Transforming Business Process Models: Enabling Programming at a Higher Level

Wei Zhao
Computer and Information Sciences Department
University of Alabama at Birmingham
Birmingham, AL 35294-1170, U.S.A.
zhaow@cis.uab.edu

Barrett R. Bryant, Fei Cao
Computer and Information Sciences Department
University of Alabama at Birmingham
Birmingham, AL 35294-1170, U.S.A.
{bryant,caof}@cis.uab.edu

Kamal Bhattacharya
IBM T.J. Watson Research
P O Box 218, Yorktown Heights, NY 10598
kamalb@us.ibm.com

Rainer Hauser
IBM Zurich Research
Saeumerstrasse 4, Rueschlikon 8803, Switzerland
rfh@zurich.ibm.com

Abstract

Two practical paradigms are presented, which facilitate domain concepts to be directly used to model business operations: the first paradigm is based on the business artifacts and their life cycle; the second paradigm is based on the business tasks and their sequencing. Transformation is an effective way to bridge the gap between business level analysis and IT solutions. We present algorithms that transform business process models based on these two paradigms into IT solutions of web service platform. The specific problems we addressed in this transformation are: 1) how to generate the implementation code of an optimal size; 2) how to preserve the natural structure of business process models in the generated code.

1. Introduction

Even though the evolution of programming languages has provided the opportunity to raise programming abstraction significantly higher than using machine specific considerations, there is still a wide gap between the requirements analysis (domain-specific considerations) and technology platforms (virtual machines). Some existing modeling tool sets [For98, VWD, WBI] and graphical modeling languages such as UML [UML03] facilitate business analysts to directly model the business operations. In order to minimize the gap between the business level analysis and technology platforms, such models have to be automatically or semi-automatically transformed into executable code representations in the same way programs written in a high level programming languages are compiled into executable code.

This paper investigates two practical paradigms under which business operations can be modeled at an appropriate abstraction amenable to the business level users, or even to the end users. More importantly, these paradigms provide enough semantic formalism so that automatic or semi-automatic IT level representation can be derived. The specific contribution of this paper lies in the methods we proposed to realize this derivation.

In the first paradigm, business artifacts are treated as first class citizens, business operations are modeled by tracing the life cycle of business artifacts. In the second paradigm, business tasks are treated as the primary entities under consideration; business operations are represented by the ordering of business tasks. We build an abstract graph model for both paradigms. Depending on the different interpretation of the abstract graph model, the styles of the code generated from two paradigms are slightly different.

This paper is organized as follows. Two paradigms and the associated transformation systems are presented in sections 2 and 3 respectively. Section 4 discusses related work. The paper concludes in section 5.

2. Modeling with Business Artifacts

Business artifacts as an approach for modeling business operations were proposed by Nigam and Caswell [Nig03] and the approach is called Operational Specification (OPS). Business artifacts constitute concrete, identifiable, and self-describing information chunks that the business creates and maintains. Artifact processing is a way to describe the operations of a business. The end-to-end processing of a specific artifact, from creation to completion and archiving are captured in the artifact life cycle. Following the idea of modeling using business artifacts, Adaptive Business Objects (ABO) [Kum03], as a component model, explicitly defines the life cycle of the artifacts as a state machine. The state machine orchestrates a set of business activities and events that triggers the state transition in the artifact life cycle. ABO integrates

people, process, and information by explicitly modeling who (business role) can do what (business activity) with a particular artifact when (i.e. which state). For example, for the business collaboration between the provider and the retailer, the contract would be identified as an artifact to aggregate the terms and distributed data to be agreed on. The life cycle of the contract models the collaboration process. Figure 1 is a simplified life cycle of the contract. All the business events and activities happen on the edge of the graph. The nodes of the graph are the states of the artifacts in their life cycle.

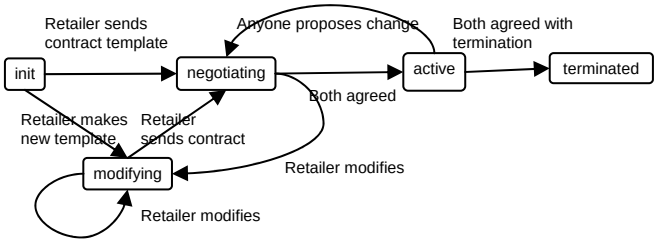

Figure 1. The life cycle of the contract between the provider and the retailer

Next, we discuss our method to transform ABO to an implementation platform, Business Process Execution Languages (BPEL) [BPE03]. In this paper, we concentrate on how the skeleton (the state machine) of the ABO is transformed into BPEL.

2.1. Problem Analysis on the Transformation from the State Machine to BPEL

BPEL is a structured web service orchestration language. A partner link is assigned to each collaborator in a business process. Each web service invocation is through a specific partner link. The partner link corresponds to the business role in ABO. The main control flow constructs in BPEL include "switch", "while", "sequence", and "flow". The "flow" construct indicates concurrent activities. The endeavor of this compilation is how to translate the directed graph (the state machine representation) to structured statements.

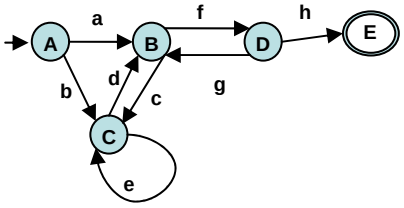

Figure 2. The abstract graph of figure 1

For the convenience, we use simple letters to mark the nodes and the edges of the graph in figure 1. The abstract graph is shown in figure 2.

There exist two simple approaches that can translate a directed graph to structured statements. The first one is called the State Machine Controller (SMC) approach. It has been proved by [Boh66] that every program can be translated into a program with a single "while" and a single "switch" statement. Based on this approach, figure 2 can be translated to the following pseudo code. The correspondence of the pseudo code and BPEL is a straightforward one-to-one correspondence. In order to save space, we present our code fragments in pseudo code.

```
nextnode=A
while nextnode !=E
   switch
   case nextnode =A
          in A
          if event-a, nextnode =B
          if event-b, nextnode =C
   case nextnode =B
         in B
         if event-f, nextnode =D
         ……
```

The drawback of this approach is run time inefficiency for large business processes. There will be an average test time of O(n/2) for every state transition, where n is the number of nodes in the process graph. It can be seen that the semantic process structure is not revealed from the code at all because of the uniform code representation. Furthermore, this approach essentially encodes "goto", a low-level language concept, in high-level language constructs. However, the advantage of this simple method is that the code size is O(n), where n is the number of edges (the number of "if" statements is as big as the number of edges).

The second simple method traverses the graph and outputs two types of code at each node: a conditional statement such as "switch" or "if-else", and a loop statement such as "while" if the node is the entry of a natural loop. A natural loop is a loop that has a dominating entry, e.g., the loop with node B and D. Based on this scheme, the pseudo code for figure 2 is:

```
in A
switch
  case event-a, in B
     while event-f
         in D
         switch
            case event-h, in E, exit
            case event-g, in B
  case event-b, in C
     ……?
```

However, the loop involved with B and C cannot be represented because it does not have a dominating entry, i.e. two entries B and C do not dominate each other. This type of the loop is called an irreducible loop [Aho86]. Zhao et al. [Zha05] have an in-depth analysis of the definitions and the problems with irreducible loops. A

graph that contains irreducible loops is called an irreducible graph. Irreducibility is a classic problem in compiler theory. Traditionally, irreducibility is solved by using node splitting techniques [Coc69] to translate an irreducible graph to a reducible graph. However, node-splitting techniques result in an equivalent but exponential reducible graph [Car03], or at most with the controllable size but worst case exponential complexity [Jan97].

2.2. Proposed Algorithm on the Transformation from the State Machine to BPEL

Our approach is based on the observation that a Regular Expression (RE) is a theoretical model of the structure of the structured programming languages. RE has structured control flow constructs: concatenation, or, and star. Therefore, we first transform the state machine to an RE; RE is then translated into BPEL using syntax-directed translation. Instead of treating an RE as a definition of a regular language, we consider an RE sentence as a program written in a Regular Expression Language (REL). Therefore, the REL can be easily customized to support specific features of our transformation system. This approach named REL has been discussed in detail in [Zha05]. However in REL, whenever there is an irreducible loop, some nodes and edges (i.e., paths) are repeated in the generated code. Strategies are also discussed in [Zha05] that control the complexity of the generated code by minimizing the repetition. Nevertheless, the strategies cannot eliminate the repetition at all. Therefore the upper bound (in the case of strongly connected components) of the generated code is exponential to the number of the nodes and edges in the input graph.

The solution presented in this paper solves the problem of the exponential size by combining REL with the SMC approach. The general idea is that whenever an irreducible loop is detected during the transformation from state machines to REL, we switch to SMC and divert back to the normal process after the irreducible loop is passed. The resulting combined method generates code of size O(n) where n is the number of edges in the process model for an arbitrary directed graph. This will be proved later in this section. At the same time, the algorithm preserves the natural structure of the business process in the generated BPEL code. The preservation of process structure is needed because the generated BPEL code from our algorithm is abstract BPEL code. Software engineers have to go through the generated code to bind the specific information about each web service such as IP and port in order to make the BPEL executable. This platform-specific information is not available at the business modeling level. As a result, our algorithm overcomes the disadvantages of both two simple methods in section 2.1.

REL uses the Set Equation algorithm [Den78] to translate a directed graph into an RE. The first step of REL is to extract a set of equations for a directed graph. The set of equations for the graph in figure 2 is as follows ("+" means XOR):

```
A=aB + bC                          (1)
B=cC + fD                          (2)
C=eC + dB                          (3)
D=gB + hE                          (4)
E=• (since E is the final state)   (5)
```

In these equations, the capital letters (nodes) can be treated as unknowns as in mathematical equations; the lower case letters (the edges) are the coefficients.

There are 3 types of rules to solve the set of equations (1) through (5).

1. Standard algebraic substitution: substitute an unknown with its value.
2. Standard RE algebraic laws:
   a. Commutative +: R+S = S+R
   b. Associative +: R+ (S+T) = (R+S) +T
   c. Associative concatenation: R (ST) = (RS) T
   d. Distributive +: R(S+T)=RS+RT, (S+T)R =SR+TR
   e. Anti-distributive + is the inverse of d.
3. Arden's rule for the removal of self-recursion. For example, `C=eC+dB => C=e*dB`. Please refer to [Den78] for the proof of Arden's rule.

To solve the equations, all the unknowns have to be substituted except the start node. The solution of the start node, node A in our example, would be the resulting RE.

In [Zha05], we use the mathematical notations as in equations (1) to (5). In this paper, we present the equations and rules based on their implementation structure. The overview of the implementation of REL is shown in figure 3. There are 4 major components: the extractor extracts some abstract information needed for computation such as the abstract graph and the equations; the equation solver solves the equation based on this information; the generated RE syntax tree goes through the RE optimizer; code generation is achieved by using the visitor pattern [Gam95] to walk through the RE syntax tree.

The class diagram of the implemented system is shown in figure 4. The abstract graph is represented by the *Node* class that for each node records the outgoing edges, the next states, the predecessors, the dominator set, and the equation. RE is implemented with an object-oriented structure based on the definition of RE. Each RE class is also a tree node. To give an example, the object-oriented syntax tree of A's equation is shown in figure 5. We solve the equations by manipulating the RE syntax tree of each node. Upon the completion of

equation solving, we get a single big tree for the start node, which is the final RE solution for the graph.

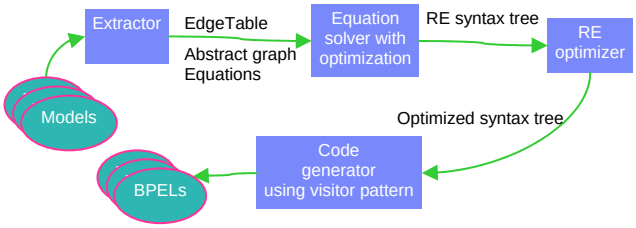

Figure 3. The implementation overview of REL

We now introduce the definition of the Irreducible Loop Region and how the equation solving rules are adapted to accommodate SMC. An *Irreducible Loop Region (ILR)* consists of the Common Immediate Dominator (CID) of the loop entries and a set of nodes that can reach the loop exits without going through CID. The CID of an ILR is called the header of the ILR. The composition relationship of ILRs and ILRs, and ILRs and natural loops is: they are disjoint, nested, or sharing the same header. The detailed proof of this property is presented in [Zha05]. This property also holds true for composition of natural loops [Aho86].

The loop exits can be detected automatically when the loop exit optimization is applied after the application of Arden's rule. An ILR can also be detected automatically if the equations are solved bottom up from the dominator tree of the graph. The dominator tree of figure 2 is shown in figure 6. Specifically, bottom-up means the reverse order of the breadth-first enumeration of the dominator tree. If there is a path in the abstract graph connecting nodes that have the same immediate parent in the dominator tree, there is an ILR. The nodes that are involved in this path are the loop entries; their immediate parent node is the CID. When an ILR is detected, the normal equation solving process is interrupted and we translate the ILR using the SMC approach. The principle observation for this idea is that both the natural loop and the ILR can be collapsed into a single node: a natural loop can be collapsed into its dominating entry; an ILR can be collapsed into its CID.

Two more governing rules for substituting nodes with the same immediate parent in the dominator tree: 1) if no ILR is detected, nodes at the same layer can be substituted in any order; 2) nodes that are not involved in an ILR should be substituted before the ILR.

Let's look at the example in figure 2. Traversing the dominator tree in a bottom-up direction, the first and the second candidate nodes for substitution are E and D.

After E and D have been substituted, the equations of A and C remain the same. Only B's equation is updated. The tree structure of B's equation is shown in figure 7.

```
A=aB + bC
B=cC + f(gB+h)
C=eC + dB
```

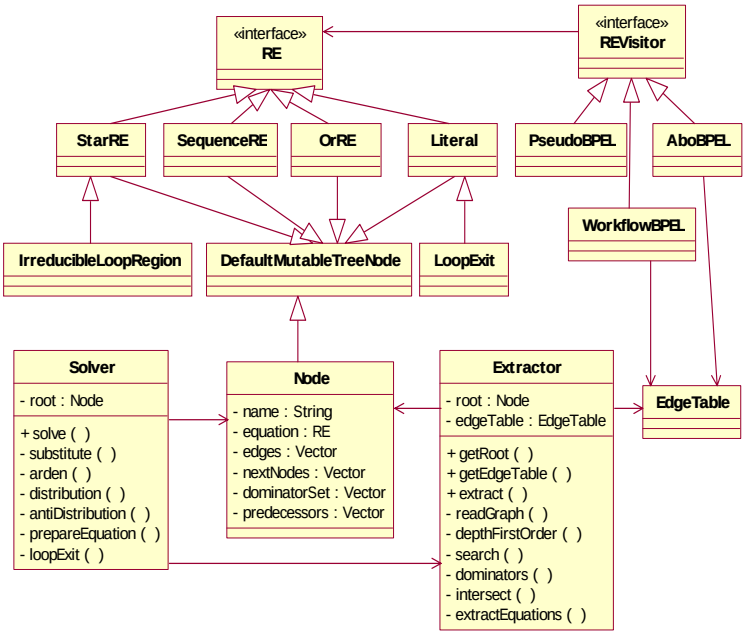

Figure 4. The class diagram of the implemented system

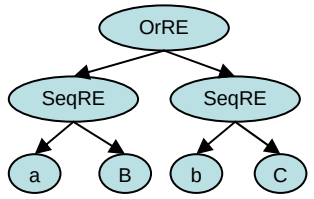

Figure 5. The A's equation

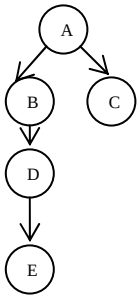

Figure 6. The dominator tree of figure 2

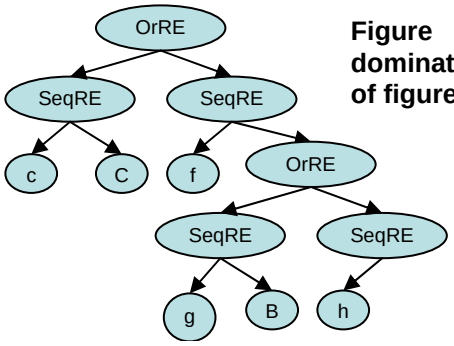

Figure 7. B's equation after the substitution of E and D

At this point, we have 3 nodes left: A, B, and C. Since B's equation has node C and C's equation has node B, there is a path connecting B and C. At the same time, B and C has the same immediate parent in the dominator tree. Therefore, an irreducible loop is detected with B and C as loop entries. A is the CID for this loop. All the nodes that are dominated by B or C are the nested structure rooted at B or C respectively. A, B, C, and their nested structures comprise an ILR. As can be seen from this illustration, whenever an ILR is detected all the nested nodes should have already been substituted. The reason is obvious. Based on our substitution order, all the dominated nodes will be substituted before the dominating nodes.

In the implementation, the body of the ILR is represented by the *IrreducibleLoopRegion* object, a sub-class of the *StarRE*. The *IrreducibleLoopRegion* stores the RE for the CID and the loop entries. Before this information can be stored, extra operations have to be performed on the equations of loop entries.

The equation for each entry node is factored into two equations so that one equation contains the path to other entry nodes and the other equation denotes the nested structures. In our example, both B and C have a nested natural loop with B and C as the header respectively. Therefore, Arden's rule should be applied to B and C for the nested natural loop.

```
B'= cC
B''=f(gB+h)=fgB+fh =(fg)*fh=(f[h]g)* h
```

The rules used to derive the equation for B'' are (in this order) distribution, Arden's rule, and loop exit optimization. The loop exit optimization puts the exit points (denoted by "[ ]" syntactical markers) explicitly inside the loop body. The syntax tree structure of the equation B'' is shown in figure 8. Similarly, C's equation is separated into:

```
C'= dB
C''=e*
```

In this example, the equation of A only contains the separate paths to B and C. If A contains a path to some other nodes or contains nested structures, A's equation should also be factored.

Using Arden's rule for natural loops results in a *StarRE* followed by the loop exits (e.g., `h` in the equation of B''). Likewise, using SMC for an ILR results in an *IrreducibleLoopRegion* followed by the exits of this ILR. The exits out of an ILR are all the exits detected within the body of the ILR. In this example, only one exit is detected, that is h. The final tree is presented in figure 9. As can be imagined, an ILR can certainly be nested deeply in a larger graph.

Code generation is performed by traversing the syntax tree structure using the visitor pattern. The concrete semantic content of each edge and the next state of the each edge can be obtained by looking up the *EdgeTable* indexed by the edge identifiers. The pseudo code that should be generated from figure 9 is shown below. The case statements in the loop of SMC-style are generated from the first set of the factored equations (e.g., B' and C').

```
nextnode=A
while (nextnode !=E)
   case nextnode=A
      in A
      case event-a, nextnode=B
      case event-b, nextnode=C
   case nextnode=B
      in B
      if event-c, nextnode=C
      while event-f,
          in D
          if event-h, nextnode=E
          if event-g, in B
   case nextnode=C
      in C
      if event-d, nextnode=B
      while event-e
          in C
if event-h, in E
```

This example demonstrates the case in which natural loops are nested inside an ILR. Other cases such as an ILR nested inside a natural loop or sharing the header with a natural loop can also be represented using our algorithm. We proved that we can generate code of size O(n) for reducible graph (i.e. it contains only natural loops) using REL [Zha05]. We have shown in section 2.1 that SMC generates code of size O(n). Because we only have three types of composition relationships of ILRs and natural loops, we claim that the resulting combined method generates code of size O(n) for any arbitrary graph. Compared to the pure SMC approach, this algorithm better preserves the natural structure of the business model in the generated code.

3. Modeling with Business Tasks

The principal consideration of modeling with business tasks is to define the operational order of a set of atomic business tasks and sub-processes that again consist of atomic business tasks and sub-processes. This consideration underlines workflow modeling languages[1] such as UML and Business Process Modeling Notation (BPMN) [BPM].

[1] More information on workflow modeling languages can be found at http://tmitwww.tm.tue.nl/research/patterns/

The scheme for transforming a workflow into BPEL is almost the same as transforming the state machine into BPEL except for 4 differences:

1. To translate a workflow language into an abstract graph, we need first to remove all the decision nodes. All the decision nodes can be merged with the immediate predecessor task node because the decision node provides no additional computational semantics as there is always a true semantics from a task node to its immediate following decision node.
2. Code generation for state machine and the workflow is based on different interpretation of the abstract graph.
3. The RE syntax tree nodes are annotated with two more attributes: *code* and *condition*.
4. REL is extended to transform the concurrent processes into BPEL *flow* construct.

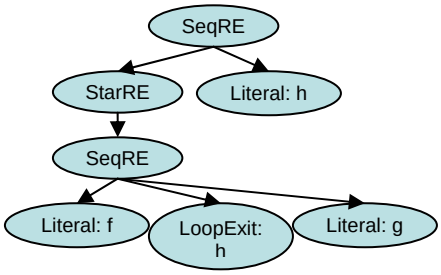

Figure 8. The syntax tree of B''

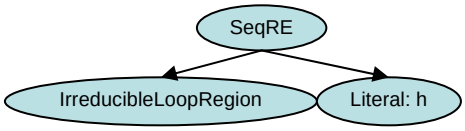

Figure 9. The final RE solution for figure 2

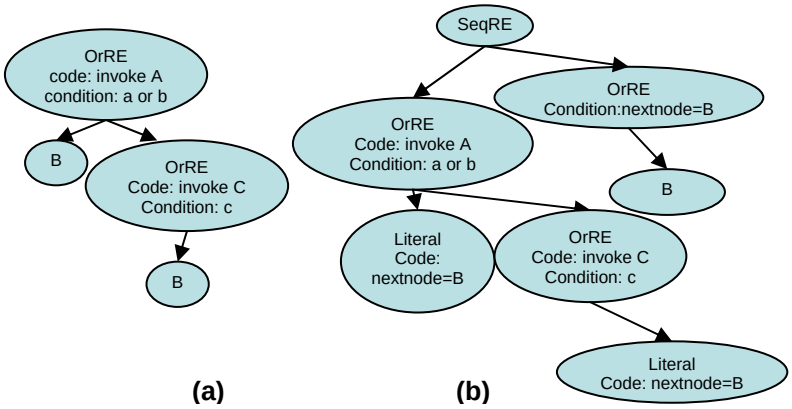

Figure 11. The factorization rule
(a): left-hand side . (b): right-hand side

The abstract graph of a workflow language is also a directed graph with only a slightly different interpretation: in the state machine, events or operations happen on the edge of the graph; in workflow languages, each node denotes a business task. We refer to the first interpretation as edge-based interpretation and the second one as the node-based interpretation. If we select BPEL as the low level implementation, each business task is an invocation of a web service. The difference in the interpretation of the abstract graph gives different styles of the generated code. The edge-based interpretation emphasizes how each edge is traversed; the node-based interpretation emphasizes how each node is visited.

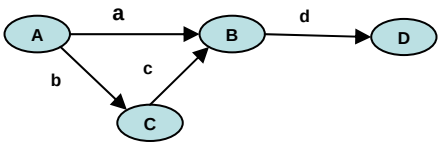

Figure 10. An example abstract graph

We look at the example graph in figure 10. If we treat this abstract graph as a state machine, we generate the following code. The code strictly interprets how each edge is traversed.

```
in A
switch
  case event-a, in B
  case event-b, in C
      event c, in B
event-d, in D
```

If we directly apply REL to node-based interpretation, we could generate the following code for figure 10:

```
invoke A
switch
  case a, invoke B
  case b, invoke C
      if c, invoke B
if d, invoke D
```

There are two identical `invoke B` statements. The executable code for each web service invocation has to include the bindings of operations, parameters, IP, and port. By eliminating the repetition, we not only minimize the size of the generated code, but also reduce the error rate. If there are multiple invocations of web services,

engineers might overlook one copy of the invocation or introduce inconsistency among multiple identical invocations in the process of manually binding web services. The factorization rule (introduced by [Amm92, Hau04, Koe04]), shown in figure 11, has to be used to eliminate multiple copies of `invoke B`. The resulting code precisely interprets how each node is visited.

```
invoke A
switch
    case a, nextnode=B
    case b, invoke C,
          if c, nextnode=B
if nextnode=B, invoke B
    if d, invoke D
```

Modeling with the life cycle of the business artifacts, an artifact is a unique entity; therefore, it can not have multiple states at any one time. As a result, there is no notion of synchronization in OPS and ABO. However, in the case of workflow, it makes sense to have multiple activities happening simultaneously. A concurrent region in the process model should be translated into a *flow* construct in BPEL. Each thread can be a complex business process; therefore, the transformation for each thread inherits the problems aforementioned such as irreducible loops. The set equation algorithm is customized to support the translation from concurrent flows to REL. The algorithm for transforming concurrent flows will be presented elsewhere.

4. Related Work

Different methods have been proposed for transforming models to models or to code [Cza03]. The visitor-based approach [Gam95] is a simple way of code generation. We also use the visitor pattern to traverse the object structure of REL to output BPEL code. However, all the existing model-to-code and model-to-model transformation systems such as template-based [Cle01], relational [Ake02], graph-transformation [And99, Agr03], and structure-driven[2] do not solve the particular problem we address for transforming a graphical process model based on "goto"s to structured statements. The existing transformation approaches aim at defining a framework that can map the entities from the source to the entities of the target; this paper investigates how to map the *structure* of the entities in the source model to the *structure* of the entities in the target model, and the source and the target are structuring things based on different principles.

There is some work on translating UML activity diagrams or similar behavior models to structured languages [Pat04, Sko04]. However they did not address specifically how loops are handled in the transformation especially when the irreducible loops are presented.

Goto-elimination has been used for removing gotos in a program [Amm92] and for compiling goto-based business process models to structured statements [Hau04, Koe04]. The latter use node-based interpretation for the abstract process graph, whereas the work in this paper interprets the graph based on both the edge-based and node-based interpretation. The methods presented in [Hau04, Koe04] do not guarantee the optimal code from the irreducible process model. Furthermore, we integrate the transformation for the non-concurrent processes and concurrent processes in REL.

RE has been used as an intermediate step through which a program with goto statements can be transformed into programs without gotos [Mor99]. However, the method uses a uniform RE "E_1*E_2"to represent the program, where E_1 is the elementary nontrivial paths back to the start node and E_2 is the paths to the stop node that do not return to the start node. The uniform RE hurts the natural structure of the process graph. More important, the method ignores the optimizations. The resulting RE is usually unnecessarily complex. In addition, [Mor99] only discusses the transformation of non-concurrent processes.

5. Conclusions

In this paper, we discuss two practical ways business operations are modeled. We did not intend to compare which paradigm is better, but rather demonstrate ways such high level programs can be transformed into low level implementations. The transformation system builds an abstract graph from the input models. We offer compilation options so that users can indicate in which paradigm the input model is developed. The system use edge-based interpretation or node-based interpretation based on the user supplied compilation options.

We use REL as an intermediate representation. REL gives the opportunity that the core part of algorithm can be reused for multiple input and output models. Based on our experience, when applying the algorithms to large industry examples (more than 100 nodes per model), REL is a powerful and concise way for us to see the structure of the code to be generated. It is usually difficult to comprehend the overall structure of the BPEL code for large scale process models without carefully reading it through.

In particular, the contribution of this paper lies in*:* 1) we guarantee the generated BPEL code for any arbitrary directed graph is of size O(n) where n is the number of edges of the process model; and 2) the structure of

[2] OptimalJ, http://www.compuware.com/products/optimalj/

business process models is naturally reflected in the generated code.

6. Acknowledgement

This research is supported in part by the U. S. Office of Naval Research under the award number N00014-01-1-0746, and the IBM student internship program.

7. References

[Agr03] A. Agrawal, G. Karsai, F. Shi, "Graph Transformations on Domain-Specific Models", technical report, ISIS-03-403, 2003.

[Aho86] Aho, A., Sethi, R., and Ullman, J., *Compilers-Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[Ake02] D. H. Akehurst S. Kent, "A Relational Approach to Defining Transformations in a Metamodel", Proc. of UML 2002, pp. 243-258, 2002.

[Amm92]Ammarguellat, Z., "A Control-Flow Normalization Algorithm and Its Complexity", *IEEE Transactions on Software Engineering*, Vol. 18, No. 3, 1992.

[And99] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schurr, G. Taentzer, "Graph Transformation for Specification and Programming" Science of Computer Programming, Vol. 34, No. 1, pp. 1-54, 1999.

[Boh66] Bohm, C., and Jacopini, G., "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules", *Communications of the ACM*, 9(5), pp.366-371, 1966.

[BPE03] Business Process Execution Language for Web Services, Version 1.1, 05 May 2003, http://www-106.ibm.com/developerworks/library/ws-bpel/

[BPM] BPMN specification: http://www.bpmn.org/Documents/BPMN%20V1-0%20May%203%202004.pdf

[Car03] Carter, L., Ferrante, J., and Thomborson, C., "Folklore Confirmed: Reducible Flow Graphs are Exponentially Larger", *Proc. of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 106-114, 2003.

[Coc69] Cocke, J., and Miller, R. E., "Some Analysis Techniques for Optimizing Computer Programs", *Proc. of 2nd Hawaii International Conference on Systems Sciences (HICSS)*, pp. 143-146, 1969.

[Cle01] J. C. Cleaveland. *Program Generators with XML and JAVA*. Prentice Hall 2001.

[Cza03] K. Czarnecki, S. Helsen, "Classification of Model Transformation Approaches", OOPSLA'2003 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.

[Den78] Denning, P. J., Dennis, J. B., and Qualitz, J. E., *Machines, Languages, and Computation*, Prentice-Hall, Inc., 1978.

[For98] Fort´e, *Fort´e Conductor Process Development Guide*, Fort´e Software, Inc. Oakland, CA, USA, 1998.

[Gam95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[Hau04] Hauser, R., and Koehler, J., "Compiling Process Graphs into Executable Code", *Proc. of the 3rd International Conference on Generative Programming and Component Engineering (GPCE'04), LNCS 3286, pp. 317-336*, 2004.

[Jan97] J. Janssen, H. Corporall, "Making Graphs Reducible with Controlled Node Splitting", ACM Transaction on Programming Languages and Systems, Vol. 19, No. 6, pp. 1031-1052, 1997.

[Koe04] Koehler, J., and Hauser, R., "Untangling Unstructured Cyclic Flows - A Solution based on Continuations", *Proc. of the International Conference on Cooperative Information Systems, LNCS 3290, pp. 121-138*, 2004.

[Kum03] S. Kumaran, P. Nandi, "Adaptive Business Object: A New Component Model for Business Applications", white paper, IBM T. J. Watson Research Center, http://www.research.ibm.com/people/p/prabir/ABO.pdf

[Mor99] Morris, P. H., Gray, R. A., and Filman, R. E., "GOTO Removal Based on Regular Expressions", *Journal of Software Maintenance: Research and Practice*, Vol. 9, No. 1, pp. 47-66, 1999.

[Nig03] A. Nigam, N. S. Caswell, "Business Artifacts: An Approach to Operational Specification", IBM Systems Journal, Vol. 42, No. 3, 2003.

[OMG] OMG RFP for Business Process Definition Metamodel, http://www.omg.org/cgi-bin/doc?bei/2003-01-03

[Pat04] Patrascoiu, O., "Mapping EDOC to Web Services Using YATL", *Proc. of the 8th International IEEE Enterprise Distributed Object Computing Conference*, 2004.

[Sko04] Skogan, D., Gronmo, R., and Solheim, I., "Web Service Composition in UML", *Proc. of the 8th International IEEE Enterprise Distributed Object Computing Conference*, 2004.

[UML] UML 2.0 Superstructure Final Adopted Specification, ptc/03-08-02, http://www.omg.org/cgi-bin/doc?ptc/2003-08-02

[VWD] Visual Workflow Designer, Softwarium company, http://www.softwarium.net/WorkflowDesigner.php

[WBI] IBM WebSphere Business Integration Modeler: http://www-306.ibm.com/software/integration/wbimodeler/

[Zha05] W. Zhao, R. Hauser, K. Bhattachaya, B. Bryant, "Compiling Business Processes: Untangle Unstructured Loops in Irreducible Flow Graphs", Technical report UABCIS-TR-2005-0505-1, Department of Computer and Information Sciences, University of Alabama at Birmingham, 2005, http://www.cis.uab.edu/zhaow/papers/irreducible.pdf.