
Compiling business processes: untangling unstructured loops in irreducible flow graphs

Wei Zhao*

Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL 35294-1170, USA
E-mail: zhaow@cis.uab.edu
*Corresponding author

Rainer Hauser

IBM Zurich Research
Saeumerstrasse 4
Rueschlikon 8803, Switzerland
E-mail: rfh@zurich.ibm.com

Kamal Bhattacharya

IBM T.J. Watson Research
P.O. Box 218, Yorktown Heights, NY 10598, USA
E-mail: kamalb@us.ibm.com

Barrett R. Bryant and Fei Cao

Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL 35294-1170, USA
E-mail: bryant@cis.uab.edu
E-mail: caof@cis.uab.edu

Abstract: This paper presents a systematic study of some major problems involved in the transformation of business process modelling languages to executable business process representations. A business process modelling language usually uses the simple concept of 'goto' to describe the operation sequence of a business. If a structured language is chosen as the executable representation, it is difficult to compile the unstructured goto flows into structured statements when the process model is irreducible. This paper discusses a method called Regular Expression Language (REL). REL is a method that can compile a business process model that is irreducible with unstructured loops to statements in structured languages with controlled code complexity. Examples are given for compiling models expressed as UML2 Activity Diagrams into the Business Process Execution Language for Web Services (BPEL4WS).

Keywords: business process model; Model Driven Architecture (MDA); model transformation; unstructured loop; irreducible loop; regular expression; compiler optimisation; Business Process Execution Language.

Reference to this paper should be made as follows: Zhao, W., Hauser, R., Bhattacharya, K., Bryant, B.R. and Cao, F. (2006) 'Compiling business processes: untangling unstructured loops in irreducible flow graphs', *Int. J. Web and Grid Services*, Vol. 2, No. 1, pp.68–91.

Biographical notes: Wei Zhao is a doctoral candidate of Computer and Information Sciences at the University of Alabama at Birmingham. Her primary research areas include model driven development, business process modelling and management, component-based software engineering, and theory and implementation of programming languages. Her research has been supported by the Naval Office of Research. She has completed eight months internship in IBM T.J. Watson Research Center in 2004. She is student member of IEEE.

Rainer Hauser is with the IBM Research Division, Zurich Research Laboratory, Rüschlikon, Switzerland. Dr. Hauser is Research Staff member in the Computer Science Department at the Zurich Research Laboratory. He received a diploma in Mathematics and a PhD in Computer Science from the Swiss Federal Institute of Technology (ETH) in 1977 and 1984. He joined IBM at the Zurich Research Laboratory in 1980 initially as a PhD student, where he has worked on image processing. He later joined the laboratory after completing his PhD. He has been part of the team working on communication systems software.

Kamal Bhattacharya is member of the Business Informatics Department, focuses on the application of model-driven architecture concepts to real-world business scenarios, adaptive enterprises and business performance management. Prior to joining IBM Research in 2001, Dr. Bhattacharya served as an e-business IT Architect for IBM Global Services in Germany and worked on several large-scale e-business projects in the automotive and travel industry. He received doctoral degree in Theoretical and Computational Physics from Georg-August University, Goettingen, Germany, in 1999.

Barrett R. Bryant is Professor and Associate Chair of Computer and Information Sciences at the University of Alabama at Birmingham (UAB). He joined UAB in 1983 after completing his PhD in Computer Science at Northwestern University. His primary research areas are on the theory and implementation of programming languages, formal specification and modelling and component-based software engineering. He has authored and coauthored over 100 technical papers in these areas. Bryant is member of ACM, senior member of the IEEE and member of the Alabama Academy of Science. He is an ACM Distinguished Lecturer and is Chair of the ACM Special Interest Group on Applied Computing (SIGAPP).

Fei Cao is a doctoral candidate at the University of Alabama at Birmingham. He was under the supervision of Dr. Bryant. His work has been supported by the Naval Office of Research. His research interests include model-driven software development, aspect-oriented programming, generative programming and distributed software system. He is a student member of ACM and ACM SIGSOFT.

1 Introduction and motivation

The evolution of programming languages (from machine languages, to assembly languages, high-level languages and modelling languages) provides the technical foundation for the transition from machine-centric development to domain-centric development. The languages for machine-centric development are designed with machine specific considerations in mind such as cache, register and imperative features. In domain-centric development, programming is performed with domain-specific concepts such as business tasks and business artifacts. Business process modelling (Frank *et al.*, 2004) is one type of domain-centric development that concerns the dynamic behaviour of a business domain (*i.e.*, an enterprise). A business process consists of a set of atomic business tasks and/or sub-business processes. Business process modelling provides an appropriate programming abstraction for business level users.

Model Driven Architecture (MDA) (Frankel, 2003) is the result of programming language evolution. In the MDA framework, business analysts create business process models called Platform Independent Models (PIMs). A business process PIM is not executable since the atomic tasks are abstract and non-executable semantic entities. A compilation engine can realise a PIM in different Platform Specific Models (PSMs) so that the process model can be executed. This paper discusses a compilation method called Regular Expression Language (REL) that compiles a business process PIM to a PSM, in which the atomic tasks are implemented as web services. Examples are based on a specific input PIM – UML activity diagram (UML2, 2003), and a specific output PSM – Business Process Execution Language for Web Services (BPEL4WS, in the rest of the paper, we will simply refer to it as BPEL) (BPEL, 2003). The contribution of REL is that a business process PIM that is irreducible with unstructured loops can be compiled into statements in structured PSM with controlled code complexity.

The rest of the paper is organised as follows. Section 2 presents an example and examines the problems we are addressing. Section 3 explains the details of REL. The implementation and experimental results are documented in Section 4. Section 5 compares our work to some related work. The paper concludes in Section 6.

2 Problem analysis

Throughout the paper, we will use the online product purchase system from Koehler *et al.* (2005) as an example. The example, shown in Figure 1, is simple and common enough to serve as an introduction. Yet, the example is complex enough to demonstrate various problems we are addressing. The description of the example is as follows:

From the initial page, an existing user has to log on to the system. If the log succeeds, the authentication service is invoked. Authentication can fail if the user provides an incorrect password, in which case the log on is repeated. If log on detects that the user does not exist, he/she should be directed to the register function.

From the initial page, new users should register. If they succeed, they can go to the log on page. The registration page can be repeated in case of mistakes.

After authentication, the user can select and configure products. The process is flexible enough so that at each step, the user can go back to make changes. There is a verification service running in parallel to make sure that the product purchase is secure.

The dynamic behaviour of this business process is captured in the UML activity diagram in Figure 1. The activity diagram contains activity nodes and activity edges. The activity node includes executable nodes (represented as rounded rectangles denoting the atomic business tasks such as logon and register), the object nodes (an abstract activity node for defining object flow, not present in Figure 1) and the control nodes (decision nodes, merge nodes, fork nodes and join nodes). The executable node, the object node and the merge node (node *M* in Figure 1) are not distinguished from the perspective of the compilation. The decision node (denoted as numbered *D* node, e.g., *D1* tests whether the user is new) can be merged with the immediate predecessor executable node because the decision node provides no additional computational semantics. This is because there is always a true semantics from an executable node to its immediate following decision node. Therefore, for the purpose of compilation, Figure 1 can be normalised to Figure 2.

Figure 1 The activity diagram for the online purchase system

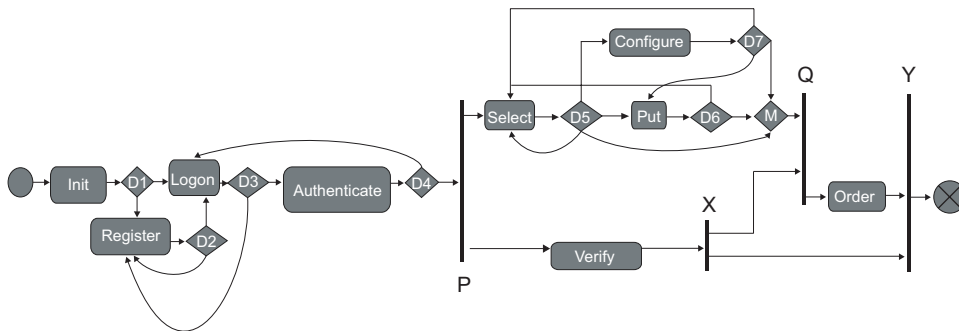
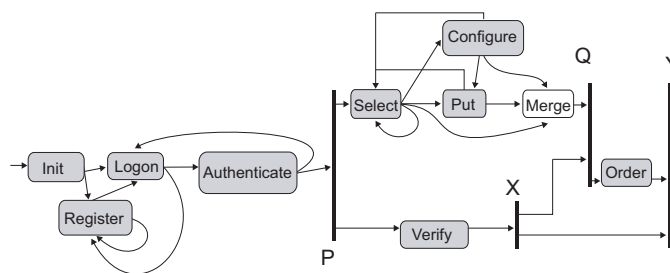
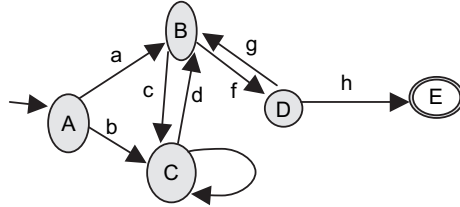


Figure 2 The normalised process model of Figure 1



The activity edges in an activity diagram include the control flow and the object flow. The semantic content (the evaluation of a test condition, the flow of a particular type of object or a condition of an event) of each edge is irrelevant for compilation. The compiler assigns each edge a unique id (e.g., the alphabetic letters in Figure 3). A hash table with the edge's ID as the index contains the next state for this edge and the semantic content of the edge.

Figure 3 The FA for the authentication portion of Figure 2

Note: A = init; B = logon; C = register; D = authenticate; E = P

However, the semantics of how each transition edge is activated is important to the compiler. The action nodes, object nodes, decision nodes and merge nodes have the XOR semantics on the associated edges. Exactly one incoming edge is activated, and the activity node enables exactly one outgoing edge. The fork node has the AND semantics on its outgoing edges; the join node has AND semantics on its incoming edges.

It is clear that a UML activity diagram is formed by a set of 'gotos'. 'Goto' is desirable at the business level modelling because of its simplicity. However, the target PSM, BPEL, is a web service orchestration language with explicit structured control flow constructs such as 'switch', 'while' and 'flow' (for concurrency). The main task of the compiler is to translate the unstructured 'goto' flows into well structured statements in BPEL. We identify two problems in this translation, detailed in the following sections.

2.1 The problem with unstructured loops

Figure 3 is an abstract representation for the part of the process before the fork node *P* in Figure 2. Each node is named by a letter. It is easy to see that Figure 3 is a Finite Automaton (FA). A non-concurrent process model with a single start node and with the XOR transition semantics can always be normalised into an FA.

A naïve transformation algorithm could traverse the graph and could output two types of code at each node: first, a conditional statement such as 'switch' or 'if-else'; second, a loop statement such as 'while' if the node is the entry of the loop. In order to write out the loop statement, a standard loop detection algorithm, such as back-edge detection (Aho *et al.*, 1986), has to be adopted as a pre-process to detect the loop entry. Suppose we have detected *B* as the entry of the loop involved with *B* and *D*, we can output the pseudo BPEL code for Figure 3 as follows:

```

invoke A
switch
  case a, invoke B
    while f
      invoke D
      switch
        case h, invoke E, exit
        case g, invoke B
      case b, invoke C
    .....?
  
```

Each invoke statement represents a web service invocation. We notice that the loop involved with *B* and *C* cannot be represented because the loop body crosses both branches coming out of *A*. This type of loop is called an unstructured loop.

Definition 1 An unstructured loop is a loop that has more than one entry. For example, the loop *BC* has two entries *B* and *C*.

Definition 2 Node *N* dominates node *M* if every path from the initial node of the flow graph to *M* goes through *N* (Aho *et al.*, 1986).

Definition 3 A flow graph is irreducible if and only if it contains an unstructured loop of which one entry does not dominate all other entries.

The flow graph of Figure 3 is an irreducible graph because neither *B* nor *C* dominates the other. We refer to this type of unstructured loop as an irreducible loop. An unstructured loop that does not cause irreducibility is a *natural loop* (*i.e.*, a loop that has a single dominating entry (Aho *et al.*, 1986)). The loop *BD* is a natural loop where *B* is the dominating entry (or the header). All the loops presented in a reducible graph are natural loops.

We consider this new definition of irreducible graphs suitable in our problem context. It allows us to identify the irreducibility of a graph by just looking at the graph without going through complex algorithms such as T1–T2 reduction or interval analysis. See Aho *et al.* (1986) for more discussion on irreducible loops.

A natural loop can be translated into structured statements as demonstrated in the example. However, the common techniques (node splitting, Cocke and Miller, 1969) for converting an irreducible graph to a reducible graph require at least 2^{n-1} state duplication where *n* is the number of nodes in the original irreducible graph (Carter *et al.*, 2003). This becomes unmanageable for even a relatively small business process. Therefore, a method is needed to translate the irreducible graph directly.

2.2 Problem with Directed Acyclic Graphs

Even if the process model is cycle free, the naïve transformation method still has problems. Suppose we delete the edges *c*, *e* and *g* in Figure 3, then the remaining graph is a Directed Acyclic Graph (DAG). We may translate the remaining graph to the following code. We notice that the bold-face code has been repeated.

```

invoke A
switch
case a, invoke B
      if f, invoke D
        if h, invoke E
case b, invoke C
      if d, invoke B
        if f, invoke D
          if h, invoke E

```

After the analysis of the problems, we conclude that the naïve method only performs well for a tree-structured process model. Next, we will discuss REL on compiling a non-concurrent business process into BPEL code. The algorithm is customised to support the compilation of concurrent flows as well. However, this is out of the scope of this paper. With REL we can solve the problems discussed in Section 2.

3 The REL algorithm

One notable difference of a business process compiler from a traditional programming language compiler is that this compiler is a linearisation tool that translates a graphic model in two-dimensional space to a well structured textual language in one-dimensional space. By ‘one-dimensional’, we mean that a structured textual language defines a sequence of statement execution. By linearising¹ a graph, we translate the logic of how each state is visited in a graph to a linear sequence of state visiting. Our approach transforms the graphic model first to an intermediate representation that is textual and theoretically equivalent to the source graphical model. We use the Regular Expression (RE) for this purpose. RE has structured control flow constructs: concatenation, or and star. The alphabet set Σ of the RE in our context is the set of all edge-IDs in the process graph.

Traditionally, an RE is treated as a definition of a language (an instance of the regular language). We place a slightly different view: an RE sentence is a program written in a Regular Expression Language (REL). Since a grammar can be defined for REL, the well established compiler techniques (syntax directed translation) can be leveraged to translate this intermediate representation to the target language. Therefore, the compilation consists of two steps:

- 1 the translation from FA to RE
- 2 the compilation from REL to BPEL.

The first step obtains a logically correct linear representation of the source model. The second step brings the intermediate logic representation to a correct format.

Our compilation method overall is thus termed REL. The REL can also be easily customised and extended to support specific features of our compilation system such as compiling concurrent processes.

Section 3.1 gives the details on how to convert an FA to an RE.

3.1 Linearisation of finite automata

The set equation algorithm (Denning *et al.*, 1978) is used to convert an FA to an RE. In the process of solving the equations, the strategies and optimisations are used to ensure the resulting RE is optimal. Section 3.1.1 discusses how the equations are extracted from an FA and how they are solved. Detailed explanation and reasoning on RE optimisations and the strategies are presented in Sections 3.1.2 and 3.1.3, respectively. Section 3.1.4 summarises how the problems we introduced in Section 2 are solved.

3.1.1 Extract and solve set equations

Definition 4 Let M be an FA, the end set $E(q)$ of a state q of M is the collection of input strings that can lead from q to an accepting state of M (Denning et al., 1978).

For example, the end set $E(A)$ of Figure 3 is the set of strings that lead the FA from state A to the accepting state E . A string in $E(A)$ consists of an ‘a’ followed by the end set $E(B)$ or a ‘b’ followed by the end set $E(C)$. Thus, the definition of $E(A)$ can be written as follows:

$$E(A) = a E(B) + b E(C)$$

The ‘+’ sign means ‘XOR’. For the readability, the previous equation is simplified to the following:

$$A = aB + bC. \tag{1}$$

Similarly, we can define the end set for each state as follows:

$$B = cC + fD \tag{2}$$

$$C = eC + dB \tag{3}$$

$$D = gB + hE. \tag{4}$$

Since E is the final state:

$$E = \varepsilon. \tag{5}$$

It is easy to see the language recognised by the FA is precisely those strings in the end set of FA’s initial state. That is, $L(\text{FA}) = E(A)$. The Equations (1) through (5) comprise a system of right-linear set equations.

Proposition 1 The end sets of a finite automata $M = (Q, \Sigma, \delta, q_0, F)$ satisfy a system of right-linear set equations: for each $q \in Q$:

$$E(q) = \sum_{q' \in Q} V(q, q')E(q') + W(q)$$

where:

$$V(q, q') = \{x \in \Sigma \mid M \text{ has } q \xrightarrow{x} q'\}$$

and

$$W(q) = \begin{cases} \varepsilon & \text{if } q \in F \\ \emptyset & \text{otherwise} \end{cases}.$$

We will be particularly interested in the solution of A , i.e., $E(A)$, that is the regular expression for the FA.

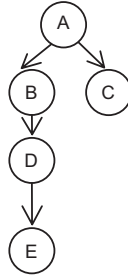
There are three types of rules to solve the set of Equations (1) through (5).

- 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, $C = eC + dB = e^* dB$. Please refer to Denning *et al.* (1978) for the proof of Arden's rule.

Given a set of equations, there exist multiple syntactically different but equivalent REs as the solution for the start state. Since the RE is an intermediate step through which we generate the target code, the complexity of the RE logic directly reflects the complexity of the target code. An effective way we can generate an optimal RE from an FA is to apply the strategy during the process of equation solving. There are two strategies:

- 1 Arden's rule is applied only before the node is going to be substituted.
- 2 There is a specific order based on which the nodes are substituted. The order is from bottom up in a dominator tree (Aho *et al.*, 1986) of a graph. Node *m* dominates node *n* if and only if *m* is an ancestor of *n* in the dominator tree. The dominator tree of Figure 3 is shown in Figure 4.

Figure 4 The dominator tree for the graph of Figure 3



In Section 3.1.3, we will show how the strategies are formulated and why they are important. Based on the strategies aforementioned, we solve the set of equations as follows:

We leave *E* out of consideration at the moment. Thus, *D* is at the bottom of the dominator tree and should be selected first. After *D* is substituted, the rest of the equations become as follows:

$$A = aB + bC$$

$$B = cC + f(gB + hE)$$

$$C = eC + dB.$$

Next, C should be substituted before B (the reason is explained in Section 3.1.3). Before C can be eliminated, Arden's rule should be applied to C to remove the recursion: $C = eC + dB = e^* dB$. After the substitution of C , A and B become as follows:

$$\begin{aligned} A &= aB + be^* dB = (a + be^* d) B \\ B &= ce^* dB + f(gB + hE) = ce^* dB + fgB + f hE = (ce^* d + fg) B + f hE \\ &= (ce^* d + fg)^* f hE \end{aligned}$$

Finally, after B is eliminated:

$$A = (a + be^* d) (ce^* d + fg)^* fhE .$$

E is the final state in Figure 3. It should be first chosen and substituted by nothing. However, in the complete process graph, E is only a place holder for the RE of the rest of the graph. The RE for Figure 3 is as follows: $(a + be^* d)(ce^* d + fg)^* fh$.

3.1.2 Discussion on RE optimisations

Besides the semantic equivalence to the input model, the efficiency of the generated code is the most important consideration for every compiler. By taking an intermediate step, we apply the optimisation of logic complexity to the RE level. The shorter the RE, the less number of loops, the less number of 'or' constructs, the more optimal the RE is. The strategies and the optimisation rules guarantee the optimal RE to be generated from an FA. Please note that all FAs in our problem domain are deterministic, therefore, all the REs in the solution set are deterministic. By 'optimal', we mean the optimal RE in the solution set. There might exist a more optimal RE than the 'optimal RE' in the solution set. For example, a non-deterministic RE is always simpler than the REs solved from the equivalent deterministic FA. A non-deterministic RE is an RE from which only a non-deterministic FA can be built directly, for example: $(a + b)^* abc$ with $\Sigma = \{a, b, c\}$. It will be an interesting research topic to find out the optimal non-deterministic RE (if it exists) for a given deterministic FA. However, it is not the concern of this paper.

There are two types of optimisations:

- 1 general RE optimisations
- 2 loop look-ahead common sub-expression elimination (short for loop exit optimisation).

General RE optimisations include common sub-expression elimination and loop normalisations. Common sub-expression elimination can be achieved by the anti-distributive law. A normalised loop after loop normalisation is a loop in the form of the following:

$$\left(\sum_{x \in R} x \right)^*$$

where R is a regular expression. Some sample normalisation rules are as follows ($u, v, w \in R$):

- $(v^*)^* \Rightarrow v^*$
- $(v^* + w^*)^* \Rightarrow (v + w)^*$
- $(v^* w)^* v^* \Rightarrow (v + w)^*$

- $(v^* w^*)^* \Rightarrow (v + w)^*$
- $v^*(w v^*)^* \Rightarrow (v + w)^*$
- $(v^* u^* + w)^* \Rightarrow (v + u + w)^*$

Loop exit optimisation is necessary because of the natural differences of loops between structured languages and FAs. In RE and BPEL, there is only one point where the loop enters and exits. That point is called the *home state* of the loop. Some other structured languages allow loops to have multiple exits, for example, multiple ‘break’ statements can be used in a single ‘while’ statement in Java. A loop with multiple exits can be easily simulated by a loop with a single exit by introducing an additional conditional variable. On the other hand, arbitrary loops in an FA can have multiple entry points and multiple exit points. The entries and the exits are not necessarily the same. The problem of multiple entries will be discussed in Section 3.1.3. The loop exit optimisation enables correct translation of loop exits from FA to RE and BPEL.

To see an example, the loop involved with B and D in Figure 3 has two exits: B and D . When this loop is directly represented in the RE shown in Section 3.1.1, the path inside the star operator starts and ends at the home state B (the node where Arden’s rule is applied). The alternative exit is represented as ‘fh’ following the star operator. Multiple alternative exiting paths would be represented by an or operator following the star. However, the path following the star has an overlap ‘f’, which is called the loop look-ahead common sub-expression, with the body of the loop. The overlap results in the non-determinism at the node B . The loop exit optimisation rule deletes the loop look-ahead common sub-expression in the strings that follow the star. A new syntactical marker ‘[]’ is used to explicitly mark the loop exits (the letter that immediately follows the deleted string) inside the body of the loop. We thus get the following RE:

$$(a + be^* d)(ce^* d + f[h]g)^* h.$$

In the definition of REL, ‘[h]’ indicates a special type of literal called loop exit. The loop exit optimisation fails only with the non-deterministic RE because, in that case, the exit indicator could be the same as the looping condition. For example, the letter ‘a’ in $(a + b)^* abc$ is both the loop exit and the looping condition. In other words, two outgoing edges from one node are both named ‘a’. Luckily, since all edges are given a distinct ID in the FA in our problem context, that is, all outgoing edges for any node are marked distinctly, no non-deterministic RE will be generated from the equation solver.

3.1.3 Discussion on equation solving strategies

We now discuss how the two strategies lead us to the optimal RE. Strategy 1 guarantees the minimum number of loops that resulted from the equation solver. The number of loops in the final RE is equal to the number of applications of Arden’s rule. If Arden’s rule is applied not immediately before the node is to be substituted, then there is always a chance that it will be applied again at the same node. For instance, if Arden’s rule is applied to B after D and before C are substituted, the set of equations become as follows:

$$\begin{aligned} A &= aB + bC \\ B &= (fg)^* (cC + fhE) \\ C &= eC + dB = e^* dB. \end{aligned}$$

After C is substituted and Arden's rule is applied to B again, we get the following:

$$A = (a + be^*d) B$$

$$B = ((fg)^* ce^*d)^* (fg)^* fhE.$$

As can be seen, we obtain two extra loops in B . Although based on the loop normalisation Rule 3, $(fg)^*ce^*d)^*(fg)^*$ is equivalent to $(fg + ce^*d)^*$, it is better to get the simplest RE in the first round.

Strategy 2 is the most important factor that controls the complexity of the RE solution. To explain this strategy, we consider the following three cases.

Case 1 The flow graph is a DAG. The RE solution will be optimal disregarding the substitution order.

Proof. Because there is no loop in the graph, no nodes can be repeated on a path from the start node to the end node. On the other hand, an RE is a way to denote all possible ways to traverse a graph. Thence, each node in a graph has to be visited as least once. Therefore, a path without node repetition is considered optimal. Multiple paths that share some nodes can be optimised by the RE anti-distribution law. Furthermore, because there is no cycle in the graph, Arden's rule is not applicable during equation solving. Among all the equation solving rules, only Arden's rule is affected by the substitution order in producing solutions of different complexity. This will be explained more in Cases 2 and 3. The RE algebraic laws do not distinguish the unknowns from their values and are therefore independent from the substitution order.

Case 2 The flow graph is a reducible graph and thus contains only natural loops. If and only if the header is substituted after the loop body, the RE solution is optimal.

Proof. Because of the nature of a loop, any arbitrary node on the loop can be treated as the start (the home state) of the loop. Any node substituted after all other nodes on the loop gets a recursion in its equation and therefore requires the application of Arden's rule. Whenever Arden's rule is applied to a node, that node becomes the home state of the loop.

In Figure 5(a), three arbitrary nodes on a natural loop are denoted as A , B and C . A is the header. The solid arrows starting from the loop show three arbitrary exits. If A is the home state, we get the following unique RE:

$$|QA| (|ABCA|)^*(a + |AB| b + |ABC| c).$$

The vertical bar denotes the path among a set of nodes. After the loop exit optimisation, the above RE becomes as follows:

$$|QA| (|AB [b] C [c] A|)^* (a + b + c).$$

The exit points are indicated explicitly along the path of the loop. As can be seen, each node in this graph is visited only once except the indication of the exits. Therefore, we can claim that the above RE is the most optimal one for this graph.

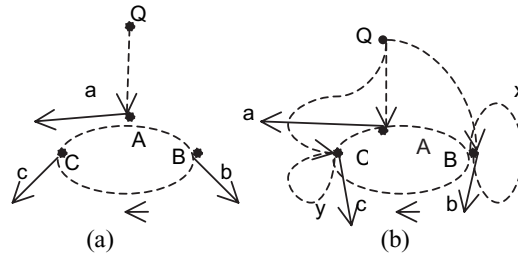
However, if B is the home state, we will get the following string:

$$|QA||AB|(|BC [c] A [a] B|)^* (b + c + a).$$

It is clear the path $|AB|$ is repeated before and in the star. The deeper B is below A , the bigger the repetition is. This shows that we get the optimal RE only if the loop header is the home state.

If a graph contains multiple natural loops, they can disjoint, nest within another or share the same loop head (Aho *et al.*, 1986). In either case, the analysis of a single loop remains valid. For the loops that share the same loop head, the generated RE star loops are in the normal form.

Figure 5 The natural loop (a) and the irreducible loop (b)



Note: Dotted line: a path evolved with multiple nodes. Solid line: a path between two nodes. Q is the common immediate dominator for the entry (entries) of the loop. The dotted circles are loops. The small arrow under a circle denotes the rotation direction of the loop.

Case 3 The process graph is irreducible. If and only if the entry node that has the largest sum of end set and pre-set (if it is also the exit of the loop) is the home state of the loop, the RE solution is optimal.

We introduce two definitions before we present the proof.

Definition 5 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 the CID. The CID of an ILR is called the header of the ILR. We can see that some nodes in an ILR are not on the loop.

Definition 6 The pre-set of a node in an ILR is the path from the CID to that node without going through the home state of the loop.

Proof. Figure 5(b) is an example of the generalised ILR. The loop ABCA is an irreducible loop. A , B and C are the entries to the loop. Loop x is a natural loop led by B . Loop y is a natural loop led by C . And a , b and c are exits at the entry points. Exits that are not also the entries of the loop should be handled the same way as in a natural loop (they are ignored in this case). Suppose B is the home state, the set equation algorithm generates the following pattern for this ILR:

$$\begin{aligned} & ((|QC| |y^*| CA| + |QA|) |AB| + |QB|) \\ & (x + |BC| |y^*| [c] |CA| [a] |AB|)^* (b + c + a) + \\ & (|QC| |y^*| c + (|QC| |y^*| CA| + |QA|) a). \end{aligned}$$

The first underlined portion is how B is reached from Q . The second underlined part is the loop itself. And the third underlined part is how the loop can be exited without going through B . The repeated portion is $y^*|CA||AB|$ in the first part and the whole string in the third underlined part. If we reorganise the repeated code, we see that $|AB| + a$ is the end set of A ; $|CA| + y^* + c$ is the end set of C ; $(|QC|y^*|CA| + |QA|)$ is the pre-set of A ; $|QC|y^*$ is the pre-set of C . From the third underlined part, we know the following: if an entry node is not also the exit of the loop, its pre-set will not cause the repetition. Therefore, the repetition of the generated code for an ILR is the end sets of the entry nodes that are not the home state, and the pre-sets of the entry nodes that are also the exits and are not the home state of the loop. Thus, if and only if the entry node that has the largest sum of end set and pre-set (if it is also the exit of the loop) is the home state of the loop, then the RE solution is optimal.

To give an example, suppose we solve the equation of Figure 3 in a different order so that B is substituted earlier than C , then C becomes the home state. The resulting RE is as follows:

$$(a(fg)^*c + b)(e + d(f[h]g)^*c)^*h + a(f[h]g)^*h.$$

This RE is more complex than that we obtained previously. From Figure 3, the pre-set of B is $a(fg)^*$. The pre-set of C does not count since C is not the exit of the irreducible loop. The end set of B is larger than that of C (the next several paragraphs explain how the size of end set can be determined). Hence, choosing C instead of B as the home state results in a larger RE.

This strategy attempts to find the best solution. However, when each entry node is weighted the same, there will be no optimal solution because each solution will be equally complex. For instance, all REs for a strongly connected component are equally complex.

Knowing the correct order of the substitution, we now discuss how this order is computed. If a graph is reducible, a depth-first ordering is sufficient since the dominating entry is always numbered lower than its dominated loop body (Aho *et al.*, 1986). However, if the graph is not reducible, one pass of depth-first search is not enough to determine the correct order of substitution. This is so because depth-first search does not guarantee the ordering for nodes that do not dominate each other. For example, depth-first search could give the ordering for Figure 3 as ABDEC (numbered from one to five). As C does not dominate BDE, so C could also be positioned before B , or after B and before DE.

Since we do not know the reducibility of the graph before we start to solve the set of equations, we use a uniform way to calculate the substitution order for both cases. First, a depth-first ordering of the nodes is computed followed by the algorithm presented in (Cooper *et al.*, 2001) for finding the dominator set for each node. For efficiency, the finding dominator algorithm has to be built on top of depth-first ordering. After the dominator set for each node is determined, a dominator tree may be constructed.

If the graph is reducible, substituting the nodes bottom-up based on a dominator tree guarantees the loop entry is substituted after the loop body. The bottom-up order ensures that the end set of the node that is to be substituted at each step contains only literals except the dominating loop entry (or entries in case of irreducible graph). This gives us the benefit that RE optimisations can be performed at each step to prevent the strings from getting too complex.

The irreducibility of the graph can be detected automatically when the equations are solved bottom-up based on a dominator tree. At the time the nodes at the same layer of the tree are to be substituted, if the end sets of the nodes do not contain each other, the graph is reducible. Hence, the nodes at the same layer can be substituted in any order. Otherwise, this indicates an irreducible loop, and we need to use the strategy laid out in Case 3. At that particular situation, the size of the end set can be determined simply by measuring the length of the equation of the corresponding node because the equation would contain only literals except the other loop entries. For example, in the example of Section 3.1.1, the end set of B is larger than that of C after D has been substituted. The pre-set of an entry node can be computed by counting the hops from the CID, which is this node's direct parent in the dominator tree, to that node.

3.1.4 Summary on how the problems with DAG and unstructured loop are solved

The removal of code repetition in a DAG can be achieved by the removal of common sub-expressions using the anti-distributive law. For the FA in Figure 3 with edges c , e and g removed, the regular expression is $afh + bdfh$, which is optimised to $(a + bd)fh$.

Regarding a loop, there are four missions:

- 1 how to detect a loop
- 2 how to scope a loop
- 3 where the entries are
- 4 where the exits are.

The first two missions are automatically solved wherever Arden's rule is applied. The algorithm for calculating the substitution order selects a right loop entry. The loop exit optimisation can detect loop exits and mark the exits explicitly inside the body of the loop.

Table 1 The mapping of syntactical constructs between REL and BPEL

<i>Construct</i>	<i>Sequential</i>	<i>Alternative</i>	<i>Loop</i>	<i>Action node</i>	<i>Flow control</i>
<i>Language</i>					
RE	Concatenation	+	*	State	Input symbol
BPEL	Sequence	Switch case	while	Web service invocation	Conditions events data

3.2 Compilation from REL to BPEL

The mapping from REL to BPEL is straightforward. Table 1 shows the mapping from REL constructs to BPEL constructs. The pseudo BPEL code for the RE of Figure 3 is shown in Figure 6. The 'if' statements in the pseudo code will be changed to conditions based on the evaluation of BPEL variables. It can also be deleted if there is no condition test and no event or data passed on that edge. The state information can be obtained by looking up the edge hash table.

Figure 6 The pseudo BPEL code for Figure 3.

```

invoke A
switch
  case a: invoke B
  case b: invoke C
    while e
      invoke C
      if d, invoke B
thisExit=false
while ((c or f) and not thisExit)
{
  switch
    case c: invoke C
      while e
        invoke C
        if d, invoke B
    case f: invoke D
      if h, thisExit=true.
      if g, invoke B
}
if h, invoke E.
    
```

Note: The RE of Figure 3 is as follows: $(a + be * d)(ce * d + f[h] g) * h$.

Figure 7 The attribute grammar of REL for pseudo code generation

```

R0 ::= (R1)
{ :R0.code=R1.code;
  R0.condition=R1.condition; ;}

|R1R2
{ :R0.code=R1.code || R2.code;
  R0.condition=R1.condition; ;}

|R1+R2
{ :R0.code= "switch" || "case:" || R1.code || "case:" || R2.code;
  R0.condition=R1.condition || "or" || R2.condition; ;}

|R1*
{ :R0.code= loops[i] || "Exit=false" || "while" || R1.condition || "and not"
  || loops[i] || "Exit" || R1.code;
  R0.condition=R1.condition;
  i++; ;}

|input
{ :R.code= "if" || input.lexeme || "invoke" || input.nextState;
  R.condition=input.lexeme; ;}

|[exit]
{ :R.code="if" || exit.lexeme || loops[i] || "Exit=true"; ;}
    
```

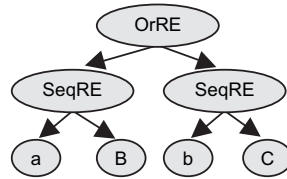
Syntax-directed translation is used to generate the code of Figure 6. The attribute grammar specification for this translation is shown in Figure 7. The lexical symbols of REL shown in bold-face are the following: '()', '[]', '*' and '+'. During the translation, code and looping-condition are synthesised attributes. The evaluations of the two attributes are shown as the semantic actions enclosed by '{: :}'. For simplicity, only the synthesis of pseudo code is shown. The '||' means the concatenation of code fragments. The syntax of the grammar is in BNF.

A global array variable 'loops' is used to ensure that the loop exit information is bound to a correct scope. The loops[i] means the current loop.

4 Experimentation

The algorithm has been completely implemented in Java. The mathematical notations of RE are implemented as an object-oriented tree structure. For example, the object-oriented syntax tree of A's equation is shown in Figure 8. BPEL code generation is achieved by using the visitor pattern (Gamma *et al.*, 1995) to walk through the RE syntax tree.

Figure 8 Equation for $A = Ab + bC$



The first part of the REL algorithm, converting FA to RE, is presented in Figure 9.

Figure 9 The algorithm for converting FA to REL

```

Input: graph G with n nodes and m edges.
Output: the optimal RE that describes the same process.
Algorithm:
DepthFirstOrderOfG  $\leftarrow$  depthFirstSearch (G);
calculateDominatorSetForEachNode (DepthFirstOrderOfG);
extractEquationsForEachNodeAndBuildDominatorTree;
Stack  $\leftarrow$  breadthFirstSearchOfDominatorTree;
while (Stack not empty) {
  Top  $\leftarrow$  Stack.firstElement();
  TopLevel  $\leftarrow$  Top.getDepthInDominatorTree();
  for (all nodes on TopLevel) {
    calculateWeightsOfNodes;
  }
  reorderNodesOnTheStackBasedOnWeights;
  for (all nodes on TopLevel) {
    substitute (Stack.pop() );
  }
}
  
```

Among all these operations, substitution is the most expensive. It takes up 80% of the computation in this algorithm. Other equation solving rules will be invoked when needed by the substitution operation. For example, Arden's rule will be used before the substitution can take place if the equation is self recursive. Rules such as distribution and anti-distribution are necessary preparation in order to apply substitution for some equations. Loop exit optimisation and loop normalisation are performed after Arden's rule. During equation solving, no matter what rules are used, all equations should keep a very important mathematical invariant (right linear). However, this invariant will break in the practical implementation data structure after some manipulation of the equations. Therefore, some other normalisation rules are also used after the application of each rule. This is done to keep the implementation data structure of the equation conforming to the mathematical properties. Two examples of these normalisation rules are as follows: if a SequenceRE₁ has a SequenceRE₂ as one of SequenceRE₁'s child, lift all SequenceRE₂'s children up as children of SequenceRE₁; if a SequenceRE has a single child, replace the SequenceRE with its child.

In terms of time complexity, depthFirstSearch takes $O(n)$ time. The calculateDominatorSetForEachNode is based on the iterative dominator algorithm described in (Cooper *et al.*, 2001). Kam and Ullman (1976) proved that an iterative algorithm traversing the graph in depth first order will halt in no more than $d(G) + 3$ passes of G , where $d(G)$ is the depth of Knuth (1971) showed that average flow graphs have a depth of 2.75. Therefore, calculateDominatorSetForEachNode takes $O(n)$ on average and $O(m * n)$ in the worst case. In addition, we have applied some implementation tricks to improve the performance of this operation, such as the following: ordered vector rather than the set is used as the data structure; set union is implemented by appending to the end of the vector; set intersection is implemented as pair wise comparison of two vectors because the order of nodes in two vectors are always consistent; and the set copy is simply an assignment of object reference as we use Java as our implementation language. The extractEquationsForEachNode-AndBuildDominatorTree takes $O(n)$. In the while loop, nodes will be processed layer by layer bottom-up in the dominator tree. The substitute() will be performed exactly $O(n)$ times. Since substitute() is the core operation of the algorithm in Figure 9, the algorithm overall takes $O(n)$.

Table 2 Experimental statistics for the REL algorithm

<i>Number of nodes</i>	<i>Number of edges</i>	<i>Time (in milliseconds)</i>	<i>Output BPEL size (in lines)</i>
3	9	80	99
5	8	60	46
6	11	100	102
14	16	120	62
40	106	770	1102
50	75	580	423
100	134	980	810

For the experimentation, we start by using a freely available UML tool ‘Poseidon for UML’² to draw UML activity diagrams. Our algorithm takes the XMI (XML Metadata Interchange³) generated by Poseidon as the input. In order to experiment with large business processes, we constructed a random XMI file generator. We ran our algorithm on a laptop with 1.53 GHz AMD processor, 480MB of RAM and Microsoft Windows XP operation system. Table 2 shows the performance of the algorithm and the output BPEL size over a set of hand-crafted and randomly generated processes.

The largest benchmark in our system has 100 nodes. We do not anticipate that any realistic business processes would exceed that limit. A business process that is too large should be modularised into sub-processes. Each sub-process will be transformed into a separate BPEL process. Both the run-time duration of the algorithm and the output BPEL size depends on the number of nodes and the number of edges of the graph. The factor of the irreducible loop is hidden from this table. The time and output are linear to the number of nodes and edges if the graph does not contain irreducible loops, for example, the graph of nodes 5, 14 and 50 are of this category. The more ILRs a graph contains, the longer the run-time duration is and the bigger the output size is. In the worst case (strongly connected component), the output size is exponential to the number of nodes. In our experiment, the first test case is a strongly connected component with three nodes. For the graphs that contain some ILRs, but are not strongly connected components, the REL algorithm generates BPEL of an optimal size.

5 Related work

There are different ways an FA can be translated into a language that has structured control flow statements.

It has been proved by (Bohm and Jacopini, 1966) and also shown in (Hauser and Koehler, 2004) that every state machine can be translated into a program with a single while and a single switch statement. Let us call this the state machine controller approach. Based on this approach, the FA in Figure 3 can be translated into the following code:

```

nextnode=A
while nextnode !=E
    switch
    case nextnode =A
        invoke A
        if a, nextnode =B
        if b, nextnode =C
    case nextnode =B
        invoke B
        if f, nextnode =D
    .....

```

The drawback of this approach is run-time inefficiency for complex 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, which gives

poor code comprehensibility. Furthermore, this approach encodes the goto, which is a low-level language concept, in high-level language constructs.

Graph reduction (Koehler *et al.*, 2005) has been used to translate a reducible flow graph to structured code. Code is output after each step of T1–T2 reduction until the graph has been reduced into a single node. However, this method leaves the problem of irreducible graphs open. The style of the code generated from this method is of the compact form:

```

repeat
  invoke A
  if e, invoke D
  if ( eg or c ), invoke C
  while ( a or ef or cd or egd )
  if ( b or ei or ch or egh ) , invoke B
    
```

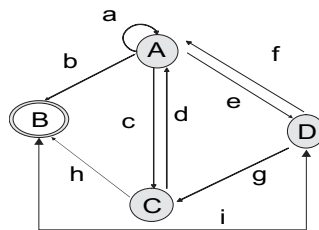
The code of this style, without additional augmentation such as concurrency in implementation, highly depends on the order of the statements. For example, in this code fragment:

```

if ( eg or c ), invoke C
if e, invoke D
    
```

the first statement cannot be evaluated since the condition *g* is produced by activity *D*. We refer to the code of this style as the code that does not preserve the ‘look-ahead = 1’ semantics of FA. This is the semantics that ensures an FA makes one transition by reading in only one input symbol. Meanwhile, the long string of the conditions gives poor compensability of the code. Just for comparison, the code generated for this reducible graph using the method presented in this paper is shown in the Appendix.

Figure 10 The FA for the product selection portion of Figure 2



Note: A: select; B: merge; C: put; D: configure

The algorithms based on goto eliminations (Hauser and Koehler, 2004) and continuation semantics (Ammarguella, 1992; Koehler and Hauser, 2004) to untangle the unstructured loops have a very similar set of rules as the set equation algorithm. The main difference is that these approaches do not guarantee the optimality of the generated code for the irreducible process graph. These algorithms also use an intermediate representation similar to pseudo code. By using RE, we systematically factor out the structure information that is only relevant to the algorithm. Another important difference is that these algorithms work on the states, whereas the method in this paper work on the edges. Consequently, the code generated by Ammarguella (1992), Koehler and Hauser (2004),

Hauser and Koehler (2004) is in a similar style of that resulted from graph reduction. On the other hand, our method introduces state repetitions in the code as the length of the generated code is bound to edges rather than states.

RE has been used as an intermediate step through which a program with goto statements can be transformed into programs without gotos (Morris *et al.*, 1999). However, the method uses a uniform RE ' $E_1 * E_2$ ' to represent the program, where E_1 are the elementary nontrivial paths back to the start node and E_2 are 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 did not pay attention to the optimisations. The resulting RE is usually unnecessarily complex. Compared to other algorithms such as Hopcroft and Ullman (1979) to convert an FA to an RE, the set equations algorithm gives us the opportunity for optimisations. It is also quite straightforward and easy to implement.

Although model transformation is quite a popular topic today, there is much work to be done on translating UML activity diagrams or similar behaviour models (Patrascioiu, 2004; Skogan *et al.*, 2004) to structured languages that does not consider the transformation for loops at all.

6 Conclusion and outlook

In this paper, a novel algorithm is presented for compiling a business process model with irreducibility to a structured flow language. The complexity of the code is controlled by using different levels of optimisations. Strategies are used during equation solving to ensure that the optimal solution. The generated RE can also be optimised using general RE optimisation rules. The benefits of our approach is that the logic related optimisations are at the RE layer. This makes the optimisations more efficient and easier to understand and to implement. The generated target code can also be optimised for platform specific concerns although we do not discuss this aspect in the paper.

Taking the RE as an intermediate step enables the reuse of core implementation of the algorithm to different source models and target languages. This supports the vision of mapping a PIM to multiple PSMs. The compilation framework discussed in this paper by taking an intermediate step could be a good practice in implementing a compiler from a graphic model to a textual language in general. Having an intermediate representation is a general solution in traditional compiler construction as well.

There are some limitations in our approach. The algorithm assumes the transition semantics in the model are XOR and AND (in the case of concurrency). The algorithm fails for transitions with OR semantics. Fortunately, OR semantics is not a standard that is supported by all the business modelling tools.⁴ According to (Wohed *et al.*, 2002), OR transitions in BPEL can be represented by links in the style of Web Service Flow Language (WSFL). Such links are not structured constructs but straightforward encodings of the model. Thus, translating the model to this representation is outside our problem domain. Patterns from workflowpatterns.com, which are realised in BPEL by 'link', 'pick' and 'serialisable scope' but are not considered in our algorithm, are the following: Multiple Choice, Implicit Termination, Interleaved Parallel Routing and Milestone.

Under the current implementation of BPEL, although a BPEL program invokes services distributed over several servers, the orchestration of these services is centralised on one server. The orchestration is written in structured languages. On the other hand, the execution control of an FA-based model is distributed in nature considering that each state stands for a server hosting a web service. We encountered the problem of the unstructured loop because we translate the distributed to centralised execution control. Efforts are taking place at IBM to decentralise BPEL (Nanda *et al.*, 2004). This work partitions a centralised BPEL program into decentralised processes. If in the future BPEL becomes decentralised, there are two implications:

- 1 The unstructured loop problem will go away in the translation from UML activity diagrams to BPEL.
- 2 The work of Nanda *et al.* (2004) is not really needed because the centralised BPEL might go away as well and decentralised BPEL comes directly from the process model.

Nevertheless, the algorithm presented in this paper still applies to translation from a business process model to any other structured language if it is not BPEL.

Independent of the model transformation domain, this paper also contributes to the set equation algorithm by introducing equation-solving strategies that give the optimal solution. Meanwhile, it is still a new strategy that the irreducibility is solved by using set equation algorithm and that RE as traditionally the problem of irreducibility is solved by node-splitting techniques.

To fully justify the usefulness of this work, we also need to answer these questions: Why bother to translate unstructured flows into structured representations? Why not design the modelling language to have the structured loops? In fact, UML2 (2003, pp.341–342) does provide a looping construct, although the specification is incomplete. Based on our customer engagement, we found out it is difficult for the business level user to comfortably use such a loop construct. It is rather easier for them to understand the concept of goto and self-loop (a single node loop). Therefore, business level users frequently use tools such as UML to construct unstructured ‘goto’ flows. It is important that we have an algorithm to translate such unstructured flows to structured representations if we choose a structured representation as the executable PSM.

Acknowledgements

The authors thank Pranam Kolari of the University of Maryland, Baltimore County, for his participation in the early development of this work. They also thank Brent Hailpern and Tim Klinger of IBM T.J. Watson Research, and Jana Koehler and Jochen Kuster of IBM Zurich Research for reviewing and editing of the paper. Thanks are also extended to Kumar Bhaskaran of IBM T.J. Watson Research for his valuable comments and support.

References

- Aho, A., Sethi, R. and Ullman, J. (1986) *Compilers-Principles, Techniques, and Tools*, Addison-Wesley.
- Ammarguellat, Z. (1992) 'A control-flow normalization algorithm and its complexity', *IEEE Transactions on Software Engineering*, Vol. 18, No. 3.
- Bohm, C. and Jacopini, G. (1966) 'Flow diagrams, turing machines and languages with only two formation rules', *Communications of the ACM*, Vol. 9, No. 5, pp.366–371.
- Business Process Execution Language (BPEL) (2003) *Business Process Execution Language for Web Services*, Version 1.1, 05 May, <http://www-106.ibm.com/developerworks/library/ws-bpel/>
- Carter, L., Ferrante, J. and Thomborson, C. (2003) 'Folklore confirmed: reducible flow graphs are exponentially larger', *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.106–114.
- Cocke, J. and Miller, R.E. (1969) 'Some analysis techniques for optimizing computer programs', *Proceedings of the 2nd Hawaii International Conference on Systems Sciences (HICSS)*, pp.143–146.
- Cooper, K., Harvey, T. and Kennedy, K. (2001) 'A simple, fast dominance algorithm', *Software – Practice and Experience*, Vol. 31, No. 4, pp.1–10.
- Denning, P.J., Dennis, J.B. and Qualitz, J.E. (1978) *Machines, Languages, and Computation*, Prentice-Hall, Inc.
- Frankel, D.S. (2003) *Model Driven Architecture: Applying MDA to Enterprise Computing*, Wiley Publishing, Inc.
- Frank, J.H., Gardner, T.A., Johnston, S.K., White, S.A. and Iyengar, S. (2004) 'Business processes definition metamodel concepts and overview', *IBM's Proposal in Response to the OMG's RFP for a Business Process Definition Metamodel*, <http://www.bpmn.org/Documents/BPDM/BPDM%20Whitepaper%202004-05-03.pdf>
- Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.
- Hauser, R. and Koehler, J. (2004) 'Compiling process graphs into executable code', *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE'04)*, pp.317–336.
- Hopcroft, J. and Ullman, J. (1979) *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley Publishing Company, Inc.
- Kam, J. and Ullman, J. (1976) 'Global data flow analysis and iterative algorithms', *Journal of the ACM*, Vol. 23, No. 1, pp.158–171.
- Knuth, E. (1971) 'An empirical study of FORTRAN programs', *Software – Practice and Experience*, Vol. 1, No. 2, pp.105–133.
- Koehler, J. and Hauser, R. (2004) 'Untangling unstructured cyclic flows – a solution based on continuations', *Proceedings of the International Conference on Cooperative Information Systems*, pp.121–138.
- Koehler, J., Hauser, R., Sendall, S. and Wahler, M. (2005) 'Declarative techniques for model-driven business process integration', *IBM Systems Journal*, Vol. 44, No. 1, pp.47–65.
- Morris, P.H., Gray, R.A. and Filman, R.E. (1999) 'GOTO removal based on regular expressions', *Journal of Software Maintenance: Research and Practice*, Vol. 9, No. 1, pp.47–66.
- Nanda, M.G., Chandra, S. and Sarkar, V. (2004) 'Decentralized execution of composite web services', *Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.
- Patrascoiu, O. (2004) 'Mapping EDOC to web services using YATL', *Proceedings of the 8th International IEEE Enterprise Distributed Object Computing Conference*.

- Skogan, D., Gronmo, R. and Solheim, I. (2004) 'Web service composition in UML', *Proceedings of the 8th International IEEE Enterprise Distributed Object Computing Conference*.
- UML2 (2003) *UML 2.0 Superstructure Final Adopted Specification*, ptc/03-08-02, <http://www.omg.org/cgi-bin/doc?ptc/2003-08-02>
- Wohed, P., van der Aalst, W.M.P., Dumas, M. and ter Hofstede, A.H.M. (2002) 'Pattern-based analysis of BPEL4WS', *QUT Technical Report*, FIT-TR-2002-04, Queensland University of Technology.

Notes

- 1 We should be careful that a direct XML encoding of a graph is not a linearisation of the graph as the encoding is still semantically two dimensional.
- 2 <http://www.gentleware.com/index.php>
- 3 <http://www.omg.org/technology/documents/formal/xmi.htm>
- 4 <http://tmitwww.tn.tue.nl/research/patterns/standards.htm>

Appendix

Based on the method presented in this paper, the RE generated for Figure 10 is:

$$(a + c[h]d + e[i](f + g [h] d))^* (b + h + i)$$

The pseudo BPEL code is:

```

thisExit=false
while (a or c or e and not thisExit)
  switch
    case a, invoke A
    case c, invoke C
      if h, thisExit =true
      if d, invoke A
    case e, invoke D
      if i, thisExit=true
      switch
        case f, invoke A
        case g, invoke C
          if h, thisExit=true
          if d, invoke A
  
```