

Object-Process Diagrams as an Explicit Algorithm-Specification Tool

ABSTRACT

Algorithms need clear and formal representations to be implemented as computer programs. The Object-Process Methodology (OPM) has been shown to successfully describe the structure and behavior of systems by combining objects and processes within an integrated, coherent set of object-process diagrams (OPDs). However, OPDs lack control-flow constructs for explicit specification of the entire process sequence, which is essential for algorithm implementation. In this article we augment the OPD notation to explicitly mark the necessary execution order among processes by introducing four basic control-flow mechanisms—sequence, branch, loop, and recursion—as well as other means, such as process ownership, to support current object-oriented design and programming concepts. The explicit representation of an algorithm also makes it possible to automatically generate the program code from the OPD set and reverse engineer existing complex code to an OPD set to enhance code understandability, maintenance, and reuse.

An algorithm is an abstraction of a particular solution strategy for a given problem. A computer solution to such a problem is an implementation that applies the strategy through a sequence of computer instructions comprising a computer program. The algorithm should be formalized into a clear representation in order to be correctly transferred from design to implementation.

To be implementable in different hardware and software environments, algorithms should be designed to be programming-language and data-structure independent. The designed algorithm

has traditionally been expressed using two optional methods: natural (“structured”) language or flowcharts. Natural language description consists of a labeled sequence of sentences. The labels are used to specify a special execution order other than the default consecutive order. A flowchart is a graphical counterpart of the textual specification, in which sentences are enclosed in boxes. The execution order between two boxes is denoted by a direct link from one box to another. Conditional branching, a special construct expressed by a diamond with the condition text inside and outlets at two corners labeled “Yes” and “No,” is introduced to indicate the execution order under certain conditions.

While these two description modes are programming-language and data-structure independent, both focus on processes and flow of control, leaving objects and the data that represent them implicit. The lack of explicit representation of objects and data makes the conversion of flowchart algorithm specification into code an incompletely specified task.

By inserting new nodes for data between process nodes, the data-flow diagram (DFD) provides more detailed data flows than flowcharts. In DFDs, data are inputs/outputs of processes, and processes can be considered to be *data transformations*.¹ The extended DFD proposed in Ward’s article¹ is an improvement over flowcharts, but it lacks expression of structural relations among objects such as Aggregation-Participation, Generalization-Specialization, and Featuring Characterization.

PLAN DIAGRAMS

Inspired by the common, fundamental characteristic shared by various engineering types, Plan Calculus² is devised for a formal representation of algorithms and programs that are used in the different phases of software development. It describes the structure of an algorithm or program using a set of plan diagrams (PDs), each of which is composed of a set of constructs classified into parts, connections (links), and constraints. It uses input/output specifications to represent processes. A pair consisting of a test (branch) specification and a joint specification is introduced to represent the conditional branching control-flow mech-

Liu Wenjin is at Microsoft Research in Beijing, China, and in the Department of Computer Science and Technology at Tsinghua University, also in Beijing, and can be contacted at wylu@microsoft.com.

Dr. Dov Dori is head of the Area of Information Systems Engineering at the Faculty of Industrial Engineering and Management, Technion, Israel Institute of Technology in Haifa, Israel. He can be contacted at dori@ie.technion.ac.il.

anism. One connection type is a special control flow, represented by directed arcs with double crosshatch marks. Another connection type is data flow, represented by simple directed arcs connecting outputs to inputs of processes. The data are labeled at the inputs and outputs with names followed by a colon and (optionally) a type constraint.

In PDs, each data flow and control flow provides a partial order of abstraction of the program text. This order is more flexible than the fixed order induced by flowcharts and DFDs. Constraints in the PD include the preconditions and postconditions of processes and tests, and the invariants of data representations, which further restrict the implementation of the parts. PDs can abstract various kinds of mechanisms of control flows and data flows. Recursion expression is allowed in a PD by specifying the sub-PD as the type within the main PD. Iterations are expressed as tail recursions.³ PDs can provide convenient graphical descriptions of algorithms at different, appropriate abstraction levels, thereby enhancing the capability to understand, interpret, and program algorithms.

Despite their usefulness, PDs, like enhanced DFDs, also suffer from a lack of adequate reference to objects. The relationship between data and processes is restricted to input/output. With the advent of object concepts, the limited expressive power of this kind of representation approach has become more apparent, while the constant increase of the complexity of algorithms and systems poses ever more stringent requirements of precise and complete specification.

The object paradigm has been gaining wide acceptance as the favored approach to system analysis and design. The entity-relationship diagram (ERD) approach,⁴ on which the object paradigm relies, shifts the emphasis from processes to data, which are referred to as "objects" in the object-based and object-oriented concepts, but it is limited to expressing relationships among data. Object-oriented (OO) analysis and design methods, e.g., *Object-Oriented Analysis and Design with Applications*,⁵ attach processes to objects and represent structure, behavior, function, and other aspects of systems using a different model for each aspect. This multiple model approach poses a severe integration problem, which makes their use for systems development in general and for concise algorithm specification in particular a difficult task due to this inherent integration problem.

THE OBJECT-PROCESS METHODOLOGY

The Object-Process Methodology (OPM)⁶⁻⁸ is a system analysis and design approach that combines ideas from OOA and DFD within a single modeling framework to represent both the static/structural and dynamic/procedural aspects of a system in one coherent frame of reference. The use of a single model eliminates the integration problem and provides for clear understanding of the system being modeled. The object-process diagram (OPD) is OPM's graphical representation of objects and processes in the universe of interest along with the structural and procedural relationships that exist among them. Due to synergy, both the information content and expressive power

of OPDs are greater than those of DFD and OOA diagrams combined.

In OPM, both objects and processes are treated analogously as two complementary classes of *things*—elementary units that make up the universe. The relationships among objects are described using structural links such as Aggregation-Participation and Generalization-Specialization. The relationships between objects and processes are described by procedural links, which are classified into effect, agent, and instrument links.

An important feature of objects and processes in OPDs is their recursive and selective scaling ability, which provides for complexity management by controlling the visibility and level of detail of things in the system. In general, things are scaled up (zoomed in) as we proceed from analysis to design and to implementation. The scaling capability provides for function definitions and calls. Specifying Generalization-Specialization among processes enables the establishment of inheritance relations among processes in a manner similar to inheritance among objects.

While OPM has been applied to system analysis and design,⁹⁻¹¹ the expressive power of OPDs also makes them instrumental in specifying the finest details of algorithms that are designed with OO concepts. The selective recursive scaling further facilitates the detailed design and algorithmic representation.¹²⁻¹⁴ The resulting consistency of algorithm descriptions across the different phases of the software development process is highly desirable and makes it amenable to computer-aided software engineering.

However, the current terminology of OPD lacks symbols for some of the relations between objects and processes that arise in complex algorithmic processes when they are coded with current OO languages. For example, control flows are only implicitly expressed through the partial order induced by the procedural links, and the process ownership is not indicated.

This article augments the OPD symbol set for the purpose of explicit specification of algorithms designed with an object-oriented approach. In the next section we introduce some implementation-oriented symbols that are added to the OPD symbol set. We then show how control flow is represented in OPDs. Finally, we take the generic graphics-recognition algorithm^{11,14,15} as a case in point to demonstrate how OPM concisely and clearly represents complex algorithms by an OPD set.

OPD SYMBOLS FOR IMPLEMENTATION SPECIFICATION

The implementation-oriented OPD symbols in Figure 1 are marked with asterisks (*) and shown along with the analysis and design OPD symbols, which are "inherited" from the previous analysis and design phases and serve the implementation phase as well.

Things and relations in the OPD notation

A *thing* is the elementary unit that makes up the universe. An *object* is a persistent, unconditional thing. A *process* is a transient thing, whose existence depends on the existence of at least one


















Things	Structural Relations	Procedural Links
Object 	Aggregation-Participation 	Agent link 
State/Value 	Featuring Characterization 	Instrument link 
Process 	Generalization-Specialization (Inheritance) 	Effect link 
	Multiple Inheritance 	Consumption/Result link 
	*Virtual Inheritance 	*Process ownership indication 
	*Instantiation 	*Control link 
	Direct Structural Link 	
	Indirect Structural Link 	

Figure 1. Implementation-augmented OPD symbol set.

object. These terms were originally proposed for systems analysis in OPM.⁶ From the design and implementation viewpoint, an object can be regarded as a variable with a specified data type, while a process is a function or a procedure operating on variables, which are objects.

An *object class* is a template of all objects that have the same set of features and behavior patterns, and whose corresponding name in the OO terminology is simply *class*. Similar to Smalltalk, an OPM object class can also be thought of as an object. This concept renders the class a relative term rather than absolute. It is relative with respect to the objects that are instantiated from it and provides for *instantiation hierarchy*. A thing's *state* at a given point in time is the set (or vector) of attribute values the thing has at that time.

Certain structural relations between two objects, namely Aggregation-Participation, Featuring Characterization, and Generalization-Specialization, collectively referred to as the fundamental relations, are represented by a triangular symbol along the link that connects them. Aggregation-Participation describes the relationship of composition between two objects. Featuring Characterization's meaning follows its name: It is the relation between a feature—an attribute or an operation ("method," "service")—and the thing that the feature characterizes. A Generalization-Specialization link between two objects induces an inheritance relationship between two object classes. Virtual inheritance (which, as Figure 4 demonstrates, allows only one subobject of the inherited class within any object of the inheriting class through multiple inheritance routes) is represented by a dotted triangle as an implementation phase symbol.

Instantiation is also an implementation-oriented symbol that indicates an object is an instance of a class. Many structural relations are transitive.

The indirect structural link, represented by a dotted line instead of a solid line, denotes the fact that one or more things along the structure hierarchy are skipped. This is a useful notation because it is frequently the case that things at intermediate levels need not be specified in certain diagrams to avoid their overloading.

Agents and *instruments* are enablers of processes. They exist before the process execution, and their state (set of attribute values) is not changed by the process execution. An *effect link* links an *affected object* to the affecting process. An *affected* is an object whose state is changed by the process. A *consume* is an object

that is consumed (and destroyed) by the process, and it no longer exists after the process execution. It can be implemented in C++ by the "delete" statement. A *resultee* is a new object constructed as a result of the process execution, such as in the C++ "new" statement. The *consumption link* is graphically represented by a one-way arrow, directed from the consumed object to the consuming process. The *result link* is also represented by a one-

way arrow, but the arrow in this case is directed from the process to the resulting object. The *effect link* is represented by a two-way (bidirectional) arrow between the affected object and the process.

Implementation consideration for OPDs

In current OO languages, processes (referred to as methods) and services (or C++ member functions) belong to and are defined within some particular object class, so that the function can be called to handle an object (instance) of such class or the class itself, as in Graphics Class in Figure 11(b). In OPDs we name this object (when the process is called to handle it) or class (when the process is called to handle the class itself) *owner* of the process. We indicate the owner of a process by adding a small blank diamond symbol along the procedural link, next to its process end, as shown in Figure 3 between Object1 and Process.

When an analysis/design OPD is elaborated into an implementation OPD, any one of the procedural links—agent, instrument, or effect—indicates that the process at one end of the link belongs to the object at the other end of the link by adding the diamond symbol, shown in Figure 1, next to the solid circle, blank circle, or arrowhead, respectively. Among the procedural links attached to any process, one at most can be indicated as the owner. This preserves compatibility with the OO concept that a process is defined within exactly one class.

Objects (other than the owner and the resulting objects) that have procedural links to the process can be implemented as parameters of the process, which in C++ are *const* for enablers (agents

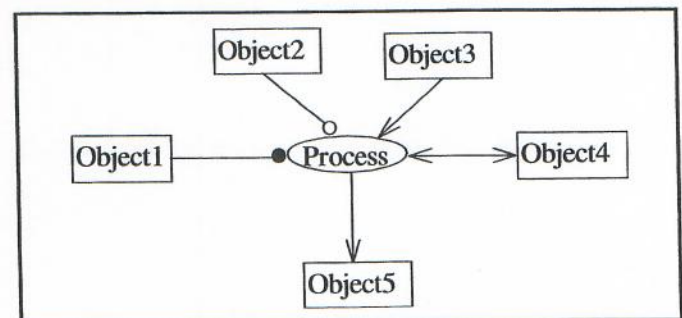


Figure 2. An analysis/design OPD showing all five types of procedural links, which from left clockwise around Process are the instrument link (solid circle), agent link, consumption link, effect link, and result link.

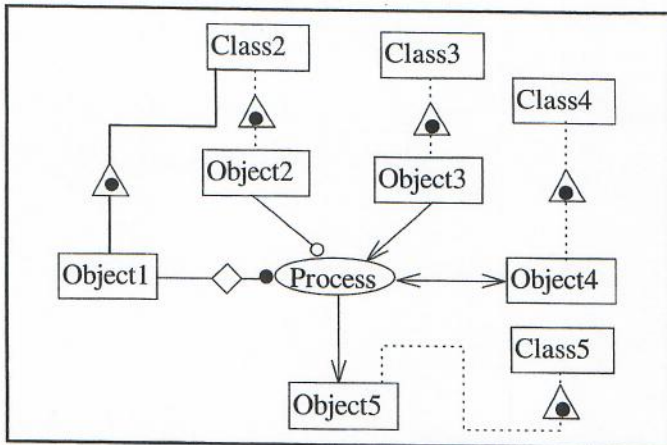


Figure 3. An alternative implementation OPD of Figure 2.

and/or instruments) and *volatile* for affectees (affected objects). Similarly, the process is a *const* process if the owner of the process is an enabler, and it is *volatile* if the owner is an affectee.

Figure 2 is an analysis OPD in which five objects (class instances) are linked to a process with five different types of procedural links. It may be implemented in a number of ways. We illustrate the extension to implementation OPDs in two of these ways, which are presented in Figure 3 and Figure 4, respectively. In Figure 3, each object is (either directly or indirectly) an instance of a class. One of the objects is indicated by the diamond symbol as the owner of the process. Because the object class that the process characterizes is not indicated, the process is considered by default to be a method of the class of which the owner is an instance—Class2 in our case. The corresponding C++ definitions and application code fragments are listed below.

```
class Class2
{
```

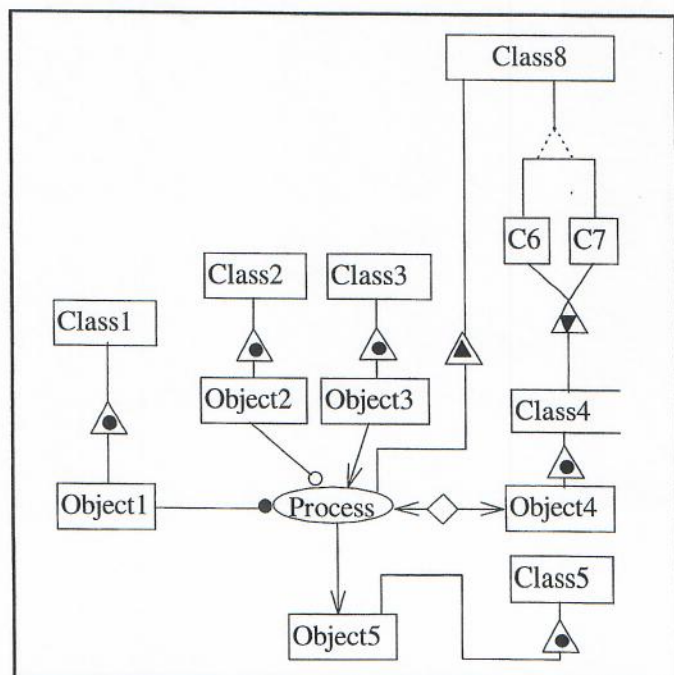


Figure 4. An implementation OPD of Figure 2.

```
Class5& Process(const Class2& Object2,
                Class3& Object3,
                Class4& Object4
                ) const;
} Object1;
```

```
Class5& Object5
= Object1.Process(Object2,
                  Object3,
                  Object4);
```

The process is a *const* process because its owner is an agent that is not changed by the process. Object2 is a *const* parameter of the process because it is an instrument of the process.

In Figure 4, the process is explicitly indicated as an operation of Class8 and should therefore be defined within Class8. Both C6 and C7 inherit from Class8. Class4 inherits from both C6 and C7, but it holds only one copy of C6 and C7 because, as indicated by the dotted triangle, C6 and C7 inherit from Class8 through virtual inheritance, which takes care of the problem of inheriting multiple copies of the same ancestor. The member function Process is no longer *const* because Object4, the owner, is an affectee, which is affected and changed by Process. The C++ code fragment that implements the OPD in Figure 4 is as follows:

```
class Class8
{
    Class5& Process(const Class1& Object1,
                  const Class2& Object2,
                  Class3& Object3
                  );
};
class C6 : virtual Class8 {
};
class C7 : virtual Class8 {
};
class Class4 : C6, C7.
{
    } Object4 ;

Class5& Object5
= Object4.Process(Object1,
                  Object2,
                  Object3) ;
```

In another way to implement the OPD in Figure 2, Object2 and Object3 can also be indicated as the owner of the process. The C++ code that implements them is similar to the code fragments shown previously. The process should be defined as a member function to the corresponding classes, and parameters and the process itself should also be marked *const* if necessary.

If an object class is indicated to be the owner, the process should also be defined as a member function, and it should be indicated as a static function. In this case, it is unnecessary to mark the process with the *const* keyword because a static member function cannot access data members of its class, as shown in Figure 5 and in the following code:

```
class Class2
{
```

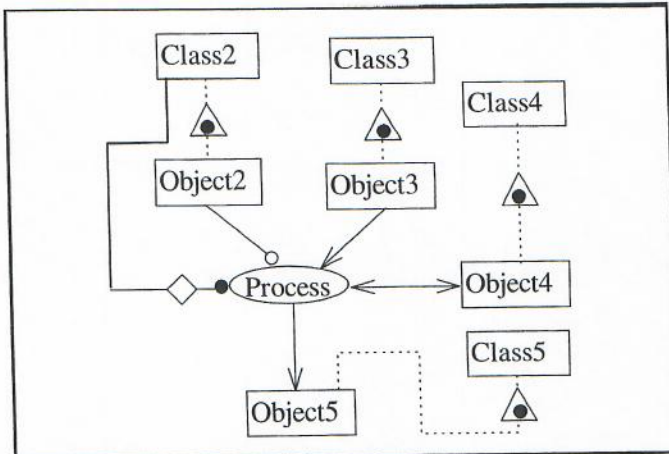



Figure 5. An implementation OPD.

```
static Class5& Process(const Class2& Object2,
    Class3& Object3,
    Class4& Object4
);

Class5& Object5
    = Class2 :: Process(Object2,
        Object3,
        Object4);
```

If an object class is used as a parameter of a process, such as the class *AGraphicsClass* in Figure 11(a) and Figure 11(b), the process should be defined as a template function, as in the two template functions shown in Listing 1.

In the analysis phase, a process may have more than one resulting object. However, C++ implementation allows at most one resulting object, which is the return value of the process. Hence, if there are two or more resulting objects in an analysis OPD, some of them should be implemented as affected objects, or all of them should be grouped into a single aggregate object. Another alternative is to reorganize the processes so that they obey the previous rule.

CONTROL-FLOW REPRESENTATION IN OPD Sequence and (conditional) branch structure

OPDs use the top-down time line⁷ and the data flow implied by the procedural links to define some of the control-flow sequencing. Cases in which the control does not flow from top down are marked by *control links*. A control link is graphically represented by a dashed arrow. It links a process or a state of an object to a process to explicitly indicate the flow of control. Control links describe sequential and "GOTO" control-flow mechanisms. They need not be used when the partial order of processes is clearly defined by the data-flow dependency. Thus, the control link is unnecessary in Figure 6(a) because B1 is an instrument to both P1

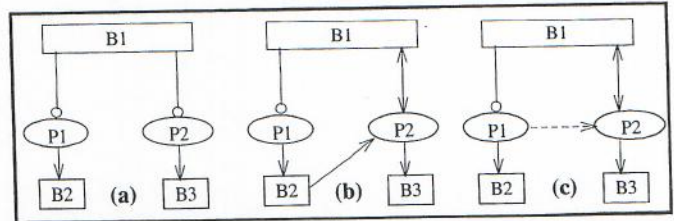


Figure 6. The use of control flows in OPDs. (a) unnecessary because of parallelism; (b) unnecessary because of the data-flow dependency; (c) ambiguous if no control link.

and P2. Hence, either process order yields the same results, as the two processes may be executed in parallel. In Figure 6(b), the control link is also unnecessary because the data-flow dependency requires that the process order be "P1 then P2." However, in Figure 6(c), the effects of the two possible execution orders may be quite different because B1 is affected by P2. The control link specifies the process order as "P1 then P2," which is in accord with the data-flow dependency and eliminates the ambiguity of process execution order.

Branching is represented by *control objects*, as in Figure 7. The number of possible branches is decided by the number of *states* (possible values) that the object may hold. For two possible values, the control object represents an if-then-else statement. If the number of possible values is more than two, it represents a switch statement. The conditional branching control flows converge at some point to end the branching. This converging point is P3 in Figure 7(a) and Pe in Figure 7(b). The corresponding C/C++ code fragments for the OPDs in Figure 7 are shown as follows:

Listing 1. Outline of the C++ implementation of the generic graphics-recognition algorithm.

```
template <class AGraphicsClass>
void GraphicDataBase :: detect(AGraphicsClass*)
{
    Primitive* APrimitive;
    while ((APrimitive = AGraphicsClass::firstComponent(this)) != NULL)
        constructFrom((AGraphicsClass*)0, APrimitive);
}

template <class AGraphicsClass>
AGraphicsClass* GraphicDataBase :: constructFrom(AGraphicsClass*,
    (const Primitive)* APrimitive)
{
    AGraphicsClass* AGraphics = new AGraphicsClass();
    if (AGraphics -> fillWith(APrimitive))
    {
        for (int direction=0; direction<=AGraphics->maxDirection(); direction++)
            while (AGraphics -> extend(this,direction));
        if (AGraphics -> isCredible()) {
            AGraphics -> addToDataBase(this);
            return AGraphics;
        }
    }
    delete AGraphics;
    return NULL;
}
```

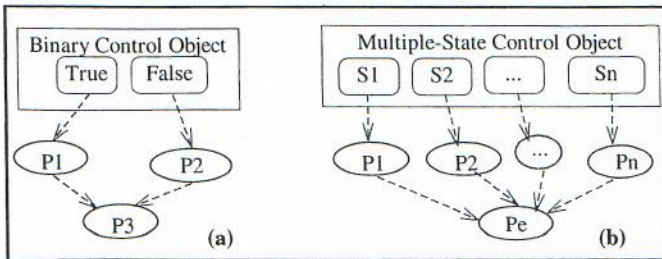



Figure 7. OPD representations of conditional branching mechanisms. (a) if-then-else; (b) switch.

```

if (Binary_Control_Object == True)
    P1();
else P2();
P3();

```

```

switch (Multiple_State_Control_Object)
{ case S1 : P1(); break;
  case S2 : P2(); break;
  .....
  case Sn : Pn(); break;
  default : break;
}
Pe();

```

The control link, like the control flow in Plan Calculus,² does not determine the exact process sequence. The process order can be chosen arbitrarily as long as it is compatible with the partial order specified by the data and control flow in the OPD.

Recursion and iteration

Unlike Plan Calculus,² OPDs allow loops of both data and control flow. In such a loop, a starting process should be explicitly specified by a control link in order to start the iteration. A binary (two-state) control object is involved. One state (referred to as the *exit state*) leads to an exit from the iteration, and another (referred to as the *loop state*) leads to the continuation of the iteration. The control object is governed by the results of a *testing process*. The continuation of the iteration should finally go back to the starting process. With the new definitions, OPD can distinguish between two patterns of iteration: *while-do* and *repeat-until*. The *while-do* pattern is characterized by a starting process (possibly the testing process) followed by its resulting control object. Repeat-until is characterized by a control object whose loop state leads the control link back to the starting process. For iteration, as a special form of *while-do* patterns, can be recognized by finding an index indicator involved in the iteration. See Figure 8(a) and Figure 8(b) for illustrations of OPD representations of iteration. Here are the corresponding C/C++ code fragments that implement the OPDs in Figure 8:

```

while (Testing_Process() == Loop)
    Iterating_Process();
Next_Process();

do
{

```

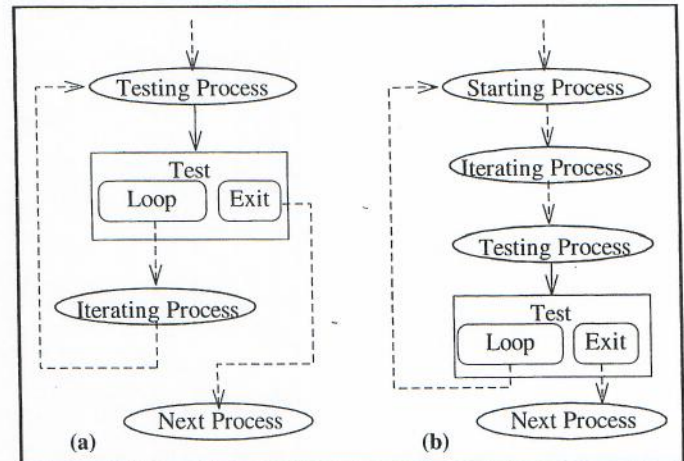


Figure 8. Illustrations of iteration and recursion. (a) while-do pattern; (b) repeat-until pattern.

```

Starting_Process();
Iterating_Process();
} while (Testing_Process() == Loop);

```

As an exception to the general branching mechanisms, branching that occurs inside an iteration may not have a joint end, because the conditional break and continue control mechanisms are

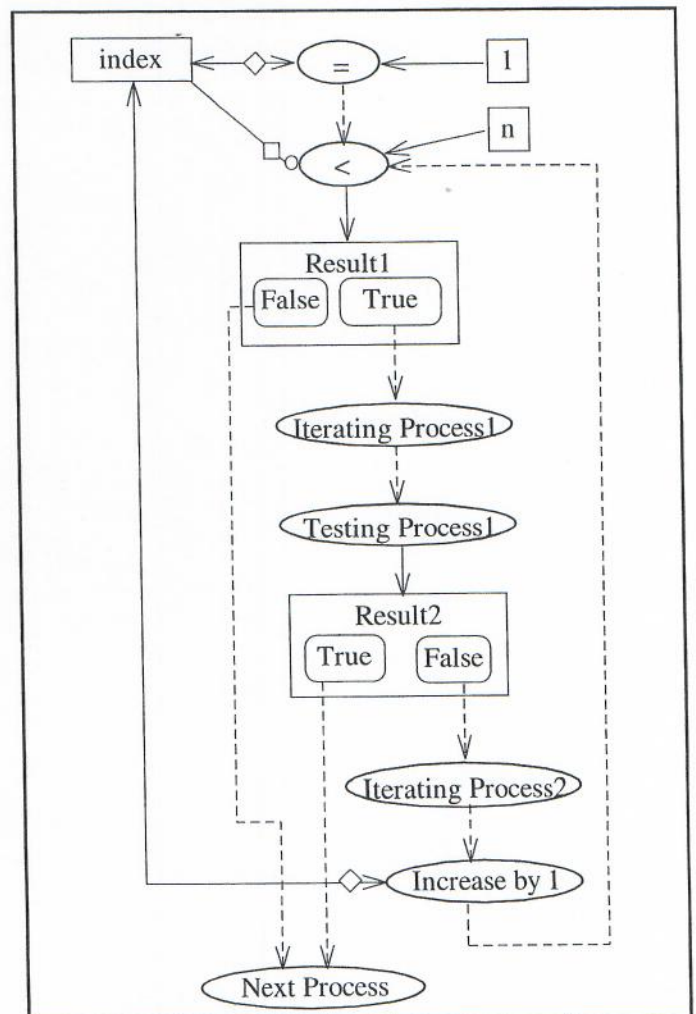


Figure 9. Illustration of a break link from a for iteration.

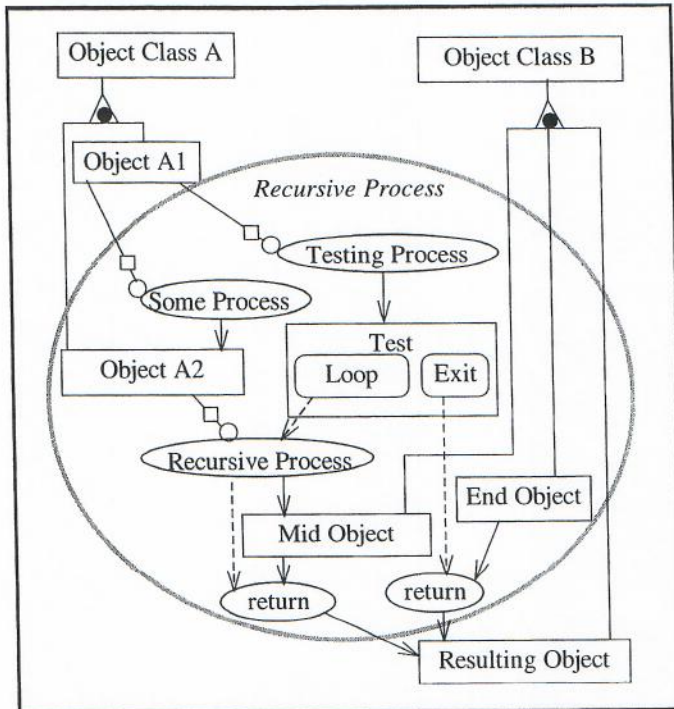


Figure 10. Illustration of recursion in OPD.

allowed in an iteration. A control link leading from the iterating process to the starting process is a continue link, while a control link leading from the iterating process to the end of the iteration

is a break link. Figure 9 is an OPD of the usage of a break link from a for iteration. Its corresponding C/C++ code is as follows:

```
for (index=1 ; index<n ; index++)
{
    Iterating_Process1() ;
    if (Testing_Process1() == True)
        break;
    Iterating_Process2();
}
```

Next_Process();

Recursion is another structure that prevails in the algorithms and programs and should be clearly expressed by OPDs at the algorithmic detailed design level. Inspired by Plan Calculus,² we use the same process inside the zoomed-in process to express re-

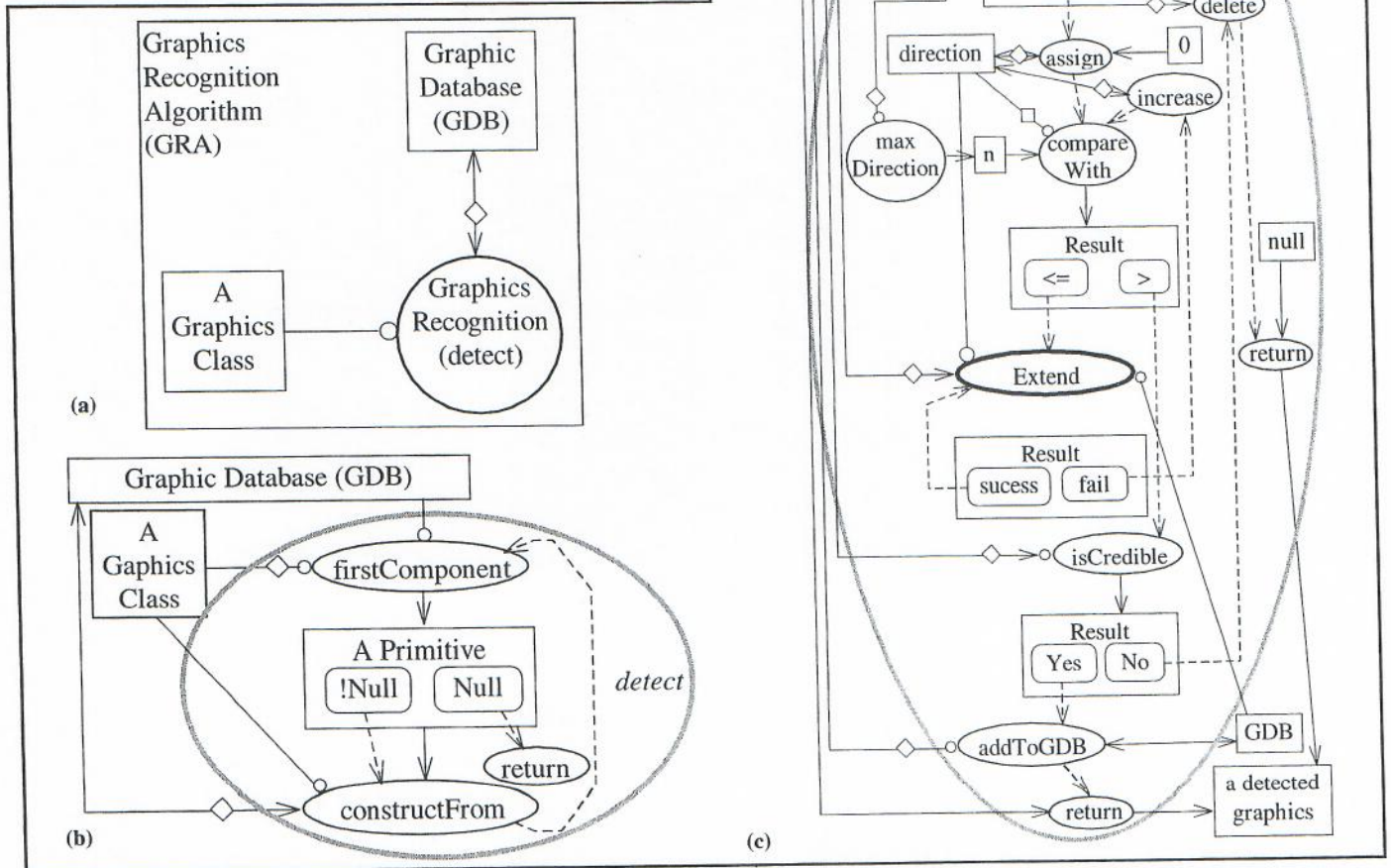


Figure 11. OPD illustration of the graphics-recognition algorithm (process). (a) top-level OPD; (b) zooming into "detect" (GRA) process in (a); (c) zooming into constructFrom process in (b).

cursion. The same process may also occur more than once at the same level if necessary to process different objects. At least one control branch should be involved in the recursion to terminate it. Figure 10 is an example of a recursion in OPD. Here is the corresponding C/C++ code fragment:

```
Object_Class_B&
Object_Class_A ::
Recursive_Process(void) const
{
    if (Testing_Process() == Exit)
        return End_Object;
    Object_Class_A& Object_A2
        = Some_Process();
    return Object_A2.Recursive_Process();
}
```

Representation of an entire algorithmic function

The details of a process (or *procedure* or *function*, as it may be called) are expressed in an OPD by scaling up the process. A special process, referred to as the *return process*, is introduced to terminate a procedure or function, as done in many programming languages. The number of inputs and the number of outputs of the return process should be equal and should be either 0 or 1. There may be many occurrences of the return process in a single OPD. Each of them concludes the blown-up process in a control-flow branch. The conditional branch control-flow mechanism is usually involved in OPDs when more than one occurrence of the return process appears in a process. If the conditional branch control flow consists of a return process, it is like the conditional break and continue control mechanisms in iterations, which do not have a joint process for such a branch. Figure 10 is an example in which the return process in the "exit" branch ends the blown-up process. If the return process outputs an object, the resulting object of the return process should be the same as the blown-up process object, and it is depicted outside that process, as shown in Figure 10.

A return process does not have an owner. It is a starting process inside a procedure and is not necessarily required. It can be deduced by the data and control-flow dependencies. Some auxiliary empty processes may also be used to keep the pattern representation consistent if necessary.

OPD REPRESENTATION OF THE GENERIC GRAPHIC-OBJECT-RECOGNITION ALGORITHM

In this section, we use a set of OPDs to describe a generic graphics-recognition algorithm.^{11,14,15} The algorithm is a two-step procedure. In the first step, we find a key component of a possibly existing graphic of the class we are detecting. In the second step, we construct graphics of this class from the key component found in the first step and try to extend it by finding its other components. The algorithm can be applied to the detection of any class of graphic objects.^{11,16} The algorithm is described in two C++ template functions shown in Listing 1. Its OPD representation is shown in Figure 11.

CONCLUSION

We have extended OPDs to handle algorithmic representation. Control flows in the algorithmic process are expressed by control links. Structural and procedural links among objects and processes are specified in more detail for the sake of implementation. The resulting extended OPDs can serve as a graphic tool for both detailed design and implementation specification, as well as for reverse engineering of existing program code. Transforming textual code into OPDs has great potential as a toll for software systems maintenance and redesign. ■

References

1. Ward, P. T. "The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing," *IEEE Transactions on Software Engineering*, 12(2):198-210, 1986.
2. Charles, R. and C. W. Richard. *The Programmer's Apprentice*, ACM Press and Addison-Wesley, Reading, MA, 1990.
3. Abelson, H. and G. J. Sussman. *Structure and Interpretation of Computer Programs*, McGraw-Hill, New York, NY, 1985.
4. Chen, P. P. S. "The Entity Relationship Model—Toward a Unifying View of Data," *ACM Trans. on Database Systems*, 1(1):9-36, 1976.
5. Booch, G. *Object-Oriented Analysis and Design with Applications*, Benjamin-Cummings, Redwood City, CA, 1994.
6. Dori, D. "Object-Process Analysis: Maintaining the Balance Between System Structure and Behaviour," *Journal of Logic and Computation*, 5(2):227-249, 1995.
7. Dori, D. and M. Goodman. "From Object-Process Analysis to Object-Process Design," *Annals of Software Engineering*, 9:1-25, 1996.
8. Liu, W. and D. Dori. "Extending Object-Process Diagrams for the Implementation Phase," in *Proc. of the Third International Workshop on the Next Generation of Information Techniques and Systems*, Neve Ilan, Israel, pp. 207-214, June 30-July 3, 1997.
9. Dori, D. "Automated Understanding of Engineering Drawings: An Object-Oriented Analysis," *Journal of Object-Oriented Programming*, 7(5):35-43, Sept. 1994.
10. Dori, D. and Y. Dori. "Object-Process Analysis of a Hypertext Organic Chemistry Studyware," *Journal of Computers in Mathematics and Science Teaching*, 15(1/2):65-84, 1996.
11. Liu, W. and D. Dori. "Automated CAD Conversion with the Machine Drawing Understanding System," *Proceedings of 2nd IAPR Workshop on Document Analysis Systems*, Malvern, PA, pp. 241-259, Oct. 1996.
12. Dori, D. and W. Liu. "Vector-Based Segmentation of Text Connected to Graphics in Engineering Drawings," *Advances in Structural and Syntactical Pattern Recognition*, P. Perner, P. Wang, and A. Rosenfeld, Eds., *Lecture Notes in Computer Science*, 1121:322-331, Springer, New York, NY, 1996.
13. Dori, D., D. Hubunk, and W. Liu. "Improving the Arc Detection Method in the Machine Drawing Understanding System," in *Document Recognition IV—Proc. SPIE'97*, L. M. Vincent and J. J. Hull, Eds., San Jose, CA, *SPIE* 3027:124-134, Feb. 1997.
14. Liu, W. and D. Dori. "A Generic Integrated Line Detection Algorithm and Its Object-Process Specification," *Computer Vision and Image Understanding (CVIU)*, Special Issue on Document Image Understanding and Retrieval, 70(3):420-437, 1998.
15. Liu, W. et al. "Object Recognition in Engineering Drawings Using Planar Indexing," *Proc. of the First International Workshop on Graphics Recognition*, Penn. State Univ., PA, pp. 53-61, 1995.
16. Liu, W. and D. Dori. "Genericity in Graphics Recognition Algorithms," *Graphics Recognition—Algorithms and Systems*, K. Tombe and A. Chhabra, Eds., *Lecture Notes in Computer Science*, 1389:9-21, Springer, New York, NY, Apr. 1998.

Quantity reprints of this article can be purchased by phone: 717.560.2001, ext. 39 or by email: sales@rmsreprints.com.