Automatic Extraction of Process Control Flow from I/O Operations

Pedro C. Diniz[1] and Diogo R. Ferreira[2]

[1] IST/INESC-ID, Technical University of Lisbon, Portugal
[2] IST/INOV, Technical University of Lisbon, Portugal
{pedro.diniz,diogo.ferreira}@tagus.ist.utl.pt

Abstract. Many end users will expect the output of process mining to be a model they can easily understand. On the other hand, knowing which objects were accessed in each operation can be a valuable input for process discovery. From these two trends it is possible to establish an analogy between process mining and the discovery of program structure. In this paper we present an approach for extracting process control-flow from a trace of read and write operations over a set of objects. The approach is divided in two independent phases. In the first phase, Fourier analysis is used to identify periodic behavior that can be represented with loop constructs. In the second phase, a match-and-merge technique is used to produce a control-flow graph capable of generating the input trace and thus representing the process that generated it. The combination of these techniques provides a structured and compact representation of the unknown process, with very good results in terms of conformance metrics.

Keywords: Process mining, Control-flow graphs, Fourier analysis

1 Introduction

Since the publication of [1], Petri nets became the preferred formal framework for the analysis [2], modeling [3], verification [4], mining [5] and conformance checking [6] of business processes. However, most workflow and Business Process Management (BPM) systems typically make use of proprietary modeling languages [7]. Despite efforts such as the development of workflow patterns [8], a vast community of end users still makes use of informal languages and notations, some of which have their origin in flowcharting [9], and many of which are reminiscent of basic programming concepts such as decision and loop constructs.

There is a number of reasons for the continuing use of such languages. First, it is often the case that the goal is to discuss process models with domain experts rather actually pursuing process analysis or enactment [10]. Second, there is often a perceived notion that process modeling is a kind of programming [11], hence the resemblance between modeling and programming constructs. Third, recent developments in Web service technology and SOA[3] have also contributed

[3] SOA: Service-Oriented Architecture

to shorten the distance between process modeling and programming, as business processes may be implemented as compositions of web services [12]. Fourth, business analysts may be encouraged to use graphical languages when there are mechanisms to automatically translate them to executable models; for example, it is possible to translate process models in BPMN[4] to executable descriptions in BPEL[5] [13].

This sort of top-down implementation of business processes is in some way available in every BPM system (see for example [14]). But if we look at bottom-up approaches, and in particular to the problem of discovering business processes from event logs, we realize that current process mining techniques are able to recover process models in a representation that might not always match what the end users may be familiar with. Some techniques generate dependency graphs [15, 16], others use probabilistic models [17, 18], and most of the current techniques are geared towards retrieving Petri net models [19]. To communicate with end users, it may be necessary to translate Petri nets into other kinds of models, including business process notations such as EPC[6] [20, 21].

In this paper we present a technique for extracting process behavior directly as a structure of programming constructs. Although we make use of just a couple of basic constructs, namely decisions and loops, it is possible to extend the same technique to more intricate elements. The goal is to extract a control-flow graph whose structure and building blocks resemble those of a computer program written in an imperative programming language. The nature and scope of business processes are obviously quite different from those of software programs, but a representation in terms of programming constructs may help the end user relate more easily to the results of process mining over a given event log.

The analogy can be extended even further when we consider the content of the input log. Typically, process mining techniques require that each event in the input log is associated with a specific process instance [5]; in some scenarios this context may be unavailable. On the other hand, process mining techniques focus mainly on the control-flow perspective and only recently have begun to take into account information about data or objects being accessed, which makes it possible to identify data dependencies between events [22]. The availability of such information becomes critical when the process models must comply with known object life-cycles [23].

In this work we consider that the input event log is available as a trace of all `read` and `write` operations over a set of objects. Such read and write operations could have been recorded, for example, as accesses to document repository, or even to a version-control system. They can also be regarded as low-level events such as memory, disk, or database accesses. The objects that are accessed in these operations can be regarded either as workflow-relevant data [24] or as program variables. The log contains no information about activity, process instance, or

[4] BPMN: Business Process Modeling Notation
[5] BPEL: Business Process Execution Language for Web Services
[6] EPC: Event-Driven Process Chain

business context; it is simply a trace of all I/O operations performed by an unknown process, whose structure is to be determined.

The remainder of this paper is structured as follows. In section 2 we provide an overview of the overall approach, comprising two phases: the first phase which finds the boundaries of loops and their control variables, and the second phase which merges the sub-traces of each loop into a well-structured control-flow graph (CFG). These algorithms are explained in detail in sections 3 and 4, respectively. In section 5 we present experimental results on the application of the described approach to a set of sample traces, and evaluate the results according to a set of conformance metrics defined in [6]. Finally, section 6 concludes the paper.

2 Overview

In the following sections we describe an approach for automatically uncovering a possible control-flow representation from a sequential input trace of I/O operations. In this context, the input trace is regarded as a numbered sequence of basic `read` and `write` operations on a set of variables. Each variable, $v_i \in V$ represents an object in the specific domain of objects for the process (*e.g.*, a document, or a database table or row) that is read or updated (written).

The approach is structured into two major phases as depicted in figure 1. In a first phase, we rely on Fourier analysis of the input trace to detect operations that occur in repeatable patterns or time slots in the trace thus exhibiting some periodic behavior. This analysis determines if periodic behavior is present and which variable(s) are possible *loop control* variables, *i.e.* variables that control the execution of loop constructs in the process structure that generated the observed trace.

At the end of the first phase, the approach has identified the runs of existing loops and separated them into sub-traces of the original input trace. In a second and completely separate phase, the algorithm creates a CFG by matching and merging identical operations in the different sub-traces, while respecting the sequential relation between the operations in each sub-trace and in the resulting CFG.

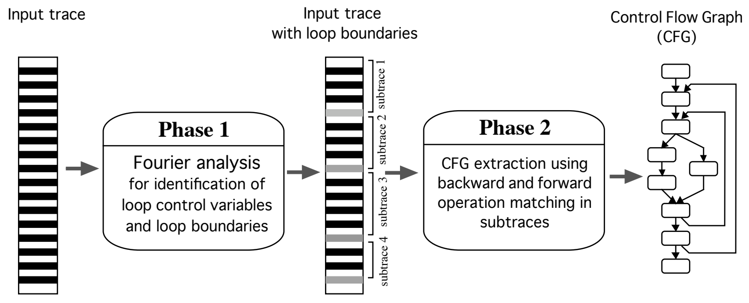

Input trace
Input trace with loop boundaries
Control Flow Graph (CFG)
Phase 1
Fourier analysis for identification of loop control variables and loop boundaries
Phase 2
CFG extraction using backward and forward operation matching in subtraces
subtrace 1
subtrace 2
subtrace 3
subtrace 4

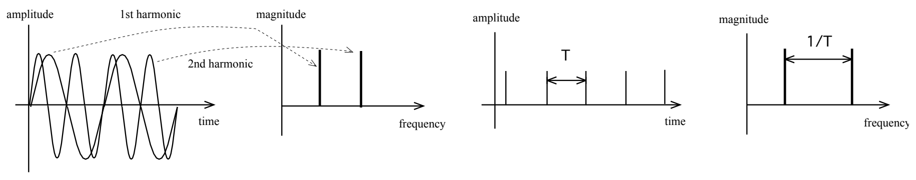
Fig. 1. Overall approach.

3 The Loop-Finding Algorithm

The first phase of the algorithm aims at finding loop constructs. We first describe the basic algorithm to uncover a single loop construct and then describe how to use this approach to find nested loop structures.

3.1 Fourier Transformation

To find a loop construct in a given section of a large trace the algorithm makes use of a digital signal processing technique – the Discrete Fourier Transform (DFT). Fourier techniques [25] translate periodic time-domain signals to the frequency domain. A periodic time-domain signal such as a sine function is represented in the frequency domain by a single magnitude and phase coefficient as depicted in figure 2(a). The Fourier transform of a generic periodic time-domain signal is a series of magnitude and phase coefficients, each corresponding to one of the harmonic frequencies of the given signal. This frequency-domain representation is called the spectrum of the input signal; a high-frequency composition on the spectrum reveals a fast changing signal and a low-frequency composition reveals a slow changing signal. For discrete periodic signals the same Fourier decomposition is possible and a periodic discrete signal with periodicity of T will have a frequency-domain representation exhibiting peaks separated by $1/T$ as illustrated in figure 2(b).



(a) continuous time domain signals and spectral representation

(b) discrete time domain signal and spectral representation

Fig. 2. Illustrative examples of Fourier Transformations (FT/DFT).

3.2 Algorithm Rationale and Description

The basis for the loop-finding algorithm relies on the observation that if a specific operation over a variable has a predominately periodic behavior, its *signature* signal must have a frequency-domain representation exhibiting clearly spaced peaks reflecting the periodicity of the time-domain signature signal. As such, one is able to uncover the periodicity associated with a specific variable and operation by examining its spectral representation.

For a given variable v and operation op (either `read` or `write`) the algorithm computes its signature signal $sig_{v,op}(k)$ as a time-domain discrete signal. This discrete signal is composed of N samples $0 \leq k \leq N-1$, one for each time sample in the input trace. The signature signal $sig_{v,op}(k)$ has value 1 for positions in the trace with an operation *op* over the v variable, and 0 otherwise. Figure 4(b) depicts a sample signature for the variable `i` and the `write` operation. The algorithm computes the Fourier transform of the signature signal for the variable v. It derives its spectral representation as a set of N complex coefficients, $S_{v,op}(k)$ with $0 \leq k \leq N-1$ and computes the corresponding magnitude $\|S_{v,op}(n)\|$ of the coefficients given by the norm-2 of its real and imaginary components as depicted by the equation in figure 3 (left).

Once the spectral representation is computed, the algorithm examines the peaks in the spectrum and determines the distances between peaks to discover the various base frequencies of the original time-domain signal. To accomplish this step, the algorithm selects the middle point in the spectral representation (*i.e.,* the coefficient at $N/2$) and measures the distance from that middle point to the next peak[7]. The inverse of this distance d is the frequency with which the operation *op* on variable v occurs in the time-domain signal.

In figure 3 (right) we illustrate an example of a spectral representation of a periodic signal corresponding to an input trace with 21 operations. Excluding the zero-frequency component (which corresponds to the average or DC component of the signal) the spectral information for the signal exhibits distinct spikes separated by 3 time units in the frequency axis. In the time domain, this frequency mode corresponds to a repetition interval of approximately $(N-1)/d = (21-1)/3$ time units, *i.e.* between 6 and 7 events.

$$S_{v,op}(n) = \frac{1}{N}\sum_{k=0}^{N-1} sig_{v,op}(k)e^{-jk2\pi n/N}$$

$$\|S_{v,op}(n)\| = \sqrt{S_{v,op}^{real}(n)^2 + S_{v,op}^{imag}(n)^2}$$

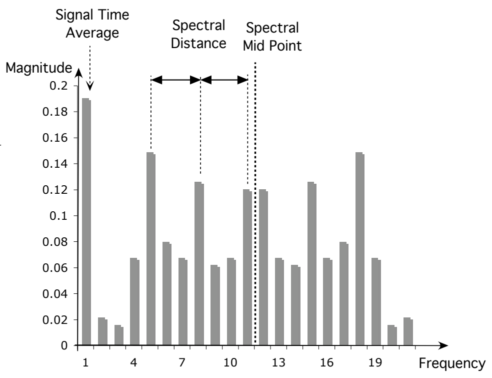


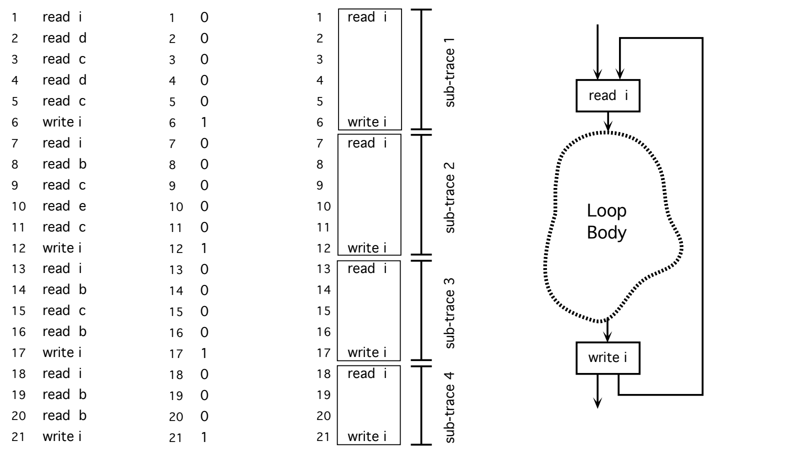
Fig. 3. DFT calculation expressions (left) and spectral example (right) where $sig_{v,op}$ is the discrete signature signal for variable v and operation *op* and $S_{v,op}(n)$ is its spectral representation for N spectral frequencies n/N for $0 \leq n \leq N-1$.

[7] A practical complication arises regarding the ability to clearly identify the peaks in the Fourier representation. The algorithm uses a thresholding step where all values below a specific percentage τ of the maximum value are eliminated. To select the value of threshold τ, the algorithm applies the Fourier analysis with values of $\tau = 1.0$ down to $\tau = 0.1$ and stops when it uncovers a feasible time-domain period.

Once the period(s) for each variable in the input trace has been identified, the algorithm selects as the loop-controlling variable the variable with the shortest period but with the highest number of occurrences in the trace[8]. Given this control variable the algorithm then scans the input trace selecting the occurrences of the variable that are located at approximate intervals corresponding to the identified period. As there can be some slight variations to the location of the occurrences of this control variable in the trace, the algorithm uses a simple windowing scheme to sync-up their occurrences in the trace.[9]

3.3 Example

Figure 4(a) presents an example of an input trace with a total of 21 `read` and `write` operations over a set of variables. For the variable `i` and operation `write` the algorithm uses Fourier analysis as depicted in figure 3 to uncover a period of approximately 6 time units, the period of `i` as the control variable for a loop construct. The algorithm then performs a linear scan over the input trace and splits the input trace into 4 sub-traces corresponding to the ranges of operations depicted in figure 4(c). The algorithm also detects a common initial operation `read i` in addition to the common final operation `write i` thus defining the entry and exit nodes of the CFG loop structure depicted in figure 4(d).



	(a)		(b)
1	read i	1	0
2	read d	2	0
3	read c	3	0
4	read d	4	0
5	read c	5	0
6	write i	6	1
7	read i	7	0
8	read b	8	0
9	read c	9	0
10	read e	10	0
11	read c	11	0
12	write i	12	1
13	read i	13	0
14	read b	14	0
15	read c	15	0
16	read b	16	0
17	write i	17	1
18	read i	18	0
19	read b	19	0
20	read b	20	0
21	write i	21	1

(a) Initial Trace (b) Signature signal sig(i,write) (c) Uncovered sub-traces for the write operation for variable i with loop boundaries (d) Control-Flow-Graph (CFG) considering loop constructs only and based on the sub-traces in (c)

Fig. 4. Sample illustrative trace with 21 operations.

[8] So that a spurious variable that occurs often but randomly is not selected.

[9] Spurious occurrences of an operation at time stamps that are not lined up with the recognized interval are not critical, as the second phase of the algorithm absorbs these instances by taking them into account as different paths in the loop control flow.

3.4 Handling Nested Loops

The approach described above enables the same algorithm to discover nested loops. The algorithm works inside-out by first finding shorter loops in the trace, corresponding to inner-loops. These loops have control variables with high spectral frequencies, as they occur more frequently than the control variables of outer loops. In order to recognize successive outer loops, the algorithm collapses the operations within an inner loop as a single aggregate macro operation and then performs a new Fourier analysis on the collapsed trace, attempting to uncover the next innermost loop. This procedure is repeated until the structure of nested loops is found and the trace collapses to a small segment where no more loops can be detected.

4 The Control-Flow Algorithm

In this second phase, the algorithm uses the sub-traces extracted during the first phase to create a well-structured CFG that can generate the entire input trace.

4.1 Algorithm Outline

The algorithm is structured into two iterative, greedy refinements phases. It terminates when it cannot refine the CFG any further. The algorithm begins by building a CFG with all the sub-traces as possible (disjoint) paths. These paths are connected to the same *source* and *sink* nodes, which represent the entry and exit nodes of a loop construct. In a first pass, the algorithm works bottom-up against the flow of the sub-traces, creating nodes in a new CFG that correspond to the operations in the sub-traces, and whenever possible it merges two identical nodes that share a common descendant node in the graph.[10] In a second pass the algorithm works top-down along the flow, this time in the CFG resulting from the first pass, and merges any two identical nodes with a common ancestor node. At the end of these two match-and-merge phases the resulting CFG is augmented by a *back edge* which connects two nodes, respectively the *head* and *tail* of a loop, thus reflecting the iterative structure of the input trace.

There is no reason not to invert the order of these two passes as both ways will allow the algorithm to derive valid but possibly different CFGs. This may produce a slight difference in the placement of decision points, without a noticeable impact in the metrics presented ahead in section 5.

4.2 Backward Match-and-Merge

In this pass the algorithm attempts to merge nodes from each sub-trace with nodes from other traces as close as possible in the linear sequence within each trace. The merged nodes become a single node shared between sub-traces as

[10] It should be noted that at the beginning of this first pass the sink node is a common descendant of all paths.

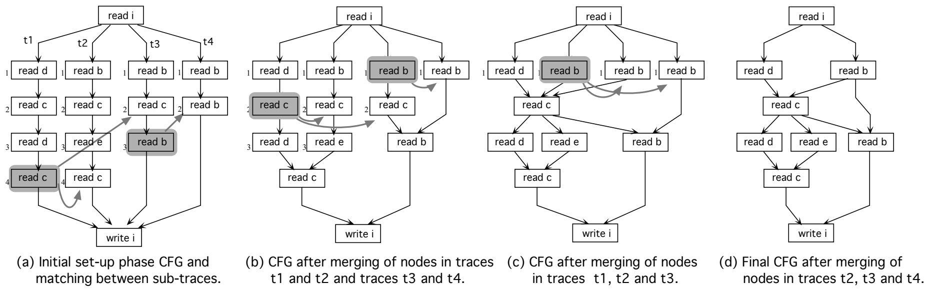

read i
t1
t2
t3
t4
read d
read b
read c
read e
write i
(a) Initial set-up phase CFG and matching between sub-traces.
(b) CFG after merging of nodes in traces t1 and t2 and traces t3 and t4.
(c) CFG after merging of nodes in traces t1, t2 and t3.
(d) Final CFG after merging of nodes in traces t2, t3 and t4.

Fig. 5. Backward Match-and-merge operation with 4 illustrative traces.

depicted in figure 5(b). The current greedy strategy for matching and merging of nodes gives preference to matches that are chronologically closer, as that increases the likelihood of merging more nodes across the sub-traces. For example, although there is a match in figure 5(a) between the operation `read c` in the sub-trace t_1 and the same operation in the sub-trace t_3, the algorithm does not merge these two operations as there is a match from an operation `read b` in sub-trace t_3 that occurs later along the control flow than that first match. As a result, the first `read c` is only matched with the same operation in sub-trace t_2 and the operation in `read b` is matched with the same operation in sub-trace t_4. Such greedy matching process continues as depicted in figure 5(b) and 5(c) until there are no more possible matches, resulting in the CFG depicted in figure 5(d).

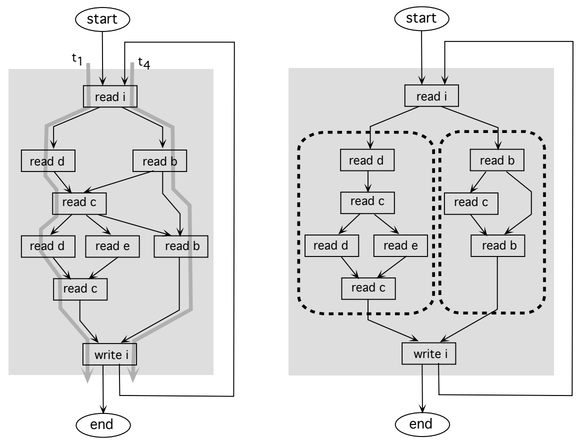
4.3 Forward Match-and-Merge

After having constructed a CFG in the first backward pass, the algorithm performs a forward match-and-merge to attempt to merge nodes that might have been left unmatched in the first phase[11]. This forward pass is also a greedy process where the algorithm iteratively merges consecutive adjacent nodes along the control flow until no further merges are possible or the end of sub-traces is reached. For the CFG derived at the end of the first step as depicted in figure 5(d), there are no opportunities for this particular transformation. The algorithm merges no nodes in this second pass and the final CFG with loop control-flow is shown in figure 6(a). As an illustrative example we depict in this final CFG two execution paths corresponding to the sub-traces t_1 and t_4 in the input sub-traces.

4.4 Structured Control-Flow

While the algorithm described above is effective in finding common subsequences among all traces preserving the sequential execution of the control flow, it can generate control-flow structures that in the lexical sense are not perfectly nested.

[11] A typical scenario occurs when two or more sub-traces get *out-of-sync* and the corresponding nodes are then left unmatched.



(a) Complete CFG with loop construct and sample execution paths through the loop body.

(b) Complete CFG with loop construct using structured aggregation of nodes during CFG construction.

```
begin
  while (Pred1) do
    read i;
    if (Pred2) then
       read d;
       read c;
       if (Pred3) then
          read d;
       else
          read e;
       end if
       read c;
    else
      read b;
      if (Pred4) then
        read c;
      endif
      read b;
    end if
     write i;
  end while
end
```

(c) High-level programming constructs for CFG derived in (b) using generic conditional and loop predicates.

Fig. 6. Complete CFG with loop as derived by the algorithm.

These imperfectly nested control-flow structures cannot be described using high-level programming constructs without the use of `goto` or other arbitrary control-flow transfer primitives [26]. More importantly, in the context of process flows, it substantially complicates conformance checking between processes.

To overcome this problem we developed a variant of the control-flow algorithm that always generates structured CFGs at the expense of the loss of some sharing of nodes between sub-traces. The revised algorithm keeps the matching of nodes confined to a subset of traces that have been already matched in a previous matching step. The algorithm thus partitions the input traces into disjoint subset of traces which are then refined as the matching progresses. As a result, the matching and thus the merging is always done between traces that have a common descendant towards the sink node. The resulting CFG is thus less compact than the CFG generated by the first control-flow algorithm.

Figure 6 illustrates the CFGs resulting from the two variants of the algorithm for the same 4 illustrative sub-traces used in figure 5. Figure 6(a) presents the CFG resulting from the application of the first algorithm, whereas figure 6(b) presents the CFG obtained using the variant of the algorithm that generates structured CFGs. For this second CFG we depict the corresponding high-level program description that can generate the various traces in figure 6(c).

4.5 Algorithmic Complexity

Given that both the backward and the forward passes traverse and advance through each sub-trace at least one operation at a time in each iteration, the algorithm eventually terminates. At each iteration the algorithm performs in the worst case $O(n^2)$ when matching operations between the k sub-traces. On the other hand, each sub-trace is $O(n)$ long and this length is linearly related to the length of the input trace. Considering book-keeping and the manipulation of auxiliary data structures as a constant, the worst-case time complexity becomes $O(n^3)$ where n is the number of operations in each sub-trace.

Despite this worst-case complexity behavior, we do anticipate the algorithm to perform well given that at each step more than one node can be processed.[12]

5 Results

We have implemented the above algorithms in approximately 4,000 lines of C code. In this section we describe a series of experiments with sample traces to evaluate the ability of the algorithm to extract a CFG for each input trace. In all these experiments we have used the revised version of the algorithm in order to derive structured CFGs. We then evaluated each of the derived CFGs according to a set of fitness and appropriateness metrics defined in [6].

5.1 Sample Traces

To support our experiments we generated a set of 4 traces with `read` and `write` operations over a set of scalar variables such as `i` or `a`.[13] Each trace is generated by a different C program that outputs the various operations reflecting its own execution.[14]

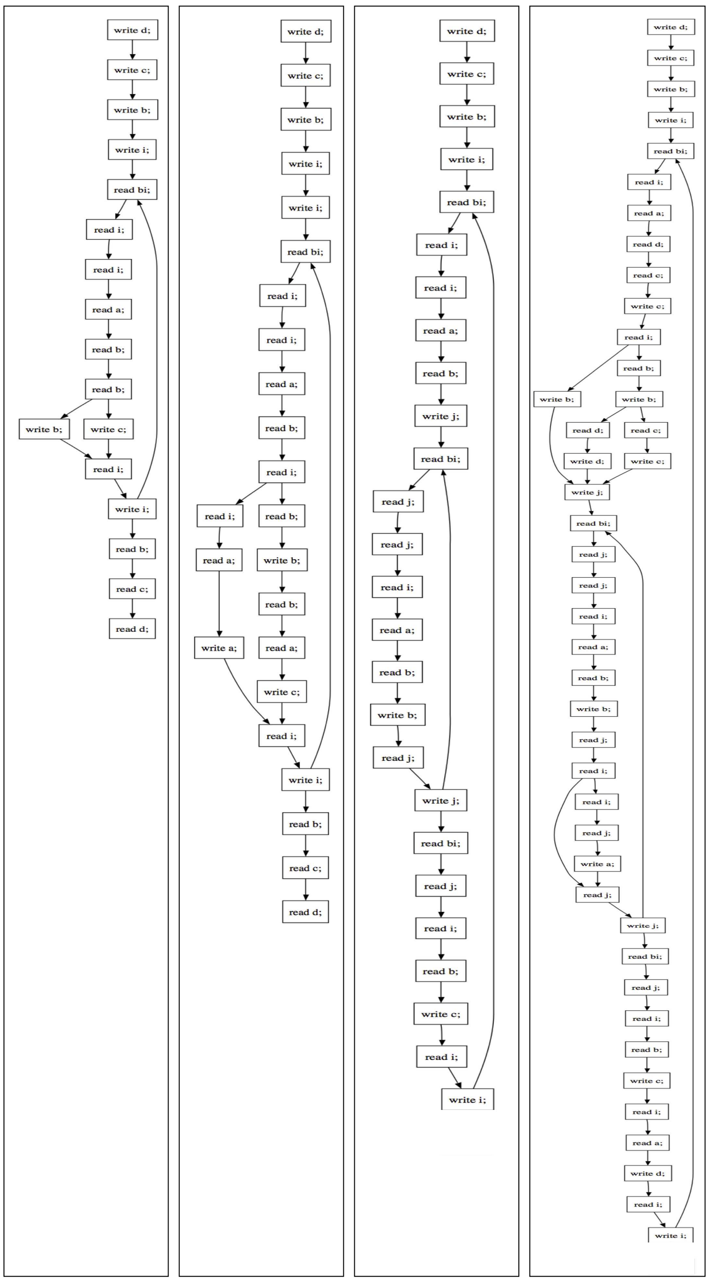
Table 1 presents the experimental results for the various traces. In the left section of the table we have a series of results characterizing the traces used, whereas on the right section we have results regarding metrics as discussed in the next section.

In table 1 we characterize each trace by its length or number of operations, and also by the number of variables that each trace contains. These are shown in columns 2 and 3, respectively. Column 4 presents the control variables extracted by the Fourier analysis phase of our algorithm, indicating for each variable the uncovered period (as an interval of two values). In column 5 we report the number of sub-traces identified for each control variable, and in column 6 we indicate the nested structure of the discovered CFG. Figure 7 shows the CFGs discovered for the 4 sample traces.

[12] Common string-matching algorithms such as the longest common sub-sequence (LCS) or the shortest common super-sequence (SCS) problems are impractical. Both LCS [27] and SCS [28] problems require $O(n^2)$ solutions for two sequences of length n, but are NP-hard for k strings. An incremental 2-at-a-time approach would lead to a time complexity of $O(n^k)$.

[13] The various input traces and CFG outputs are available at the following location: http://www.dei.ist.utl.pt/∼ped/submissions/BPM08

[14] For example, in the execution of loop constructs the trace generator will output the operations required to evaluate the loop control predicate. Updating the loop control variable, say `i = i + 1;` involves a read operation followed by a write operation.



(a) CFG for input trace 1

(b) CFG for input trace 2

(c) CFG for input trace 3

(d) CFG for input trace 4

Fig. 7. Complete CFGs for the 4 sample traces.

Table 1. Characteristics of the sample traces used in experiments, and metrics for the derived CFG solutions.

Trace	**Characteristics**					**Metrics**						
	number operations	number variables	control var. and period	number sub-traces	nesting structure	c_E	c_T	f	a_B	a'_B	a_S	a'_S
#1	52	7	i:[8,9]	5	single loop	1.000	1.000	1.000	0.988	0.924	0.588	1.000
#2	109	6	i:[12,13]	8	single loop	1.000	1.000	1.000	0.994	0.849	0.458	1.000
#3	1034	6	i:[101,102] j:[8,9]	10 101	doubly nested	1.000	1.000	1.000	0.998	1.000	0.407	1.000
#4	1108	7	i:[121,122] j:[8,9]	10 82	doubly nested	1.000	1.000	1.000	0.996	0.953	0.311	1.000

5.2 Metrics

The algorithms described above produce a CFG that is in effect one possible explanation for the behavior observed in the input trace. To assess the interest and potential of this approach, and to be able to compare it with other process mining algorithms, it is necessary to determine the extent to which the produced CFG is actually a good explanation for the original behavior. This assessment can be done by applying a set of metrics to compare the behavior allowed in the model with the behavior observed in the input trace. Such analysis can be done, for example, by checking if the process model would allow all events in the trace to occur in the same order. This is the underlying rationale for the *fitness* metric [6]. Other metrics – such as *behavioral appropriateness* and *structural appropriateness* [6] – are also useful to check that the model is a compact and non-redundant representation of the intended process.

Even though these metrics have been defined for Petri net models, it is possible to apply them to CFGs with small adaptations. The only metric that requires a significant adaptation is the *fitness* metric, as it is based on replaying the trace in the model. While the execution semantics of Petri nets are formally well-established (via tokens, places and transitions), those of CFGs may vary; for example, splits and joins could be given different interpretations. In the CFGs we use above, there are only OR-splits and OR-joins so the semantics become quite simple. We therefore redefined the *fitness* metric as the percentage of trace events that can be successfully replayed in the CFG in the same order.

Table 1 shows the results obtained for each trace and metric. Table 2 summarizes the purpose of each metric and provides a brief explanation of the typical results obtained in several runs of the algorithm for different input traces. A detailed description of these metrics can be found in [6].

Table 2. Applying the metrics defined in [6].

Metric	**Results**
Log coverage: $c_E = \frac{\lvert\{e \in E \mid l_E(e) \in L_T\}\rvert}{\lvert E\rvert}$	This metric checks if each event in the input trace is represented as a node label in the CFG. Since they all are, the metric is always 100%.
Model coverage: $c_T = \frac{\lvert\{t \in T_V \mid l_T(t) \in L_E\}\rvert}{\lvert T_V\rvert}$	This metric checks if each label in the CFG appears as an event in the trace. Since they all do, the metric is always 100%.
Fitness: $f = \frac{1}{2}\left(1 - \frac{m}{c}\right) + \frac{1}{2}\left(1 - \frac{r}{p}\right)$	This metric checks whether events can be replayed in the model in the same order as they appear in the trace. It is originally defined in terms of consumed (c) and produced tokens (p), as well as missing tokens (m) during replay and remaining tokens (r) after replay. Since we are using CFGs rather than Petri nets, this metric has been redefined as the percentage of trace operations that can be replayed in the same order in the CFG. As the trace is always a possible path in the resulting CFG, the metric is always 100%.
Simple behavioral appropriateness: $a_B = \frac{\lvert T_V\rvert - x}{\lvert T_V\rvert - 1}$	This metric uses the number of *visible tasks* $\lvert T_V\rvert$ (which in our case is the number of CFG nodes) and the mean number (x) of possible transitions at each step when replaying the trace. As the produced CFGs usually have a few decision points, at some steps there is more than one option for the next step, and hence the metric is usually close but not equal to 100%. Depending on trace length and the number of decision points, it is usually above 99%.
Advanced behavioral appropriateness: $a'_B = \frac{\lvert S^l_F \cap S^m_F\rvert}{2 \cdot \lvert S^m_F\rvert} + \frac{\lvert S^l_P \cap S^m_P\rvert}{2 \cdot \lvert S^m_P\rvert}$	This metric computes the *follows* and *precedes* relations both for the trace (S^l_F and S^l_P) and for the CFG (S^m_F and S^m_P). These relations basically say whether any given pair of operations *always*, *sometimes* or *never* follow each other. Based on these relations, the metric checks whether the CFG allows for more variability in behavior than that observed in the trace. Due to the presence of decision points, the CFG usually does allow for more variability, hence the metric is typically in the range 85%-95%.
Simple structural appropriateness: $a_S = \frac{\lvert L\rvert}{\lvert N\rvert}$	This metric checks the amount of duplicate nodes in the CFG (*i.e.* nodes with the same label) by dividing the number of distinct labels $\lvert L\rvert$ by the number of nodes $\lvert N\rvert$. As the match-and-merge procedure avoids merging nodes that would cause the model to become unstructured, it is usually the case that a number of duplicate nodes remain in the CFG. In relatively complex CFGs, the metric can be as low as 30%. It should be noted, however, that the true model (*i.e.* the program that generated the input trace) may have duplicate operations, so this simple metric is not a reliable measure of accuracy in our experiments.
Advanced structural appropriateness: $a'_S = \frac{\lvert T\rvert - (\lvert T_{DA}\rvert + \lvert T_{IR}\rvert)}{\lvert T\rvert}$	This metric penalizes the model when there are *redundant invisible tasks* (T_{IR}) or *alternative duplicate tasks* (T_{DA}). An *invisible task* is an operation that is represented in the model but is not present in the trace; in our CFGs there are no invisible tasks. Two CFG nodes are said to be *alternative duplicate tasks* if they have the same label but there is no possibility that they could occur in the same trace. The match-and-merge steps in our algorithm eliminate these alternative duplicate nodes, hence the value for this metric is always 100%.

6 Conclusion

The analogy between process modeling and programming suggests that it should be possible to approach process mining from the perspective of discovering program structure. This is useful not only because process modeling and execution languages make use of building blocks that resemble programming constructs, but also because the way a program operates on variables can be seen as being analogous to the way workflow participants manipulate documents and other data objects.

In this paper we described a combination of techniques to extract process behavior directly as a control-flow graph from a trace of all read and write operations over a set of objects. The approach is divided in two main phases where, in the first phase, signal processing techniques are used to detect periodic behavior that can be potentially represented with loop constructs. This is an interesting challenge since the problem of delimiting repeating behavior is not addressed by current process mining techniques, and the DFT proves to be an effective tool for this purpose. It also enables the algorithm to uncover nested loops, which can be abstracted as sub-processes. Furthermore, as a signal processing technique it is inherently more robust to noise than discrete algorithmic approaches.

In the second phase, the algorithm proceeds with a set of unsupervised match-and-merge steps to produce a structured graph by consolidating the behavior of different sub-traces, and by reducing their differences to a set of decision points. This technique can be used as a process mining algorithm by itself, if several traces of the same process are provided as input. In this case, the algorithm would be equivalent to beginning with M3 in [6] and proceed by merging nodes until an appropriate model is found. The match-and-merge procedure described in section 4.4 ensures that the outcome is a structured model.

Overall, the combination of these techniques is able to produce compact and accurate models of control-flow behavior, and exhibits very good results in terms of conformance metrics. Although only loops and decision constructs have been addressed in the present work, we are currently studying techniques to support other behavioral patterns, in particular parallel constructs.

Acknowledgment

The second author would like to acknowledge the support of *Fundação para a Ciência e a Tecnologia* (FCT) through project no. PTDC/CCI/70512/2006.

References

1. Aalst, W.v.d.: The application of petri nets to workflow management. Journal of Circuits, Systems and Computers **8**(1) (1998) 21–26
2. Aalst, W.v.d.: Woflan: a petri-net-based workflow analyzer. Systems Analysis Modelling Simulation **35**(3) (1999) 345–357
3. Aalst, W.v.d.: Loosely coupled interorganizational workflows: Modeling and analyzing workflows crossing organizational boundaries. Information and Management **37**(2) (2000) 67–75
4. Aalst, W.v.d., ter Hofstede, A.: Loosely coupled interorganizational workflows: Modeling and analyzing workflows crossing organizational boundaries. Information Systems **25**(1) (2000) 43–69
5. Aalst, W.v.d., Weijters, A., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE Transactions on Knowledge and Data Engineering **16**(9) (2004) 1128–1142
6. Rozinat, A., van der Aalst, W.: Conformance checking of processes based on monitoring real behavior. Information Systems **33**(1) (2008) 64–95
7. Aalst, W.v.d., ter Hofstede, A.: YAWL: Yet another workflow language. Information Systems **30**(4) (2005) 245–275
8. Aalst, W.v.d., ter Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow patterns. Distributed and Parallel Databases **14**(1) (2003) 5–51
9. Rosemann, M., Recker, J., Indulska, M., Green, P.: A study of the evolution of the representational capabilities of process modeling grammars. In: Proceedings of the 18th Conference of Advanced Information Systems Engineering (CAiSE 2006), Berlin, Springer (2006) 447–461
10. Aalst, W.v.d., ter Hofstede, A., Weske, M.: Business process management: A survey. In: Proceedings of the International Conference on Business Process Management (BPM 2003), Berlin, Springer (2003) 1–12
11. Yang, G.: Process library. Data and Knowledge Engineering **50**(1) (2004) 35–62
12. Emig, C., Weisser, J., Abeck, S.: Development of SOA-based software systems – an evolutionary programming approach. In: Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT-ICIW'06), Washington, DC, IEEE Computer Society (2006) 182
13. Ouyang, C., Dumas, M., ter Hofstede, A., van der Aalst, W.: Pattern-based translation of BPMN process models to BPEL web services. International Journal of Web Services Research **5**(1) (2008) 42–61
14. Kloppmann, M., Knig, D., Leymann, F., Pfau, G., Roller, D.: Business process choreography in websphere: Combining the power of bpel and j2ee. IBM Systems Journal **43**(2) (2004) 270–296
15. Agrawal, R., Gunopulos, D., Leymann, F.: Mining process models from workflow logs. In: Proceedings of the 6th International Conference on Extending Database Technology (EDBT'98), Berlin, Springer (1998) 469–483
16. Greco, G., Guzzo, A., Pontieri, L.: Mining hierarchies of models: From abstract views to concrete specifications. In: Proceedings of the 3rd International Conference on Business Process Management (BPM 2005), Berlin, Springer (2005) 32–47
17. Herbst, J., Karagiannis, D.: Integrating machine learning and workflow management to support acquisition and adaption of workflow models. In: Proceedings of the 9th International Workshop on Database and Expert Systems Applications, IEEE (1998) 745–752

18. Ferreira, D., Zacarias, M., Malheiros, M., Ferreira, P.: Approaching process mining with sequence clustering: Experiments and findings. In: Proceedings of the 5th International Conference on Business Process Management (BPM 2007), Berlin, Springer (2007) 360–374
19. Aalst, W.v.d., van Dongen, B., Herbst, J., Maruster, L., Schimm, G., Weijters, A.: Workflow mining: A survey of issues and approaches. Data and Knowledge Engineering **47**(2) (2003) 237–267
20. van Dongen, B., van der Aalst, W.: Multi-phase process mining: Building instance graphs. In: Proceedings of the International Conference on Conceptual Modeling (ER 2004), Berlin, Springer (2004) 362–376
21. Verbeek, H., van Dongen, B.: Translating labelled P/T nets into EPCs for sake of communication. BETA Working Paper Series WP 194, Eindhoven University of Technology (2007)
22. Rozinat, A., Mans, R., van der Aalst, W.: Mining CPN models: Discovering process models with data from event logs. In: Proceedings of the Seventh Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2006), Denmark, University of Aarhus (2006) 57–76
23. Kuester, J., Ryndina, K., Gall, H.: Generation of business process models for object life cycle compliance. In: Proceedings of the 5th International Conference on Business Process Management (BPM 2007), Berlin, Springer (2007) 165–181
24. Hollingsworth, D.: The workflow reference model. Technical Report TC00-1003, Workflow Management Coalition (1995)
25. Oppenheimer, A., Shaffer, R.: Digital Signal Processing. Prentice-Hall, Englewood Cliffs, N.J., USA (1975)
26. Aho, A., Sethi, R., Ullman, J.: Compilers: Principles, Techniques and Tools. Addison-Wesley, Inc., Reading, Mass., USA (1986)
27. Hirschberg, D.S.: Algorithms for the longest common subsequence problem. Journal of the ACM **24**(4) (1977) 664–675
28. Fraser, C., Irving, R.: Approximation algorithms for the shortest common supersequence. Nordic Journal of Computing **2**(3) (1995) 303–325