, each representing an object.

2.2 Paths

Given a graph $G = (V, E, L)$, a *path* is a sequence $v_0 l_1 v_1 l_2 \dots l_n v_n$ where $v_0, \dots, v_n \in V$, and for all $0 \leq i < n$ we have that $v_i \xrightarrow{l_{i+1}} v_{i+1} \in E$, and $l_1, \dots, l_n \in L$. We call v_0 and v_n the *source* and *target* of the path, respectively. If $p_1 = v_0 \dots v_i$ and $p_2 = v_i \dots v_n$, then we define the concatenation $p_1 \cdot p_2 = v_0 \dots v_i \dots v_n$. Notice that $p_1 \cdot p_2$ contains only one copy of the meeting point v_i . Let P_1 and P_2 be sets of paths such that all paths in P_1 have target v , and all paths of P_2 have source v . Then we define

$$P_1 \cdot P_2 = \{p \mid p = p_1 \cdot p_2 \text{ where } p_1 \in P_1 \text{ and } p_2 \in P_2\} .$$

For the remainder of this subsection, let R denote an arbitrary set of paths of a given class graph. We are mainly interested in the paths obtained by removing a prefix containing only \diamond -labeled edges. First, we define an auxiliary function **Reduce**: intuitively, $\mathbf{Reduce}(R)$ is the set of paths obtained by removing all partial \diamond prefixes from each path in R . Formally, for a path set R we define

$$\mathbf{Reduce}(R) = \{v_n \dots v_{n+m} \mid \exists v_0, v_1 \dots v_{n-1} \text{ such that } v_0 \diamond v_1 \diamond \dots \diamond v_n \dots v_{n+m} \in R\}$$

Using **Reduce**, we define $\mathbf{Select}(R, u)$ to be the set of suffixes of paths in R that start with a given node u after skipping a leading \diamond -labeled prefix. Formally:

$$\mathbf{Select}(R, u) = \{v_0 \dots v_n \mid v_0 \dots v_n \in \mathbf{Reduce}(R), v_0 = u\} .$$

Using **Select**, we define $\mathbf{Car}(R, u)$ to be the set of the first edges in $\mathbf{Select}(R, u)$, and for a given label l , we define $\mathbf{Cdr}(l, R, u)$ to be the set of tails of $\mathbf{Select}(R, u)$ where the head has label l . Formally:

$$\begin{aligned} \mathbf{Car}(R, u) &= \{v_0 \xrightarrow{l_1} v_1 \mid v_0 l_1 v_1 \dots v_n \in \mathbf{Select}(R, u)\} \\ \mathbf{Cdr}(l, R, u) &= \{v_1 \dots v_n \mid v_0 l_1 v_1 \dots v_n \in \mathbf{Select}(R, u), l_1 = l\} . \end{aligned}$$

2.3 Traversal Specifications

Fix a class graph $G = (V, E, N \cup \{\diamond\})$. A *traversal specification* denotes a set of paths. Formally, it is an expression generated by the grammar

$$D ::= [A, B] \mid D \cdot D \mid D + D$$

where $A, B \in V$.

The semantics of traversal specifications is summarized in the following table.

Specification D	$\text{PathSet}(D)$	$\text{Source}(D)$	$\text{Target}(D)$
$[A, B]$	All paths from A to B	A	B
$D_1 \cdot D_2$	$\text{PathSet}(D_1) \cdot \text{PathSet}(D_2)$	$\text{Source}(D_1)$	$\text{Target}(D_2)$
$D_1 + D_2$	$\text{PathSet}(D_1) \cup \text{PathSet}(D_2)$	$\text{Source}(D_1)$	$\text{Target}(D_1)$

For a traversal specification to be meaningful, it has to be *well formed*. A traversal specification is well formed if (1) it determines a *source* node and a *target* node, (2) each concatenation has a “meeting point,” and (3) each union of a set of paths preserves the source and the target. Formally, the predicate **WF** is defined in terms of the two functions **Source** and **Target** given in the table above, and the following recursive definition.

$$\begin{aligned} \text{WF}([A, B]) &= \text{true} \\ \text{WF}(D_1 \cdot D_2) &= \text{WF}(D_1) \wedge \text{WF}(D_2) \wedge \text{Target}(D_1) =_{\text{nodes}} \text{Source}(D_2) \\ \text{WF}(D_1 + D_2) &= \text{WF}(D_1) \wedge \text{WF}(D_2) \wedge \\ &\quad (\text{Source}(D_1) =_{\text{nodes}} \text{Source}(D_2)) \wedge (\text{Target}(D_1) =_{\text{nodes}} \text{Target}(D_2)) \end{aligned}$$

If G is a class graph and D is a well-formed traversal specification, then $\text{PathSet}_G(D)$ is a set of paths in G from $\text{Source}(D)$ to $\text{Target}(D)$, as defined in the table above. We usually omit the subscript G when it is clear from the context.

The following basic lemma is taken from [10].

Lemma 2.1 *If $\text{WF}(D)$, then (i) $\text{PathSet}(D)$ is well defined and (ii) each path in $\text{PathSet}(D)$ starts in $\text{Source}(D)$ and ends in $\text{Target}(D)$.*

We henceforth assume that all traversal specifications are well formed. We shall use traversal specifications to denote path sets in class graphs and object graphs.

We can represent a path by the string of its (node and edge) labels. Viewing path sets as sets of strings, we may represent path sets by regular expressions over the alphabet $\text{ClassName} \cup N \cup \{\diamond\}$. For example, let D be the traversal specification of the adaptive program in Figure 2, and let G be the class graph from Figure 2. Employing standard notation for regular expressions [3], and denoting by $L(E)$ the language defined by a regular expression E , we have that

$$\text{PathSet}_G(D) = L(\text{Exp}(\diamond \text{Comp}(\text{arg1} + \text{arg2}) \text{Exp})^+ \diamond \text{Num}) .$$

2.4 Adaptive Programs

Following [10], we define adaptive programs as follows. First, define a *wrapper map* to be a mapping of class names to code segments called wrappers (the idea is that when an object is visited during the traversal of an adaptive program, the appropriate wrapper code will be executed). Now, an *adaptive program* is a pair (D, W) , where D is a traversal specification, and W is a wrapper map. Intuitively, given an object graph Ω and a node o in Ω , the interpretation of an adaptive program (D, W) is roughly “for the subgraph of Ω reachable from o : traverse the objects on paths induced by D in depth-first order, and execute the wrapper code specified by W for each object visited.” Formally, the semantics is given by the function **Run** defined as follows.

$$\begin{aligned} \text{Run}(D, W)(G, \Omega, o) &= \text{Execute}_W(\text{Traverse}(\text{PathSet}_G(D), \Omega, o)) \\ \text{where } \text{Traverse}(R, \Omega, o) &= \begin{cases} H & \text{if } \Omega \vdash_s o : R \triangleright H, \text{ for some } H \\ \perp & \text{otherwise} \end{cases} \end{aligned}$$

If Ω is an object graph, o a node in Ω , R a path set over G , and H a sequence of objects, then the judgment

$$\Omega \vdash_s o : R \triangleright H$$

means that when traversing the object graph Ω starting in o , and guided by the path set R , then H is the *traversal history*, that is, the sequence of objects that are traversed. Formally, this holds when the judgment is derivable using the following rule:

$$\frac{\Omega \vdash_s o_i : \text{Cdr}(l_i, R, \text{Class}(o)) \triangleright H_i \quad \forall i \in 1..n}{\Omega \vdash_s o : R \triangleright o \cdot H_1 \cdot \dots \cdot H_n} \quad \begin{array}{l} \text{if } \text{Car}(R, \text{Class}(o)) = \\ \{\text{Class}(o) \xrightarrow{l_i} w_i \mid i \in 1..n\}, \\ o \xrightarrow{l_i} o_i \text{ is in } \Omega, i \in 1..n, \text{ and} \\ l_j < l_k \text{ for } 1 \leq j < k \leq n. \end{array}$$

The label s of the turnstile indicates “semantics.” Notice that for $n = 0$, the rule is an axiom; it is then simply

$$\frac{}{\Omega \vdash_s o : R \triangleright o} \quad \text{if } \text{Car}(R, \text{Class}(o)) = \emptyset.$$

Notice that **Traverse** is well defined: if both $\Omega \vdash_s o : R \triangleright H_1$ and $\Omega \vdash_s o : R \triangleright H_2$, then $H_1 = H_2$. This can be proved by induction on the structure of the derivation of $\Omega \vdash_s o : R \triangleright H_1$.

The call $\text{Execute}_W(H)$ executes in sequence the wrappers for the class of each of the objects in H . We leave Execute_W unspecified, since its definition depends on the language in which the code wrappers are written.

Example. Let

$$R = \text{Exp} (\diamond \text{Comp} (\text{arg1} + \text{arg2}) \text{Exp})^+ \diamond \text{Num} .$$

We get:

$$\text{Car}(R, \text{Class}(i1)) = \text{Car}(R, \text{Comp})$$

$$\begin{aligned}
&= \left\{ \text{Comp} \xrightarrow{\text{arg1}} \text{Exp}, \text{Comp} \xrightarrow{\text{arg2}} \text{Exp} \right\} \\
\text{Cdr}(\text{arg1}, R, \text{Class}(i1)) &= L(\text{Exp} (\diamond \text{Comp} (\text{arg1} + \text{arg2}) \text{Exp})^* \diamond \text{Num}) \\
\text{Cdr}(\text{arg2}, R, \text{Class}(i1)) &= L(\text{Exp} (\diamond \text{Comp} (\text{arg1} + \text{arg2}) \text{Exp})^* \diamond \text{Num})
\end{aligned}$$

Let R' denote $\text{Cdr}(\text{arg1}, R, \text{Class}(i1)) = \text{Cdr}(\text{arg2}, R, \text{Class}(i1))$. Clearly, $\text{Car}(R', \text{Class}(i2)) = \emptyset$ and $\text{Car}(R', \text{Class}(i4)) = \emptyset$. Let Ω be the object graph in Figure 2. Assuming $\text{arg1} < \text{arg2}$ in the total order of the labels, we get the following derivation:

$$\frac{\Omega \vdash_s i2 : R' \triangleright i2 \quad \Omega \vdash_s i4 : R' \triangleright i4}{\Omega \vdash_s i1 : R \triangleright i1 i2 i4}$$

Thus, the traversal history is $i1 i2 i4$.

2.5 The Target Language

We will compile adaptive programs into an object-oriented target language without inheritance. A program in the target language is a partial function which maps a pair of a class name and a method name to a method. A method is a tuple of the form $\langle l_1.m_1, \dots, l_n.m_n \rangle$, where $l_1 \dots l_n \in N$ and $m_1 \dots m_n$ are method names. When invoked, such a method executes by invoking $l_i.m_i$ in order.

If Ω is an object graph, o a node in Ω , m a method name, P a program in the target language, and H a sequence of objects, then the judgment

$$\Omega \vdash_c o : m : P \triangleright H$$

means that when sending the message m to o , we get a traversal of the object graph Ω starting in o so that H is the traversal history. Formally, this holds when the judgment is derivable using the following rule:

$$\frac{\Omega \vdash_c o_i : m_i : P \triangleright H_i \quad \forall i \in 1..n}{\Omega \vdash_c o : m : P \triangleright o \cdot H_1 \cdot \dots \cdot H_n} \quad \begin{array}{l} \text{if } P(\text{Class}(o), m) = \langle l_1.m_1 \dots l_n.m_n \rangle \\ \text{and } o \xrightarrow{l_i} o_i \text{ is in } \Omega, i \in 1..n. \end{array}$$

The label c of the turnstile indicates ‘‘code’’. Intuitively, the rule says that when sending the message m to o , we check if o understands the message, and if so, we invoke the method. Notice that for $n = 0$, the rule is an axiom; it is then simply

$$\frac{}{\Omega \vdash_c o : m : P \triangleright o} \quad \text{if } P(\text{Class}(o), m) = \langle \rangle.$$

Given a program in the target language, it is straightforward to generate, for example, a C++ program.

3 A Simplified Semantics of Adaptive Programs

In this section we specify a new semantics of adaptive programs, and prove that it is equivalent to the one given in Section 2. The main difference between the new and the old semantics is the way they treat the subclass relation. To emphasize the difference, we use the term ‘‘words’’ for paths without subclass edges. Informally, the idea is as follows.

The semantics of adaptive programs which was given in Section 2 has the following property. When a path set is used to guide a traversal, \diamond -labels are skipped along the way by the operations **Car** and **Cdr**. In this section, we define a simpler semantics which has the property that all \diamond -labels are removed before the traversal begins. The new semantics greatly simplifies the compiling algorithm presented in Section 4. Our notion of word is related to that of “calling path” in [4].

Our first step is to define the functions transforming path sets into strings (words), while deleting abstract classes. Define a *word* to be a sequence $v_0 l_1 v_1 l_2 \dots v_n$ where v_0, \dots, v_{n-1} are names of concrete classes, $l_1, \dots, l_{n-1} \in N$, and v_n is the name of either an abstract or a concrete class. Next, we define the function **SimplifyPath** which maps paths to words as follows. Given a path p , the function **SimplifyPath** is the string obtained from p by removing all \diamond labels and abstract class names, except for the last class name in p . Observe that if p is a path in a flat class graph, then **SimplifyPath**(p) is a word. To see that, recall that in flat class graph, every outgoing edge of an abstract class is a subclass edge, and every outgoing edge of a concrete class is a construction edge. Thus, in a path, except for the last class, a class is abstract if and only if the following label is \diamond . Finally, for a path set R , we define $\text{Simplify}(R) = \{\text{SimplifyPath}(p) \mid p \in R\}$.

Example. Let D be the traversal specification of the adaptive program in Figure 2, and let G be the class graph from Figure 2. We have:

$$\text{Simplify}(\text{PathSet}_G(D)) = L((\text{Comp}(\text{arg1} + \text{arg2}))^+ \text{Num}) .$$

Next, we define traversal of objects in terms of strings. Let R denote a set of strings. We use the functions **First** and **Chop**, defined as follows:

$$\begin{aligned} \text{First}(R) &= \{x \mid \exists \alpha.(x\alpha \in R)\} \\ \text{Chop}(R, x) &= \{\alpha \mid x\alpha \in R\} . \end{aligned}$$

If Ω is an object graph, o a node in Ω , R a word set, and H a sequence of objects, then the judgment

$$\Omega \vdash_n o : R \triangleright H$$

means that when traversing the object graph Ω starting in o , and guided by the word set R , then H is the traversal history. Formally, this holds when the judgment is derivable using the following rule:

$$\frac{\Omega \vdash_n o_i : \text{Chop}(\text{Chop}(R, \text{Class}(o)), l_i) \triangleright H_i \quad \forall i \in 1..n}{\Omega \vdash_n o : R \triangleright o \cdot H_1 \cdot \dots \cdot H_n} \quad \begin{array}{l} \text{if } \text{First}(\text{Chop}(R, \text{Class}(o))) = \{l_i \mid i \in 1..n\}, \\ o \xrightarrow{l_i} o_i \text{ is in } \Omega, i \in 1..n, \text{ and} \\ l_j < l_k \text{ for } 1 \leq j < k \leq n. \end{array}$$

The label n of the turnstile indicates “new semantics.”

For example, let

$$S = L((\text{Comp}(\text{arg1} + \text{arg2}))^+ \text{Num}) .$$

Notice that S is a set of words now. We get:

$$\begin{aligned} \text{First}(\text{Chop}(S, \text{Class}(i1))) &= \text{First}(\text{Chop}(S, \text{Comp})) \\ &= \{\text{arg1}, \text{arg2}\} \\ \text{Chop}(\text{Chop}(S, \text{Class}(i1)), \text{arg1}) &= L((\text{Comp}(\text{arg1} + \text{arg2}))^* \text{Num}) \\ \text{Chop}(\text{Chop}(S, \text{Class}(i1)), \text{arg2}) &= L((\text{Comp}(\text{arg1} + \text{arg2}))^* \text{Num}) . \end{aligned}$$

Let S' denote $\text{Chop}(\text{Chop}(S, \text{Class}(i1)), \text{arg1}) = \text{Chop}(\text{Chop}(S, \text{Class}(i1)), \text{arg2})$. Clearly, we have that $\text{First}(\text{Chop}(S, \text{Class}(i2))) = \emptyset$ and $\text{First}(\text{Chop}(S, \text{Class}(i4))) = \emptyset$. Let Ω be the object graph in Figure 2. Carrying on our assumption that $\text{arg1} < \text{arg2}$, we get the following derivation:

$$\frac{\Omega \vdash_n i2 : S' \triangleright i2 \quad \Omega \vdash_n i4 : S' \triangleright i4}{\Omega \vdash_n i1 : S \triangleright i1 i2 i4}$$

Thus, in this case, the new semantics gives the same traversal history as the old semantics. We now prove that the new semantics is equivalent to the one given in Section 2. We start with two lemmas.

Lemma 3.1 *For a path set R and a concrete class u ,*

$$\{l \mid (u \xrightarrow{l} v) \in \text{Car}(R, u) \text{ for some } v\} = \text{First}(\text{Chop}(\text{Simplify}(R), u)) .$$

Proof. Suppose first $(u \xrightarrow{l} v) \in \text{Car}(R, u)$ for some v . Then there is a path $ul\alpha \in \text{Select}(R, u)$ for some α , and hence $ul\alpha \in \text{Reduce}(R)$. Thus we can find v_1, \dots, v_n for some $n \geq 0$ such that $v_1 l_1 \dots v_n l_n ul\alpha \in R$ and $l_i = \diamond$ for $i \in 1..n$. Moreover, since u is concrete, there exists α' such that $ul\alpha' = \text{SimplifyPath}(v_1 l_1 \dots v_n l_n ul\alpha)$, so $ul\alpha' \in \text{Simplify}(R)$, and hence $l\alpha' \in \text{Chop}(\text{Simplify}(R), u)$. We conclude that $l \in \text{First}(\text{Chop}(\text{Simplify}(R), u))$.

Suppose then that $l \in \text{First}(\text{Chop}(\text{Simplify}(R), u))$. Now we can find a word α such that $ul\alpha \in \text{Simplify}(R)$, and since u is concrete there must be v_1, \dots, v_n for some $n \geq 0$ such that $v_1 l_1 \dots v_n l_n ul\alpha \in R$ and $l_i = \diamond$ for $i \in 1..n$. Moreover, $ul\alpha \in \text{Reduce}(R)$ so $ul\alpha \in \text{Select}(R, u)$, and hence there is some v such that $(u \xrightarrow{l} v) \in \text{Car}(R, u)$. ■

Lemma 3.2 *For a path set R , a concrete class u , and a label $l \in N$,*

$$\text{Simplify}(\text{Cdr}(l, R, u)) = \text{Chop}(\text{Chop}(\text{Simplify}(R), u), l) .$$

Proof. Suppose first that $p \in \text{Simplify}(\text{Cdr}(l, R, u))$. We can then find $p' \in \text{Cdr}(l, R, u)$ such that $p = \text{SimplifyPath}(p')$. Hence, $ulp' \in \text{Select}(R, u)$, so $ulp' \in \text{Reduce}(R)$. Thus, we can find v_1, \dots, v_n for some $n \geq 0$ such that $v_1 l_1 \dots v_n l_n ulp' \in R$ and $l_i = \diamond$ for $i \in 1..n$. Moreover, since u is concrete, and $p = \text{SimplifyPath}(p')$, we have $ulp = \text{SimplifyPath}(v_1 l_1 \dots v_n l_n ulp')$, so $ulp \in \text{Simplify}(R)$. We conclude that $p \in \text{Chop}(\text{Chop}(\text{Simplify}(R), u), l)$.

Suppose then that $p \in \text{Chop}(\text{Chop}(\text{Simplify}(R), u), l)$. Clearly, $ulp \in \text{Simplify}(R)$, so since u is concrete, we can find a path p' and v_1, \dots, v_n for some $n \geq 0$ such that $p = \text{SimplifyPath}(p')$ and $v_1 l_1 \dots v_n l_n ulp' \in R$ where $l_i = \diamond$ for $i \in 1..n$. Hence, $ulp' \in \text{Select}(R, u)$, so $p' \in \text{Cdr}(l, R, u)$. We conclude that $p \in \text{Simplify}(\text{Cdr}(l, R, u))$. ■

Theorem 3.3 $\Omega \vdash_s o : R \triangleright H$ if and only if $\Omega \vdash_n o : \text{Simplify}(R) \triangleright H$.

Proof. Suppose first that $\Omega \vdash_s o : R \triangleright H$ is derivable. We proceed by induction on the structure of the derivation of $\Omega \vdash_s o : R \triangleright H$. Since $\Omega \vdash_s o : R \triangleright H$ is derivable, we have that

$$\begin{aligned} H &= o \cdot H_1 \cdot \dots \cdot H_n \\ \text{Car}(R, \text{Class}(o)) &= \{\text{Class}(o) \xrightarrow{l_i} w_i \mid i \in 1..n\} \\ o \xrightarrow{l_i} o_i &\text{ is in } \Omega, i \in 1..n \\ l_j &< l_k \text{ for } 1 \leq j < k \leq n \\ \Omega \vdash_s o_i : \text{Cdr}(l_i, R, \text{Class}(o)) \triangleright H_i &\text{ is derivable for all } i \in 1..n . \end{aligned}$$

From the induction hypothesis we get that

$$\Omega \vdash_n o_i : \text{Simplify}(\text{Cdr}(l_i, R, \text{Class}(o))) \triangleright H_i$$

is derivable for all $i \in 1..n$. From Lemma 3.2 we then get that

$$\Omega \vdash_n o_i : \text{Chop}(\text{Chop}(\text{Simplify}(R), \text{Class}(o)), l_i) \triangleright H_i$$

is derivable for all $i \in 1..n$. From Lemma 3.1 we get that $\text{First}(\text{Chop}(\text{Simplify}(R), \text{Class}(o))) = \{l_i \mid i \in 1..n\}$, so the side condition of the rule for \vdash_n is satisfied. We conclude that $\Omega \vdash_n o : \text{Simplify}(R) \triangleright H$ is derivable.

The converse is proved similarly. ■

4 Compiling Adaptive Programs

The compiler of [10] will reject the adaptive program and class graph of Figure 2, as discussed in [10]. The reason is that the code that would be generated looks as follows.

<pre> CLASS Comp VAR arg1, arg2: Exp METHOD m arg1.m; arg2.m END END </pre>	<pre> CLASS Num METHOD m — Wrapper code here END END </pre>
---	---

This code does not correctly handle objects that are simply `Nums`, such as `i2` in Figure 2. When the message `m` is sent directly to `i2`, it executes the wrapper code even though the execution has not processed any `Comp` object first.

In this section we present a compiling algorithm which can compile all combinations of adaptive programs and class graphs. The algorithm consists of two steps: first, given a class graph G and a traversal specification D , we represent $\text{Simplify}(\text{PathSet}_G(D))$ by a finite state deterministic automaton. Then, using the automaton, code is generated. We give two algorithms to compute the intermediate automaton: the first one is a polynomial-time algorithm which applies only to a special kind of traversal specifications (“product specifications”). The second algorithm applies to all traversal specifications, but its running time may be exponential.

We start with the basic construction for specifications of the form $[A, B]$. For the remainder of this subsection, we fix a class graph $G = (V, E)$ where $V \subseteq \text{ClassName}$, and alphabet $\Sigma = V \cup N \cup \{\diamond\}$.

4.1 The Basic Construction

We now show how, given a traversal specification of the form $[A, B]$ and a class graph G , to compute an automaton which accepts $\text{PathSet}_G([A, B])$. We use the notation of [3]. The basic structure of the automaton does not depend on A and B : only the start and the final states are defined by them. Specifically, define $\text{FA}(G)$ to be the set of non-deterministic automata where:

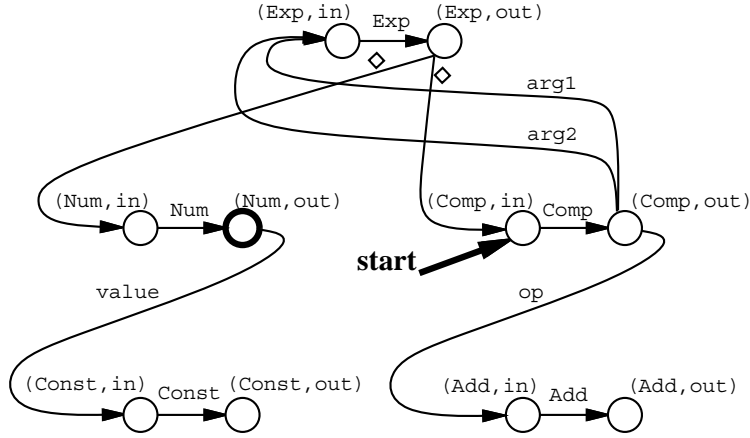


Figure 3: The automaton $\mathbf{Auto}(\mathbf{Comp}, \mathbf{Num})$ corresponding to the class graph of Figure 2 and the specification $[\mathbf{Comp}, \mathbf{Num}]$. The single final state is indicated by a boldface circle.

- the states set is $Q = V \times \{in, out\}$,
- the transition function is defined by

$$\begin{aligned} \delta((C, in), C) &= \{(C, out)\} && \text{for all } C \in V \\ \delta((C, out), l) &= \{(C', in)\} && \text{for each } C \xrightarrow{l} C' \in E \\ \delta(q, a) &= \emptyset && \text{otherwise} \end{aligned}$$

An example of the construction is given in Figure 3.

Computations of automata in $\mathbf{FA}(G)$ are closely related to paths in the class graph. Given an automaton with state set Q , alphabet Σ and transition function δ , define a (partial) computation to be an alternating sequence of states and symbols $\sigma = \langle q^0 a^1 q^1 a^2 q^2 \dots a^n q^n \rangle$, where $q^i \in Q$, $a^i \in \Sigma$, and for all $0 \leq i < n$, $q^{i+1} \in \delta(q^i, a^i)$. Given a computation $\sigma = \langle q^0 a^1 q^1 a^2 q^2 \dots a^n q^n \rangle$, define $\mathbf{String}(\sigma)$ to be the sequence of symbols $a^1 a^2 \dots a^n$. From the definition of $\mathbf{FA}(G)$, we have the following immediate property.

Lemma 4.1 *Let σ be any computation of an automaton in $\mathbf{FA}(G)$. Then $\mathbf{String}(\sigma)$ is a path in the class graph G .*

Given class names $A, B \in V$, we define $\mathbf{Auto}(A, B)$ to be the automaton in $\mathbf{FA}(G)$ with start state (A, in) and a single final state (B, out) .

Let $L(M)$ denote the language accepted by an automaton M . We state the following straightforward fact for later reference.

Lemma 4.2 *Let A, B be two classes in a class graph G . Then*

- (1) $\mathbf{Auto}(A, B)$ can be constructed in time polynomial in $|G|$.
- (2) $L(\mathbf{Auto}(A, B)) = \mathbf{PathSet}(A, B)$.

This simple construction is subsequently used in Sections 4.2 and 4.3.

4.2 Automata for Product Specifications

A traversal specification is said to be *product specification* if it is generated by the grammar

$$D ::= [A, B] \mid D \cdot D$$

where A and B are class names such that $A \neq B$. In this subsection we show how to efficiently compute deterministic automata for product specifications, assuming that all class names referenced in the specification are concrete.

Our construction consists of two stages: first we show how to create a deterministic version for an automaton in $\text{FA}(G)$, and then we show how to combine such automata into a single deterministic automaton which accepts the paths generated by a product specification.

We start with the observation that an automaton in $\text{FA}(G)$ has only the following two potential sources for non-determinism:

- (1) “dead-ends,” i.e., for some states $q \in Q$ and symbols $a \in \Sigma$ we may have that $\delta(q, a) = \emptyset$, or
- (2) \diamond -transitions, i.e., if $f(q) = (C, \text{out})$ for some abstract C , then possibly $|\delta(q, \diamond)| > 1$.

This special structure allows us to get deterministic versions of automata in $\text{FA}(G)$ in polynomial time. Given an automaton $M \in \text{FA}(G)$, we show how to obtain a deterministic automaton $\text{Determinize}(M)$ which accepts the same set. Our description is given in steps: dead-ends are easy to deal with, but it is more convenient to defer their treatment to a later point. We start by addressing \diamond -transitions.

The idea is to contract path segments whose labels are abstract class names and \diamond 's. We use the following definitions extensively.

Definition 4.1

- (1) A symbol $a \in \Sigma$ is **transparent** if either $a = \diamond$ or $a = A$ for an abstract class $A \in V$.
- (2) A computation $\sigma = q^0 a^1 q_1 \dots a^n q^n$ is **a^n -shrinkable** from q^1 to q^n if
 - $a^1, a^2 \dots a^{n-1}$ are transparent and a^n is not transparent, and
 - $\delta(q^n, a) = \emptyset$ for all transparent symbols $a \in \Sigma$.

Let $M \in \text{FA}(G)$ with $M = (Q, \Sigma, \delta, q_0, F)$ such that $F = \{(C, \text{out})\}$ for a concrete class name C . We define $\text{Determinize}(M)$ to be the finite automaton over Σ with state set Q , start state q_0 , and final states in F , where all its transitions are ‘contractions’ of shrinkable computations of M . Formally, the transition function of $\text{Determinize}(M)$ is δ^* , defined for all states $q \in Q$ and non-transparent symbols a by

$$\delta^*(q, a) = \{q' \mid \text{there exists an } a\text{-shrinkable computation from } q \text{ to } q'\} .$$

An example of the construction is given in Figure 4.

We describe the properties of $\text{Determinize}(M)$ in the following lemma.

Lemma 4.3 *Let M be an automaton in $\text{FA}(G)$. Then:*

- (1) $\text{Determinize}(M)$ can be computed in polynomial time.

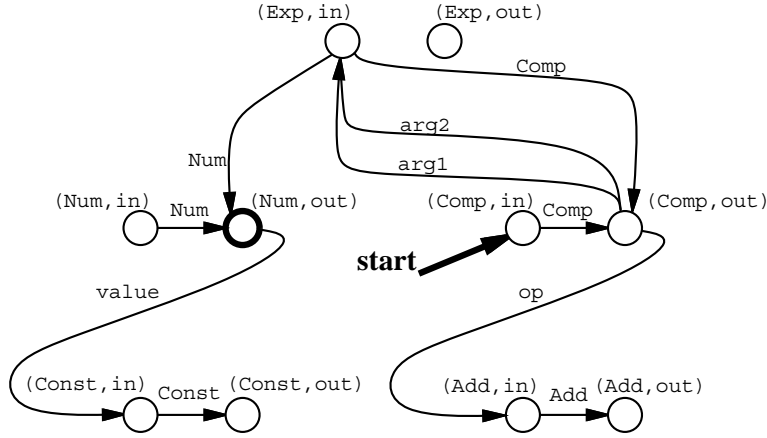


Figure 4: The automaton $\text{Determinize}(\text{Auto}(\text{Comp}, \text{Num}))$ corresponding to the class graph of Figure 2 and the specification $[\text{Comp}, \text{Num}]$.

- (2) The transition function δ^* of $\text{Determinize}(M)$ satisfies $|\delta^*(q, a)| \leq 1$ for all q, a .
- (3) $L(\text{Determinize}(M)) = \text{Simplify}(L(M))$.

Proof. (1) is obvious. We now prove (2). Suppose, for the sake of contradiction, that $|\delta^*(q, a)| > 1$. If $q = (C, \text{out})$ for some class C , then it must be the case that C is concrete, since there is no shrinkable computation of M which ends at (C, out) for an abstract C . Moreover, since all edges outgoing from C in G are reference edges, we have that all non-shrinkable computations from q are of the form $\langle (C, \text{out}), a, (C, \text{in}) \rangle$, where all the a 's are distinct labels from N , and hence $|\delta^*(q, a)| \leq 1$ and this case is ruled out.

If $q = (C, \text{in})$ for some class C , we need to consider two cases. If C is concrete, then the only shrinkable computation from q is $\langle (C, \text{in}), C, (C, \text{out}) \rangle$ and this case is ruled out too. If C is abstract, we argue that if σ is an a -shrinkable computation from q , then $a = C'$ for some non-abstract class C' : this follows from fact that a -shrinkable computations with $a \in N$ must start with a state (C'', out) where C'' is a concrete class name. Now, any C' -shrinkable computation where C' is a concrete class name must be to the state (C', out) . It follows that if $q = (C, \text{in})$ for an abstract class C , we have that $\delta^*(q, a) \subseteq \{(C', \text{out})\}$ if $a = C'$, and $\delta^*(q, a) = \emptyset$ otherwise. In any case, we have a contradiction

To see that (3) holds, consider a path $p \in L(M)$, and let σ_p be an accepting computation of M on input p . Consider the sequence σ'_p obtained from σ_p by omitting all subsequences of the form $q^{i+1}a^{i+1} \dots a^{i+n}$, where the symbols $a^{i+1} \dots a^{i+n}$ are transparent, and the symbols a^i and a^{i+n+1} are non-transparent. Since $F = \{(C, \text{out})\}$ for a concrete class name C , we have that the last symbol in $\text{String}(\sigma_p)$ is non-transparent, and hence $\text{String}(\sigma'_p) = \text{SimplifyPath}(p)$, and that σ'_p is a computation of $\text{Determinize}(M)$. The other containment is proved by essentially the same argument. ■

To complete the determinization of automata in $\text{FA}(G)$, introduce a fresh state **dead**, and define $\delta^*(q, a) = \text{dead}$ for all states $q \in Q \cup \{\text{dead}\}$ and symbols $a \in \Sigma$ such that $\delta^*(q, a) = \emptyset$.

Combining Lemmas 4.2 and 4.3, we obtain the following result.

Theorem 4.4 *Let $[A, B]$ be a traversal specification and let G be a class graph. Then there exists a deterministic finite automaton $M_{[A, B]}$, constructible in time polynomial in $|G|$, such that $L(M_{[A, B]}) = \text{Simplify}(\text{PathSet}_G([A, B]))$.*

We now describe the second stage of the construction: given deterministic automata accepting $\text{PathSet}(D_1)$ and $\text{PathSet}(D_2)$, combine them into a deterministic automaton accepting $\text{PathSet}(D_1 \cdot D_2)$. Our construction relies on the invariant that in all the automata we construct, there is a single final state (C, out) , and a single state of the form (C, in) such that $\delta((C, \text{in}), C) = (C, \text{out})$. It is easy to see that this property holds for any automaton $M = \text{Determinize}(\text{Auto}(A, B))$ where A and B are class names. We will also show that the invariant is maintained throughout the construction.

The idea is that given two automata M and N , such that the final state of M is $(C, \text{out})^M$ and the initial state on N is $(C, \text{in})^N$ for the same class name C , we “identify” the states $(C, \text{in})^M$ with $(C, \text{in})^N$ and $(C, \text{out})^M$ with $(C, \text{out})^N$. We show that due to a special property of traversal specifications, it suffices to retain only the transitions from $(C, \text{out})^N$.

Formally, let $M = (Q^M, \Sigma, \delta^M, q_0^M, F^M)$ and $N = (Q^N, \Sigma, \delta^N, q_0^N, F^N)$ be two automata, such that $F^M = \{(J, \text{out})^M\}$ for some class name J , and $q_0^N = (J, \text{in})^N$ for the same J . If $\delta^N(q_0^M, J) \in F^N$ (i.e., N accepts the string ‘ J ’, define $M \cdot N = M$. Otherwise, let $(J, \text{in})^M \in Q^M$ be the unique state in M of the form (J, in) such that $\delta^M((J, \text{in})^M, C) = (C, \text{out})^M \in F^M$. The existence of such a state is guaranteed by the invariant. Assume, without loss of generality, that the states are named so that $Q^M \cap Q^N = \emptyset$. The automaton $M \cdot N = (Q, \Sigma, \delta, q_0, F)$ is defined as follows (see Figure 5).

- $Q = Q^M \cup Q^N \setminus \{(J, \text{out})^M, (J, \text{in})^N\}$
- $q_0 = q_0^M$
- $F = F^N$
-

$$\begin{aligned}
\delta(q, a) &= \delta^M(q, a) && \text{for all } a \in \Sigma \text{ and } q \in Q^M, q \neq (J, \text{out})^M, \delta^M(q, a) \neq (J, \text{out})^M \\
\delta(q, a) &= \delta^N(q, a) && \text{for all } a \in \Sigma \text{ and } q \in Q^N, q \neq (J, \text{in})^N, \delta^N(q, a) \neq (J, \text{in})^N \\
\delta(q, J) &= \delta^N(q_0, J)^N && \text{for all } q \in Q^M \text{ such that } \delta^M(q, J) = (J, \text{out})^M \\
\delta(q, a) &= (J, \text{in})^M && \text{for all } q \in Q^N \text{ such that } \delta^N(q, a) = (J, \text{in})^N
\end{aligned}$$

We wish to prove that $L(M \cdot N) = L(M) \cdot L(N)$. We start with a general property of traversal specifications. For a node $A \in V$, define *cycle from A to A* to be a path whose source and target are both A . The following lemma holds true for *any* specification (not necessarily product specifications).

Lemma 4.5 *Suppose that $p \in \text{PathSet}(D)$ for a traversal specification D and that $p = sAr$ for some paths s, r and a class name $A \in V$. Then for any cycle c from A to A , $s c r \in \text{PathSet}(D)$.*

Proof. By induction on the structure of D . ■

We now prove the main property of automata for product specifications.

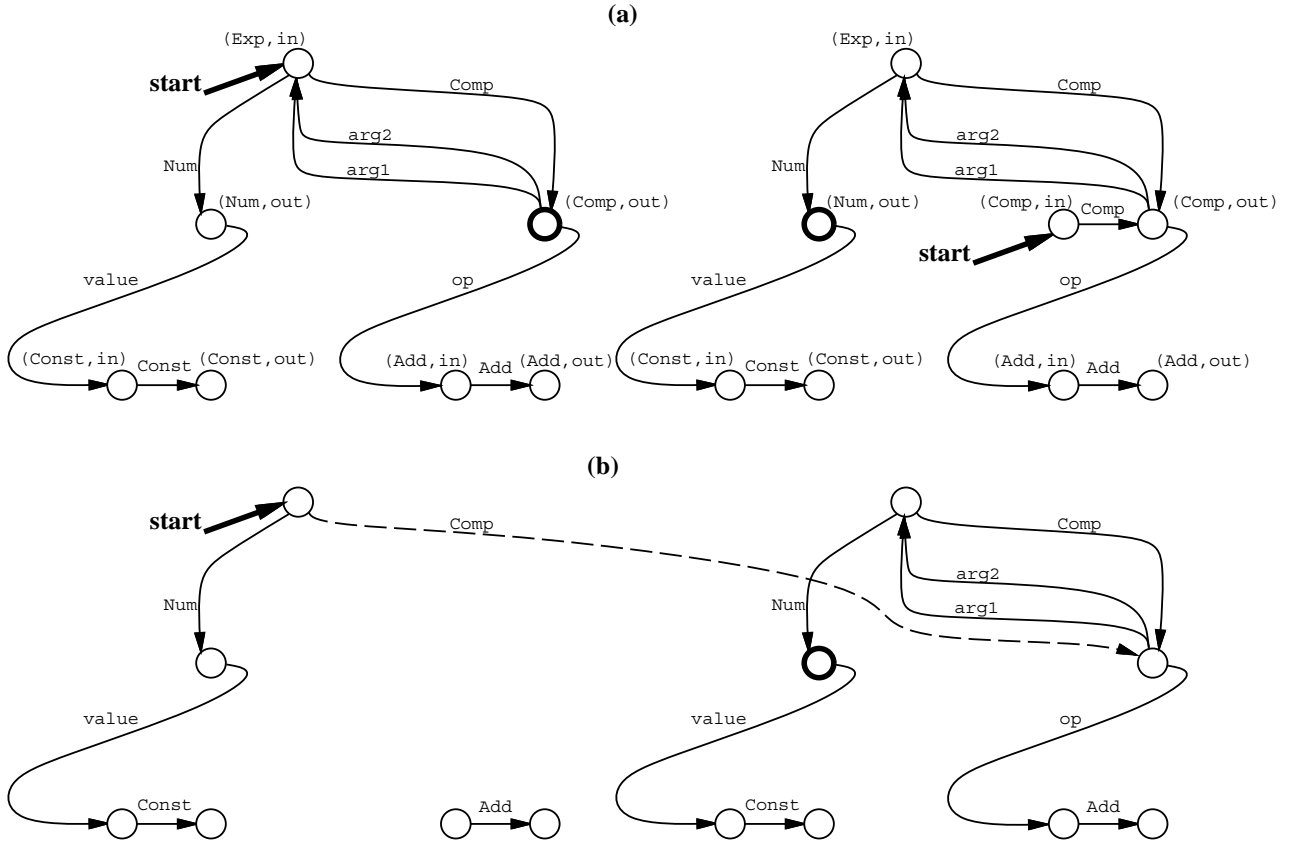


Figure 5: Construction of the automaton $\text{Determinize}(\text{Auto}(\text{Exp}, \text{Comp})) \cdot \text{Determinize}(\text{Auto}(\text{Comp}, \text{Num}))$. Top: the automata $\text{Determinize}(\text{Auto}(\text{Exp}, \text{Comp}))$ and $\cdot\text{Determinize}(\text{Auto}(\text{Comp}, \text{Num}))$ with unreachable states removed. Bottom: the product automaton, with the new transitions given in dashed arrows.

Lemma 4.6 *Let M and N automata such that $M \cdot N$ is defined, and assume that $L(M) = \text{PathSet}(D_1)$ and $L(N) = \text{PathSet}(D_2)$ for some traversal specifications D_1, D_2 . Then, for $M \cdot N$ defined as above we have that*

1. $M \cdot N$ is a deterministic finite automaton constructible in time polynomial in $|M|$ and $|N|$.
2. $L(M \cdot N) = \text{PathSet}(D_1 \cdot D_2)$.
3. The final states of $M \cdot N$ are $\{(C, \text{out})\}$ and there exists a single state (C, in) in $M \cdot N$ such that $\delta((C, \text{in}), C) = (C, \text{out})$.

Proof. Claim (1) follows immediately from the construction. To see that (3) holds, observe that the only transitions of N affected by the construction are transitions into q_0^N . However, since $\delta^N(q_0^N, J) \notin F$, it follows that the transitions into the single accepting state of N are not changed in the product automaton, and since $F = F^N$ by construction, the invariant property is inherited from N . In the remainder of this proof we show that (2) is true, using the notations of the construction above.

First we prove that $\text{PathSet}(D_1 \cdot D_2) \subseteq L(M \cdot N)$. Consider a path $p \in \text{PathSet}(D_1 \cdot D_2)$, and suppose that $p = s \cdot r$, where s is the shortest prefix of p such that $s \in \text{PathSet}(D_1)$. Using the definition of path concatenation, let s end with J and let r start with J . Note that by assumption, $s \in L(M)$. Consider the computation σ^M of M on s : the last state in σ^M is $(J, \text{out})^M$, since it is the only accepting state in M ; moreover, by our choice of s , $(J, \text{out})^M$ is visited only once in σ^M .

Consider the string r . Since $s \cdot r \in \text{PathSet}(D_1 \cdot D_2)$, there must be a decomposition $p = p_1 \cdot p_2$ such that $p_1 \in \text{PathSet}(D_1)$ and $p_2 \in \text{PathSet}(D_2)$. Since s is the shortest prefix of p in $\text{PathSet}(D_1)$, it must be the case that p_2 is a suffix of r ; since both r and p_2 start with the same symbol J , we have that r consists of a cycle from J to J followed by a path in $\text{PathSet}(D_2)$. Hence, by Lemma 4.5, we have that $r \in \text{PathSet}(D_2)$.

Consider now a computation σ^N of N on the input string r . Since $r \in \text{PathSet}(D_2)$, we have that $r \in L(N)$. Consider the computation obtained from concatenating σ^M and σ^N , after deleting the last symbol in σ^M (which is $(J, \text{out})^M$) and the first two symbols in σ^N (which are $(J, \text{in})^N$ and J). The resulting sequence is an accepting computation of $M \cdot N$ for $p = sr$, and hence $p \in L(M \cdot N)$.

To prove that $\text{PathSet}(D_1 \cdot D_2) \supseteq L(M \cdot N)$, consider a string $p \in L(M \cdot N)$, and let σ_p be its corresponding accepting computation. First, observe that σ_p contains $\langle \dots, (J, \text{in})^M, J, (J, \text{out})^N, \dots \rangle$, since the start state of $M_1 \cdot M_2$ is in Q^M , the only accepting state of $M \cdot N$ is in Q^N , and the only transition from Q^M to Q^N is $\delta((J, \text{in})^M, J) = (J, \text{out})^N$. Let $\sigma_p = \sigma_s J \sigma_r$, where σ_s is the prefix of σ_p up to the *last* occurrence of $(J, \text{in})^M$. Decompose $p = sr$, where $s = \text{String}(\sigma_s)$.

Since σ_r consists solely of states in Q^N , it is obvious that $\langle (J, \text{in})^N, J, \sigma_r \rangle$ is an accepting computation of N , and hence $r \in L(N)$, which means that $r \in \text{PathSet}(D_2)$. The situation with σ_s is a little more complicated, since it may contain states from both Q^M and Q^N . To deal with that, we decompose further $\sigma_s J = \sigma_1 J \sigma_2$, where σ_1 is the maximal prefix of σ_s which consists solely of states of Q^M . By our choice, both σ_1 and σ_2 must end with $(J, \text{in})^M$, σ_1 starts with q_0^M , and σ_2 starts with $(J, \text{out})^N$. Let $s_1 = \text{String}(\sigma_1)$ and $s_2 = \text{String}(\sigma_2)$. By construction, the string $s_1 J$ corresponds to an accepting computation of $M \langle \sigma_1, J, (J, \text{out})^M \rangle$, and hence $s_1 J \in \text{PathSet}(D_1)$. Next, note that the string $J s_2 J$ starts with J , ends with J , and corresponds to some path in the class graph. Therefore, by Lemma 4.5, $s J = s_1 J s_2 J \in \text{PathSet}(D_1)$, and hence, $p = s_1 J \cdot s_2 J \cdot r \in \text{PathSet}(D_1 \cdot D_2)$. ■

We summarize the results of this subsection in the following theorem.

Theorem 4.7 *Let D be a product specification and let G be a class graph. Then there exists a deterministic finite-state automaton M , constructible in time polynomial in $|G|$ and $|D|$, such that $L(M) = \text{PathSet}(D)$.*

Remark: Our construction works in the case where

$$\text{Simplify}(\text{PathSet}(D_1) \cdot \text{PathSet}(D_2)) = L(\text{Determinize}(\text{Auto}(D_1)) \cdot \text{Determinize}(\text{Auto}(D_2)))$$

This holds if we assume that all class names used in D_1 and D_2 (in particular, their join point) are concrete. The construction given in Section 4.3 below does not require this assumption.

4.3 General Specifications

We now turn to general specifications. It turns out that the construction for this case is conceptually much simpler than the specialized one given in Section 4.2, as it relies on standard automata-theoretic techniques. However, the running time of the algorithm may be exponential, since it contains the subset-construction determinization as one of the steps.

We start by defining three operations on nondeterministic automata.

- If M_1, M_2 are automata, then $M_1 \oplus M_2$ is the automaton such that $L(M_1 \oplus M_2) = L(M_1) \cup L(M_2)$. $M_1 \oplus M_2$ can be computed by introducing a fresh start state with ϵ transitions to the start states of M_1 and M_2 [3].
- If M_1, M_2 are automata, then $M_1 \odot M_2$ is the automaton defined as follows. The states of $M_1 \odot M_2$ are the disjoint union of the states of M_1 and the states of M_2 , together with a fresh state m . The start state of $M_1 \odot M_2$ is the start state of M_1 . The final states of $M_1 \odot M_2$ are the final states of M_2 . The transitions of $M_1 \odot M_2$ are the union of the transitions of M_1 and the transitions of M_2 , together with ϵ -transitions from each final state of M_1 to m , and ϵ -transitions from m to each state in M_2 which can be reached from the start state of M_2 by a sequence of ϵ -transitions followed by one non- ϵ -transition. For an example of this construction, see Figure 6.
- If M is an automaton which accepts only paths in some class graph, then $\text{Solidify}(M)$ is the automaton defined as follows. The states of $\text{Solidify}(M)$ are those of M together with a fresh state s . The start state of $\text{Solidify}(M)$ is that of M . The only final state of $\text{Solidify}(M)$ is s . The transitions of $\text{Solidify}(M)$ are defined as follows.

$$\begin{aligned}
 u &\xrightarrow{\epsilon} v \text{ if } u \xrightarrow{\epsilon} v \text{ is a transition of } M \\
 u &\xrightarrow{a} v \text{ if } u \xrightarrow{a} v \text{ is a transition of } M \text{ where } a \in N \cup \{u \mid \neg \text{Abstract}(u)\} \\
 u &\xrightarrow{\epsilon} v \text{ if } u \xrightarrow{a} v \text{ is a transition of } M \text{ where } a \in \{\diamond\} \cup \{u \mid \text{Abstract}(u)\} \\
 u &\xrightarrow{l} s \text{ if there is a path in } M \text{ from } u \text{ to a final state of } M \text{ which consist} \\
 &\quad \text{of one } l\text{-transition followed by a sequence of } \epsilon\text{-transitions.}
 \end{aligned}$$

For an example of this construction, see Figure 7.

The important properties of the constructions are given in the following lemma.

Lemma 4.8 *Given automata M_1 and M_2 accepting path sets in a class graph G ,*

- (1) $L(M_1 \oplus M_2) = L(M_1) \cup L(M_2)$,
- (2) *If $L(M_1) \cdot L(M_2)$ is defined, then $L(M_1 \odot M_2) = L(M_1) \cdot L(M_2)$, and*
- (3) $L(\text{Solidify}(M_1)) = \text{Simplify}(L(M_1))$.

Proof. For (1), see [3]. To see (2), first consider a string x accepted by $M_1 \odot M_2$. Note that the start state is a state of M_1 , all the final states are states of M_2 , and the only transitions between states of M_1 and states of M_2 are the ϵ -transitions through a final state of M_1 and the fresh state m . It follows that an accepting computation σ of $M_1 \odot M_2$ on x may be decomposed $\sigma = \sigma_1 \epsilon m \epsilon \sigma_2$,

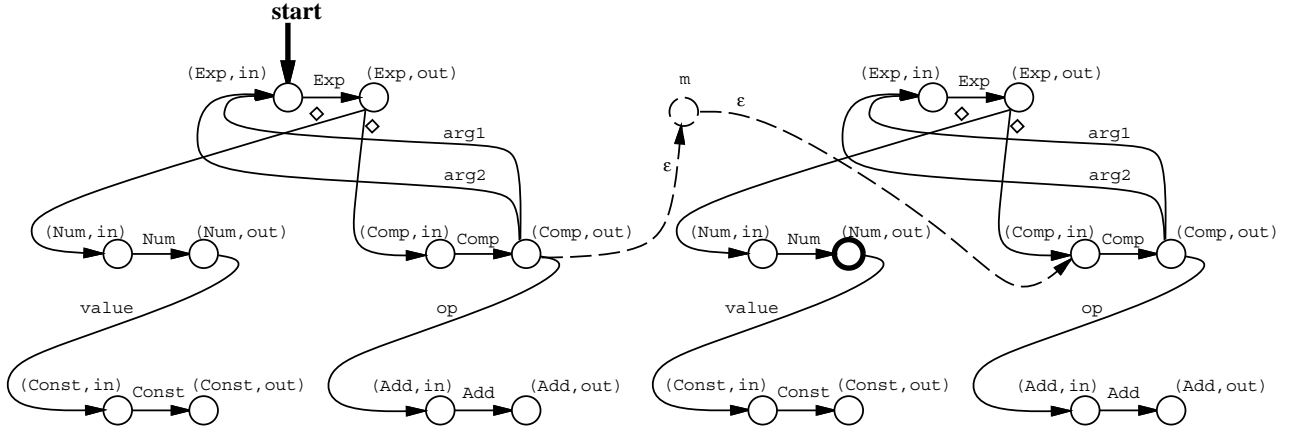


Figure 6: The automaton $\text{Auto}(\text{Exp}, \text{Comp}) \odot \text{Auto}(\text{Comp}, \text{Num})$. The new transitions and the new state m are indicated by dashed lines.

where σ_1 is an accepting computation of M_1 . Note that σ_2 starts with the start state of M_2 , followed by an ϵ -transition to some state of M_2 . Denote the last symbol in σ_1 by J . By assumption that $L(M_1) \cdot L(M_2)$ is defined, we have that the first symbol in any string accepted by M_2 must be the last symbol in any string accepted by M_1 . Hence, it follows from the construction that by replacing the first ϵ in σ_2 with J , the result is an accepting computation of M_2 , and we may conclude that $L(M_1 \odot M_2) \subseteq L(M_1) \cdot L(M_2)$. The reversed containment is proven similarly.

The proof of (3) is similar to the proof of Part (3) of Lemma 4.3 and is therefore omitted. \blacksquare

Finally, for a traversal specification D and a class graph G , define $A_G(D)$ recursively as follows.

$$\begin{aligned} A_G([A, B]) &= \text{Auto}_G(A, B) \\ A_G(D_1 \cdot D_2) &= A_G(D_1) \odot A_G(D_2) \\ A_G(D_1 + D_2) &= A_G(D_1) \oplus A_G(D_2) \end{aligned}$$

Clearly, $A_G(D)$ accepts precisely $\text{PathSet}_G(D)$, and it can be computed in $O(|D| |G|)$ steps. Hence, we can compute an automaton which accepts $\text{Simplify}(\text{PathSet}_G(D))$ in polynomial time. However, the resulting automaton is non-deterministic, and thus cannot be used directly to guide traversals. The next step in our construction is therefore to determinize the automaton accepting $\text{Simplify}(\text{PathSet}_G(D))$ using the standard subset construction. This crucial step may take exponential time.

Example. In Figure 3 we give an illustration of $\text{Auto}(\text{Comp}, \text{Num})$. The automaton $\text{Auto}(\text{Exp}, \text{Comp})$ differs in having (Exp, in) as start state and $(\text{Comp}, \text{out})$ as final state. In Figure 6 we show the automaton $\text{Auto}(\text{Exp}, \text{Comp}) \odot \text{Auto}(\text{Comp}, \text{Num})$. In Figure 7 we show the automaton $\text{Solidify}(\text{Auto}(\text{Exp}, \text{Comp}) \odot \text{Auto}(\text{Comp}, \text{Num}))$.

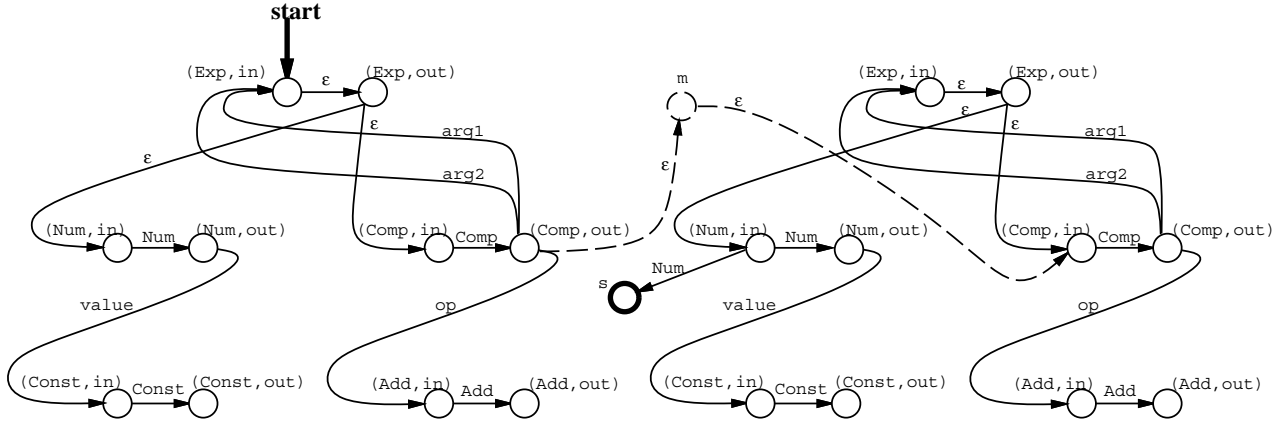


Figure 7: The automaton $\text{Solidify}(\text{Auto}(\text{Exp}, \text{Comp}) \odot \text{Auto}(\text{Comp}, \text{Num}))$.

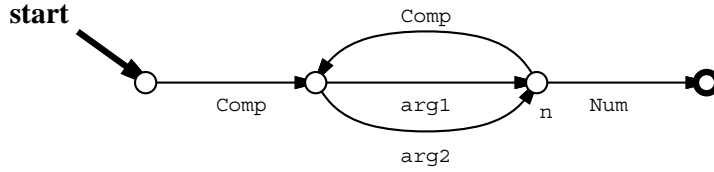


Figure 8: The minimal automaton accepting $\text{Simplify}(\text{Exp}(\diamond \text{Comp}(\text{arg1} + \text{arg2})\text{Exp})^+ \diamond \text{Num})$.

4.4 The Compiling Algorithm

In this subsection we explain how, given a deterministic finite automaton accepting simplified paths in a class graph, to generate code that will traverse objects accordingly.

Before we start generating code, we apply the standard state minimization algorithm to the given deterministic automaton. This step takes time which is polynomial in the size of the automaton, and its effect is to guarantee that the code we generate has optimal size. The final automaton of our running example is given in Figure 8.

We are now in a position to explain how to generate code. We use the following notation. Given an automaton M and a state s , let $\text{Outgoing}_M(s)$ denote the set of all transitions $s \xrightarrow{a} s'$. Let S_M be the set of states of M with an outgoing transition $s \xrightarrow{C} s'$ for some class name C .

For an automaton M , we define P_M to be a program in the target language by the following rule. The method names in P_M are the elements of S_M , defined as follows.

1. If $s \xrightarrow{C} s'$, where C is a class name, and $\text{Outgoing}_M(s') = \{s' \xrightarrow{l_i} m_i \mid i \in 1..n\}$, then $P_M(C, s) = \langle l_1.m_1 \dots l_n.m_n \rangle$, where $l_j < l_k$ for $1 \leq j < k \leq n$.
2. Otherwise, if C is a class name used in M and $s \in S_M$, then $P_M(C, s) = \langle \rangle$.

If we in case (2) have access to the class graph from which the automaton M was generated, then we can avoid the generation of many unreachable methods. In a target language with inheritance,

the empty methods can be placed in superclasses, thus reducing code size further. The wrapper code would by an implementation be inserted into the methods generated from case (1).

Example. Given the deterministic automaton shown above, the compiling algorithm emits the following code (written in a programming language-like notation).

<pre> CLASS Comp VAR arg1, arg2: Exp METHOD m case (1) arg1.n; arg2.n END METHOD n case (1) arg1.n; arg2.n END END </pre>	<pre> CLASS Num METHOD m case (2) — No code here END METHOD n case (1) — Wrapper code here END END </pre>
---	---

Notice that two method names are needed to distinguish if a Num object is reached via a Comp object or not. In the former case (method n), the wrapper code should be executed, in the latter case (method m), it should not.

The example indicates the consequence of the potentially large size of the deterministic automaton which accepts $\text{Simplify}(\text{PathSet}_G(D))$: massive wrapper code duplication, in the worst case. Notice that if we change the class graph of the example such that class Num can be reached from several classes, say Comp_1 , Comp_2 , etc, then each class Comp_i gets two methods m and n .

We conclude this paper with a proof that the compiling algorithm is correct.

Theorem 4.9 *If M_s is a deterministic automaton which accepts a word set, then*

$$\Omega \vdash_n o : L(M_s) \triangleright H \quad \text{if and only if} \quad \Omega \vdash_c o : s : P_M \triangleright H .$$

Proof. Suppose first that $\Omega \vdash_n o : L(M_s) \triangleright H$ is derivable. We proceed by induction on the structure of the derivation of $\Omega \vdash_n o : L(M_s) \triangleright H$. Since $\Omega \vdash_n o : L(M_s) \triangleright H$ is derivable, we have that

$$\begin{aligned}
H &= o \cdot H_1 \cdot \dots \cdot H_n \\
\text{First}(\text{Chop}(L(M_s), \text{Class}(o))) &= \{l_i \mid i \in 1..n\} \\
o &\xrightarrow{l_i} o_i \text{ is in } \Omega, i \in 1..n, \\
l_j &< l_k \text{ for } 1 \leq j < k \leq n, \text{ and that} \\
\Omega \vdash_n o_i : \text{Chop}(\text{Chop}(L(M_s), \text{Class}(o)), l_i) &\triangleright H_i \text{ is derivable for all } i \in 1..n.
\end{aligned}$$

There are two cases. If $\text{Chop}(L(M_s), \text{Class}(o)) = \emptyset$, then $\text{First}(\text{Chop}(L(M_s), \text{Class}(o))) = \emptyset$, so $n = 0$, and $H = o$. Moreover, there is no u such that $s \xrightarrow{\text{Class}(o)} u$ is in M , so $P_M(\text{Class}(o), s) = \langle \rangle$, and hence $\Omega \vdash_c o : s : P_M \triangleright o$ is derivable, which is the desired conclusion.

If $\text{Chop}(L(M_s), \text{Class}(o)) \neq \emptyset$, then $s \xrightarrow{\text{Class}(o)} u$ is in M_s for some u , and

$$\text{First}(\text{Chop}(L(M_s), \text{Class}(o))) = \{l_i \mid u \xrightarrow{l_i} s_i \text{ is in } \text{Outgoing}_{M_s}(u)\} .$$

Thus, the side condition of the rule for \vdash_c is satisfied. By the induction hypothesis, $\Omega \vdash_c o_i : s_i : P_M \triangleright H_i$ is derivable for all $i \in 1..n$. We conclude that $\Omega \vdash_c o : s : P_M \triangleright H$ is derivable.

The converse is proved similarly. ■

By combining Theorem 3.3 and Theorem 4.9, we obtain our compiler correctness result.

Corollary 4.10 *For a class graph G , a traversal specification D , a deterministic automaton M_s which accepts $\text{Simplify}(\text{PathSet}_G(D))$, an object graph Ω , a node o in Ω , and a traversal history H , we have*

$$\Omega \vdash_s o : \text{PathSet}_G(D) \triangleright H \quad \text{if and only if} \quad \Omega \vdash_c o : s : P_M \triangleright H .$$

In summary, compilation of an adaptive program proceeds by first computing an automaton M which accepts $\text{Simplify}(\text{PathSet}_G(D))$ and then generating the program P_M .

5 Conclusion

We have presented two new compiling algorithm for a core language of adaptive programs, based on automata constructions. Both algorithms generate efficient code for programs which could not be compiled by previous algorithms [10]. One algorithm works in the important special case of product specifications, and runs in polynomial time. The second algorithm works for any specification, but may require exponential time to compute. In future work, we will attempt to find polynomial time algorithms that generate efficient code and work for all instances.

Acknowledgments. We thank Linda Seiter and the anonymous referees for many insightful comments on a draft of the paper. This work has been partially supported by the National Science Foundation under grant numbers CDA-9015692 (Research Instrumentation), and CCR-9402486 (Software Engineering). The first author was supported by BRICS (Basic Research in Computer Science, Centre of the Danish National Research Foundation).

References

- [1] Version 5.5 of the Demeter Tools/C++, which generates C++ code, is available through the Demeter home page: <http://www.ccs.neu.edu/research/demeter/>.
- [2] Simon Gibbs, Dennis Tsichritzis, Eduardo Casais, Oscar Nierstrasz, and Xavier Pintado. Class management for software communities. *Communications of the ACM*, 33(9):90–103, September 1990.
- [3] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Publishing Company, 1979.
- [4] Walter L. Hürsch and Linda M. Seiter. Automating the evolution of object-oriented systems. In *International Symposium on Object Technologies for Advanced Software*, 1996. To appear.

- [5] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.
- [6] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.
- [7] Karl J. Lieberherr, Ignacio Silva-Lepe, and Cun Xiao. Adaptive object-oriented programming using graph-based customization. *Communications of the ACM*, 37(5):94-101, May 1994.
- [8] Karl J. Lieberherr and Cun Xiao. Object-oriented software evolution. *IEEE Transactions on Software Engineering*, 19(4):313-343, April 1993.
- [9] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.
- [10] 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.