# Discovering Branching Conditions from Business Process Execution Logs

Massimiliano de Leoni[1], Marlon Dumas[2], and Luciano García-Bañuelos[2]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
[2] University of Tartu, Tartu, Estonia

**Abstract.** Process mining is a family of techniques to discover business process models and other knowledge of business processes from event logs. Existing process mining techniques are geared towards discovering models that capture the order of execution of tasks, but not the conditions under which tasks are executed – also called branching conditions. One existing process mining technique, namely ProM's Decision Miner, applies decision tree learning techniques to discover branching conditions composed of atoms of the form "v op c" where "v" is a variable, "op" is a comparison predicate and "c" is a constant. This paper puts forward a more general technique to discover branching conditions where the atoms are linear equations or inequalities involving multiple variables and arithmetic operators. The proposed technique combine invariant discovery techniques embodied in the Daikon system with decision tree learning techniques.

## 1 Introduction

The use of business process models to analyze and automate business operations is a widespread practice. Traditionally, business process models are obtained from interviews and workshops with domain experts and workers. Studies have shown however that models obtained in this way may deviate significantly from the way processes are actually conducted on a daily basis [1]. Workers tend to take shortcuts or workarounds in order to deal with special cases or to simplify their work. At the same time, contemporary enterprise systems maintain detailed records of transactions performed by workers, which can be exploited to discover models that more faithfully reflect the way processes are actually performed.

This observation has spawned a research area known as process mining [1], which is concerned with the automated discovery of process models and other knowledge of business processes from event logs. Several algorithms for automated process discovery have been developed, which strike different tradeoffs between accuracy and comprehensibility of the discovered models.

To illustrate the capabilities and limitations of these algorithms, we consider a process for handling loan applications (cf. Figure 1). This process starts when a loan application is made. First, the loan application details are entered into a system – in particular the amount and length of the loan which are hereby treated as variables of the process. Next, data about the applicant (e.g. age and salary) are retrieved from the customer database. In parallel, the amount of

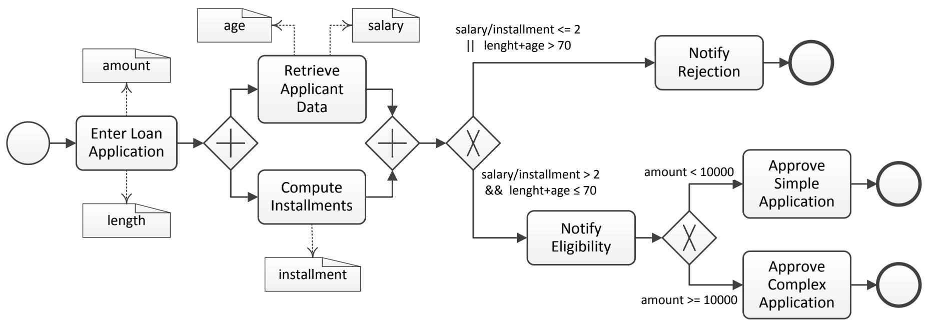


**Fig. 1.** Running example: loan application process model (in BPMN notation).

each installment is calculated and stored as a variable. The request is eligible if the applicant's salary is more than twice the amount of the installment and they would finish paying installments by the age of 70. If the applicant is not eligible, the application is rejected. If eligible, the applicant is notified and their application is forwarded for approval. If the requested amount is less that 10000, a simple approval suffices; otherwise, a more complex approval is required.

The bulk of automated process discovery algorithms are focused on extracting control-flow relations between events or tasks in a process. In the working example, these algorithms would discover the sequence relations in the model, the parallel execution relations (the "+" gateways in Figure 1) and the conditional *branching points* where a choice is made between alternative branches (the "X" gateways in Figure 1). However, they do not discover the conditions attached to the outgoing branches of these branching points – also called *branching conditions*.

An attempt to address this limitation is the Decision Miner [2] embodied in the ProM toolset [1]. The Decision Miner applies decision tree learning to infer conditions composed of atoms of the form "v op c" where "v" is a variable, "op" is a comparison predicate and "c" is a constant. In the running example, the Decision Miner can discover the condition $amount \geq 10000$ (and its dual), but not the conditions attached to the leftmost branching point in the model.

This paper tackles the problem of discovering branching conditions where the atoms are equalities or inequalities involving arithmetic expressions on multiple variables. The starting point is the invariant detection technique embodied in the Daikon system [3]. A direct application of Daikon allows us to discover invariants that hold before (or after) a task is executed. The discovered invariants are of the form "$v_1$ op $c$" or "$v_1$ op $v_2$" where $v_1$ and $v_2$ are variables and $c$ is a constant. By combining these invariants via conjunction, we can discover conditions that hold at the start of each branch. However, this approach has three limitations:

L1. Many of the invariants detected by Daikon are not suitable for inclusion in a branching condition. A branching condition should discriminate between the cases when one branch is taken and those when it is not taken, while an invariant that holds in one branch may equally well hold in the alternative branch. In the running example, Daikon may detect that invariants $age \leq 70$ or $salary \leq amount$ hold just before task *Notify Rejection*, but these are clearly not relevant for inclusion in the corresponding branching condition.

L2. Daikon does not discover conditions that include disjunctions.

L3. Daikon does not discover inequalities where the atoms involve more than two variables combined via arithmetic operators.

To overcome these limitations, we combine Daikon with decision tree learning. In particular, we use the notion of *information gain* from decision tree learning to determine which invariants should be combined into branching conditions. Three techniques of increasing degree of sophistication are proposed, which overcome each of the above three limitations in turn. The techniques have been validated on a set of test cases covering branching conditions with different structures.

The paper is structured as follows. Section 2 introduces ProM's Decision Miner and Daikon and discusses other related work. Section 3 presents the techniques for branching condition discovery while Section 4 documents their validation. Section 5 discusses some remaining limitations and directions for future work.

## 2 Background and Related Work

### 2.1 ProM Decision Miner

ProM's Decision Miner allows one to discover branching conditions for a given branching point in a process model. These conditions depend on the value of the variables of the process when the point is reached. Accordingly, the input of ProM's Decision Miner consists of an event log where each event contains: (i) a timestamp; (ii) an event type that allows to link the event to a task; and (iii) a set of ⟨ variable, value ⟩ pairs, representing the new values of variables modified by the event in question. The events in the log are grouped into traces, where each trace is a temporally-ordered sequence of events representing one execution of a process from start to end (also called a *case*). The mechanism used to group events in a log into traces is not relevant for this paper (see e.g. [1]).

ProM's Decision Miner assumes that a process model has been discovered from the event log prior to discovering the branching conditions. This process model can be obtained using one of the existing process discovery algorithms.

Given the event log, the discovered process model, and one of the branching points P in the model, ProM's Decision Miner first calculates, for each enablement of P in a trace (i.e. each traversal of P when replaying the trace against the model), the following: (i) the values of each variable in the process when P is enabled; and (ii) the identifier of the branch that was taken after the branching point was traversed (the *outcome*). A variable assignment and its associated outcome for a given enablement of P is called an observation instance. The set of observation instances of P is used to mine a decision tree. This decision tree is transformed into a disjunction of conjunctions of atomic expressions as follows: Each path from the root to a leaf labelled with outcome $t_i$ becomes a conjunctive expression associated to branch $t_i$. The branching condition of a branch $t_i$ is the disjunction of the conjunctive expressions derived from the paths leading to $t_i$.

ProM's original Decision Miner [2] cannot handle process models with cycles or with invisible ("skip") tasks. These limitations however are addressed in [4].

Since the internal nodes of a decision tree are labeled with expressions of the form variable-op-constant, the Decision Miner only discovers conditions consisting of atoms of this form. In the running example, the Decision Miner cannot discover the conditions associated with *Notify Rejection* and *Notify Eligibility.*

The Decision Miner assumes that the branches of a branching point are exclusive, i.e. exactly one of the branches is selected. Process modeling languages also support inclusive branching points where more than one branch can be taken in parallel. However, an inclusive branching point can be transformed into an exclusive branching point and one or more parallel gateways.

## 2.2 Invariant Discovery and Specification Mining

Daikon [5] is a dynamic analysis tool for inferring likely value-based invariants from a collection of execution traces. It operates by instantiating a set of invariant templates with the variables in the logs, and trying to match each instantiated template against the variable assignments recorded in the traces. It outputs a set of invariants with sufficient statistical support.

Daikon relies on code instrumentation tools that expose the value of variables at points of interest in a program. For example, Java bytecode can be instrumented so as to monitor actual parameters used in method calls. In this way, Daikon discovers method pre- and postconditions, and derives object-level invariants.

Daikon comes with a large set of invariant templates, ranging from simple relational expressions on variable/value or variable/variable pairs (e.g., $x < y$) to sophisticated templates of linear relations over multiple variables (e.g., $x - 3 * y = 115$). Daikon usually discovers interesting invariants but it may also report irrelevant invariants. In our running example, for instance, Daikon may discover the invariant $length < salary$, assuming that both variables $length$ and $salary$ have integer values. To cope with this problem, Daikon uses static analysis of the target program source code to identify meaningful combinations of variables [6]. Such analysis may reveal, for instance, that variables $salary$ and $installment$ are used together in arithmetic expressions while variables $salary$ and $length$ are not. Daikon then avoids instantiating templates that combine variables $salary$ and $length$, thus improving the relevance of the discovered invariants. In our setting, however, static analysis is not possible as no source code is available.

For the problem at hand, we can use Daikon to discover invariants for each task that follows a branching point in the target process model. These invariants could be put into conjunctive expressions, that would be then used as branching conditions. This approach has been explored in [7, 8]. However, Daikon may discover invariants that are not necessarily branching conditions. For example, $amount > 0$ holds for all activities in the process model in Figure 1. This atom may appear in all the branching conditions in the process model, even if it is not directly involved in the decision in question.

Several extensions of Daikon have been proposed. One of them [9] discovers object-level invariants including disjunctions (limitation L2 in Section 1). However, this extension requires the source code to be analyzed and in our problem setting no source code is available. Other related work includes alternative oracles for discovering potential invariants, such as the one proposed by [10], which produces higher-order polynomial invariants as opposed to only linear invariants as Daikon.

Daikon is an exemplar of a broader class of so-called specification mining techniques [11–13]. Specification mining is concerned with discovering temporal and data-dependent knowledge about a program or protocol. The discovered knowledge is represented, for example, as state machines. A distinctive feature of process mining compared to specification mining is that process mining is concerned with discovering concurrent behavior in addition to sequential behavior. Also, in process mining, no source code is assumed to be available.

## 3 From Invariants to Branching Conditions

This section proposes three techniques to address the limitations of Daikon highlighted in Section 1. Section 3.1 addresses L1. Section 3.2 extends the technique of Section 3.1 to address L2. Finally, Section 3.3 extends further to overcome L3. Section 3.4 discusses the case of N-ary branching points.

A binary branching point is denoted by a set $\{t_1, t_2\}$ where $t_1$ and $t_2$ are the first tasks of the two branches. For convenience, we associate each branching condition with the first task of the respective branch, i.e. with $t_1$ or $t_2$.

Daikon is trained on a set of observation instances relative to a task. Here, an observation instance is a function $i : V \rightarrow U$ that assigns a value $i(v)$ to each variable $v \in V$. Given a set $\mathcal{I}$ of observation instances, we abstract Daikon as a function $\textsc{DiscoverInvariantsWithDaikon}(\mathcal{I})$ that returns a conjunctive expression of atoms that are invariants with respect to instances in $\mathcal{I}$. Sometimes, Daikon is not able to discover invariants; in these cases, the special value $\perp$ is returned. Observation instances relative to a task are extracted from an event log as follows:

**Definition 1 (Event Log).** *Let $T$ and $V$ be a set of tasks and variables, respectively. Let $U$ be the set of values that can be assigned to every variable $v \in V$. Let $\Phi$ be the set of all functions $V \not\rightarrow U$ that define an assignment of values to a subset of variables in $V$. An* ***event log*** *$\mathcal{L}$ over $T$, $V$ and $U$ is a multiset of traces where each trace is a sequence of events of the form $(t, \phi)$, where $t \in T$ is a task and $\phi \in \Phi$ is an assignment of values to a subset of variables in $V$ In other words, $\mathcal{L} \in \mathcal{B}((T \times \Phi)^*)$.*[3]

The observation instance relative to an execution of a task $t$ in a given case (i.e. a process execution) consists of the values of the variables in the case prior to the execution of $t$. Algorithm 1 shows how observation instances are constructed from an event log. The output is a function $I$ that associates each task $t \in T$

[3] $\mathcal{B}(X)$ the set of all multisets over $X$.

**Algorithm 1:** GENERATEOBSERVATIONINSTANCES

```
Data: $\mathcal{L}$ – An event log over $T$ and $V$
Result: A function $I$ that associated each task in $T$ with a set of observation
        instances
1 Let $I$ be a function whose domain is $T$ and $\forall t \in T.\ I(t) = \emptyset$.
2 foreach trace $\langle (t_1, \phi_1), \ldots, (t_n, \phi_n) \rangle \in \mathcal{L}$ do
3     Let $M$ be a function whose domain is $V$ and $\forall v \in V.\ M(v) = \perp$
4     for $i \leftarrow 1$ to $n$ do
5         $I(t_1) \leftarrow I(t_1) \cup M$
6         foreach variable $v$ in the domain of $\phi_i$ do  $M(v) \leftarrow \phi_i(v)$
7     end
8 end
9 return I
```

**Algorithm 2:** DISCOVERCONJUNTIVECONDITIONSWITHDAIKON (CD+IG)

```
Data: $\mathcal{L}$ – An event log, $P$ – A process model
Result: A map that associates some transitions with the corresponding
        branching conditions
1 $I \leftarrow$ GENERATEOBSERVATIONINSTANCES($\mathcal{L}$)
2 foreach $\{t_1, t_2\} \in$ BRANCHINGPOINTS($P$) do
3     $C(t_1) \leftarrow$ DISCOVERINVARIANTSWITHDAIKON($I(t_1)$)
4     $C(t_2) \leftarrow$ DISCOVERINVARIANTSWITHDAIKON($I(t_2)$)
5     $C(t_1) \leftarrow$ BUILDCONJUNTIVEEXPR($I(t_1), I(t_2), C(t_1)$)
6     $C(t_2) \leftarrow$ BUILDCONJUNTIVEEXPR($I(t_1), I(t_2), C(t_2)$)
7     ADJUSTCONDITIONS($I(t_1), I(t_2), C(t_1), C(t_2)$)
8 end
9 return C
```

with a set of observation instances relative to $t$. The algorithm is based on the principle of replay. Each trace is associated with a function $M : V \rightarrow U$ that keeps the assignment of values to variables. After an event is replayed, function $M$ is rewritten according to the event's value assignments. Initially, for each $v \in V$, $M(v) = \perp$, where $\perp$ is a special value that identifies an undefined assignment. Before replaying an event $e$ for a certain task $t$, a new observation instance is created and added to the set of instances for task $t$ (line 5). Afterwards, $e$ is replayed and function $M$ is rewritten accordingly (line 6).

For convenience, we will say that the observation instances of a branch are the observation instances relative to the first task of that branch.

## 3.1 Discovery of Conjunctive Conditions

In order to construct the branching conditions from the invariants discovered by Daikon, we leverage on the concept of *information gain*. In data mining, the concept of information gain captures how well a given predicate distinguishes

**Algorithm 3:** BuildConjuntiveExpr

```
Data: $\mathcal{I}_1, \mathcal{I}_2$ – Two sets of observation instances, $P$ – A conjunctive expression
Result: A conjunction of a subset of the atoms in $P$ that maximizes the
        information gain
1  if $P = \perp$ then return $\perp$
2  $S \leftarrow \{p_1, p_2, \ldots, p_n\}$ s.t. $P = p_1 \wedge p_2 \wedge \ldots \wedge p_n$
3  Pick $\overline{q} \in S$ s.t. $\forall q' \in S.\ IG(\mathcal{I}_1, \mathcal{I}_2, q') \leq IG(\mathcal{I}_1, \mathcal{I}_2, \overline{q})$
4  $\overline{P} \leftarrow \overline{q}$
5  $S \leftarrow S \setminus \{\overline{q}\}$
6  while $S \neq \emptyset$ do
7      Pick $\overline{q} \in S$ s.t. $\forall q' \in S.\ IG(\mathcal{I}_1, \mathcal{I}_2, \overline{P} \wedge q') \leq IG(\mathcal{I}_1, \mathcal{I}_2, \overline{P} \wedge \overline{q})$
8      if $IG(\mathcal{I}_1, \mathcal{I}_2, \overline{P} \wedge \overline{q}) > IG(\mathcal{I}_1, \mathcal{I}_2, \overline{P})$ then $\overline{P} \leftarrow \overline{P} \wedge \overline{q}$  $S \leftarrow S \setminus \{\overline{q}\}$
9  end
10 return $\overline{P}$
```

between two or more possible outcomes (tasks in our case). In our context, the information gain of a predicate $P$ relative to a binary decision point leading to tasks T1 and T2, is a measure of how well predicate $P$ distinguishes the observations instances where task T1 is executed from those where task T2 is executed. A predicate that holds iff T1 is executed or a predicate that holds iff T2 is executed has maximum information gain. A predicate that does not give any gain (beyond random choice) when it comes to determining whether task T1 or T2 is executed has zero information gain. Given a two sets of observation instances leading to two tasks, the maximum possible value of the information gain is called the *entropy*, as formally defined below.

**Definition 2 (Entropy).** *Let $\mathcal{I}'$ and $\mathcal{I}''$ be two sets of observation instances that lead to the execution of to task $t'$ and task $t''$, respectively. Moreover, let $p(t)$ denote the probability of executing the task $t$. Then, the* entropy *of $\mathcal{I}'$ and $\mathcal{I}''$ is defined as $H(\mathcal{I}', \mathcal{I}'') = -p(t') \cdot \log_2(p(t')) - p(t'') \cdot \log_2(p(t''))$. Since $p(t')$ can be expressed as $|\mathcal{I}'|/(|\mathcal{I}'| + |\mathcal{I}''|)$, we reformulate entropy as:*

$$H(\mathcal{I}', \mathcal{I}'') = -\left(\frac{|\mathcal{I}'|}{|\mathcal{I}'| + |\mathcal{I}''|} \cdot \log_2 \frac{|\mathcal{I}'|}{|\mathcal{I}'| + |\mathcal{I}''|}\right) - \left(\frac{|\mathcal{I}''|}{|\mathcal{I}'| + |\mathcal{I}''|} \cdot \log_2 \frac{|\mathcal{I}''|}{|\mathcal{I}'| + |\mathcal{I}''|}\right)$$

Entropy is 1 if sets $\mathcal{I}'$ and $\mathcal{I}''$ are of the same size. It becomes close to 0 if the sets are of very different sizes. It is 0 if either $\mathcal{I}'$ or $\mathcal{I}''$ is empty (taking $0 \log_2 0 = 0$). The intuition is that if we partition a set into a large subset and a small one, this partition has little information, as the smaller set can be encoded with few bits. Meanwhile, if we partition a set into equal-sized subsets, more information is required to distinguish between the two subsets. Given two disjoint sets of observation instances, our goal is to identify a predicate that comes as close as possible to perfectly classifying instances between these two sets and thus fully capturing the information in this partition of observation instances. In other words, we seek a predicate that reduces as much as possible the partition's entropy.

The information gain of a predicate $P$ with respect to a set of instances is a measure that quantifies how much entropy is reduced by partitioning the set according to predicate $P$. A predicate that perfectly determines whether or not a given instance belongs to the set has an information gain equal to the entropy.

**Definition 3 (Information Gain).** *Let $\mathcal{I}'$ and $\mathcal{I}''$ be a set of observation instances for two tasks. The information gain of a predicate $P$ with respect to $\mathcal{I}'$ and $\mathcal{I}''$ is defined as follows:*[4]

$$IG(\mathcal{I}',\mathcal{I}'',P) = H(\mathcal{I}',\mathcal{I}'') - \frac{(|\mathcal{I}'_P|+|\mathcal{I}''_P|)\cdot H(\mathcal{I}'_P,\mathcal{I}''_P)}{|\mathcal{I}'|+|\mathcal{I}''|} - \frac{(|\mathcal{I}'_{\neg P}|+|\mathcal{I}''_{\neg P}|)\cdot H(\mathcal{I}'_{\neg P},\mathcal{I}''_{\neg P})}{|\mathcal{I}'|+|\mathcal{I}''|}$$

Algorithm 2 describes the technique to discover conjunctive branching conditions. Initially, we generate the observation instances through Algorithm 1. Afterwards, the algorithm iterates on each branching point $\{t_1, t_2\}$. Conditions $C(t_1)$ and $C(t_2)$ are computed by Daikon using the observation instances relative to tasks $\{t_1, t_2\}$. Then, function BUILDCONJUNTIVEEXPR is called to build a conjunctive condition by combining the invariants discovered by Daikon in a conjunction that maximizes the IG relative to the outcomes of the branching point. The conditions are then adjusted to ensure that $C(t_1) = \neg C(t_2)$.

| $X_1$ | $X_2$ | $X_3$ |
|---|---|---|
| 3 | 4 | true |
| 7 | 12 | true |
| 9 | 34 | true |
| 12 | 44 | true |

(a) $t_1$

| $X_1$ | $X_2$ | $X_3$ |
|---|---|---|
| 10 | 4 | true |
| 14 | 4 | true |
| 17 | 4 | true |
| 20 | 4 | true |

(b) $t_2$

**Fig. 2.** Examples of observation instances relative to two tasks $t_1$ and $t_2$ of a binary branching point.

Algorithm 3 shows how function BUILDCONJUNTIVEEXPR is implemented. In order to simplify the manipulation of conditions, we assume that the invariants discovered by Daikon are given as a conjunctive expression, i.e. $P = p_1 \wedge p_2 \wedge \ldots \wedge p_n$. Let $S = \{p_1, p_2, \ldots, p_n\}$ be the set of atoms of $P$. If $P$ is undefined (no invariant was discovered by Daikon) the algorithm returns $\perp$. Otherwise, the algorithm starts by picking the atom $\overline{q}$ with highest IG (line 6). This atom becomes the first conjunct in the result $\overline{P}$ (line 4). The algorithm continues by greedily adding a new atom $\overline{q}$ to the conjunctive expression $\overline{P}$ (line 9), provided that the conjunction $\overline{P} \wedge \overline{q}$ increases the IG. The loop stops when all the atoms in $P$ have been considered. The resulting expression $\overline{P}$ is then returned (line 13).

Once the conditions are built for the two branches, they are adjusted so as to ensure that $C(t_1) = \neg C(t_2)$ (function ADJUSTCONDITIONS). The adjustment is performed as follows. If Daikon was unable to discover $C(t_1)$ (or $C(t_2)$), we set $C(t_1) = \neg C(t_2)$ (or vice versa). Otherwise, if the IG of $C(t_1)$ is higher (resp. lower) than that of $C(t_2)$, we set $C(t_2) = \neg C(t_1)$ (resp. $C(t_1) = \neg C(t_2)$).

As an example, let us suppose to have an event log and a process model with a branching point $\{t_1, t_2\}$. Using the event log as input, we apply Algorithm 1 to build the observation instances $I'$, $I''$ relative to $t_1$ and $t_2$. Then, we employ Daikon with input $I'$ and input $I''$, thus discovering invariant $C(t_1)$ and

[4] Given a set $\mathcal{I}$ of observation instances and a predicate $P$, $\mathcal{I}_P$ and $\mathcal{I}_{\neg P}$ denote the sub set of instances of $\mathcal{I}$ for which predicate $P$ evaluates to true and to false, respectively.

**Algorithm 4:** DISCOVERDISJUNCTIVEEXPRESSIONWITHDAIKON (DD+IG)

```
Data: $A$ – A set of alignments, $P$ – Process Model
Result: A map that associates some transitions with the corresponding
        branching conditions
1  $I \leftarrow$ GenerateInstanceTuple($A$)
2  foreach $\{t_1, t_2\} \in$ BranchingPoints($P$) do
3      $C(t_1) \leftarrow false$
4      $C(t_2) \leftarrow false$
5      $DT \leftarrow$ BuildDecisionTree($I(t_1), I(t_2)$)
6      foreach $(t, \overline{I}) \in$ EnumeratePartitions($DT$) do
7          $J \leftarrow$ DiscoverInvariantsWithDaikon($\overline{I}$)
8          $J \leftarrow$ BuildConjuntiveExpr($I(t_1), I(t_2), J$)
9          $C(t) \leftarrow C(t) \lor J$
10     end
11     $C(t_1) \leftarrow$ BuildDisjuntiveExpr($I(t_1), I(t_2), C(t_1)$)
12     $C(t_2) \leftarrow$ BuildDisjuntiveExpr($I(t_1), I(t_2), C(t_2)$)
13     adjustConditions($I(t_1), I(t_2), C(t_1), C(t_2)$)
14 end
15 return C
```

$C(t_2)$ for $t_1$ and $t_2$, respectively. Daikon may discover the following invariants: $C(t_1)$ is $(x_1 < x_2 \land x_3 = true)$ and $C(t_2)$ is $(x_1 > 0 \land x_2 = 4 \land x_3 = true)$. Some atoms may be irrelevant as they do not discriminate the branch. E.g., to discover if any atom in $C(t_1)$ is irrelevant, we compute the IG of every atom: $IG(I', I'', (x_1 < x_2)) = 1$ and $IG(I', I'', (x_3 = true)) = 0$. We retain the atom with the highest IG, i.e. $x_1 < x_2$. Afterward, we pick $(x_3 = true)$; since $IG(I', I'', (x_1 < x_2 \land x_3 = true)) = IG(I', I'', (x_1 < x_2))$, we discard atom $(x_3 = true)$, so that $C(t_1)$ becomes the single atom $(x_1 < x_2)$. Similarly, atom $(x_3 = true)$ is discarded from $C(t_2)$: $C(t_2)$ is simplified as $(x_1 > 0 \land x_2 = 4)$. To finally obtain the branching conditions to associate with $t_1$ and $t_2$, we compute the IG of the simplified conditions $C(t_1)$ and $C(t_2)$: $IG(I', I'', C(t_1)) = 1$ and $IG(I', I'', C(t_2)) = 0.90$. Since the IG of $C(t_1)$ is higher, we set $C(t_2) = \neg C(t_1)$, i.e. $C(t_2)$ becomes $(x_1 \geq x_2)$.

## 3.2 Discovery of Disjunctive Conditions

Algorithm 4 (DD+IG) describes the technique for discovering disjunctive branching conditions. For each branching point $\{t_1, t_2\}$, we build a decision tree using the observation instances relative to tasks $t_1$ and $t_2$. In a decision tree, each path from the root to a leaf corresponds to an expression that is a conjunction of atoms of the form $v$ $op$ $c$. Such expressions are then used to partition the observation instances. In line 6, the invocation EnumeratePartitions(DT) returns a set of pairs, one for each leaf node of the decision tree. In particular, if a pair $(t, \overline{I})$ is in the set, there exists a tree path to a leaf node associated with a classification attribute value $t$ and $\overline{I}$ is the set of instances associated with that leaf. Note that there might be several leaves for each task $t$ ($t$ stands for $t1$ or $t2$). For each pair $(t, \overline{I})$, Daikon is used to discover the set of invariants $J$ for partition $\overline{I}$. From $J$ we build a conjunctive expression that maximizes the IG. The resulting conjunction is stored in $C(t)$. Once all partitions induced by the decision tree are analyzed, we proceed to combine the conjunctions into a disjunction. This is done by function BuildDisjuntiveExpr. The latter is similar to function BuildConjuntiveExpr except that atoms (in this case conjunctions) are combined in disjunctions, looking at maximizing the IG. Finally, we adjust $C(t_1)$ and $C(t_2)$ to ensure $C(t_1) = \neg C(t_2)$.

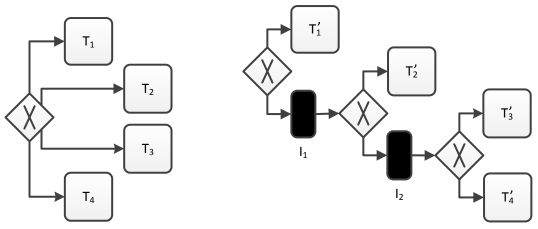
### 3.3 Extensions for Arithmetic Operators

Daikon includes invariant templates to discover linear equalities with 2 or 3 variables and an optional constant. However, there are no equivalent templates for inequalities. In process models, inequalities are common and leaving these aside is a major restriction. For instance, the running example involves inequalities with 2 variables and a constant ($length + age \leq 70$ and $salary/installment \geq 2$).

To cope with this limitation, we propose to enrich the original log with so-called *latent variables*. A latent variable is defined as a variable derived by combining multiple variables in the original log by means of one arithmetic operator ($+$, $-$, $*$ or $/$). In the running example, an example of a latent variable is "salary_div_by_installment" = salary/installment.

We extend CD+IG and DD+IG with latent variables as follows. We identify the set of numerical variables and generate all combinations of $N$ variables with one arithmetic operator (for each of the four arithmetic operators). Then, we augment the log by adding the latent variables and give it as input to Daikon. Daikon treats each latent variable as a regular variable. Thus, it discovers invariants involving one latent variable and one constant or one latent variable on one or both sides of an equality or inequality (e.g., $a + b \leq c * d$). The invariants thereby discovered are post-processed with either CD+IG and DD+IG. The extended techniques with latent variables are called CD+IG+LV and DD+IG+LV.

Importantly, invariants involving latent variables compete with invariants involving observed variables when CD+IG or DD+IG construct a branching condition out of the invariants returned by Daikon. Consider for example a situation where there are two numeric variables in the log ($x1$ and $x2$) and we seek to discover a branching condition $x1 \leq 8000 \wedge x2 \leq 8000$. Daikon naturally discovers invariant $x1 + x2 \leq 16000$ in addition to $x1 \leq 8000$ and $x2 \leq 8000$. Invariant $x1 + x2 \leq 16000$ may have a higher IG than each of the two other atoms taken separately. Thus, $x1 + x2 \leq 16000$ is integrated in the discovered condition and the other two invariants may then be left out if they do not increase the RIG. We observed this behavior when conducting preliminary tests. Accordingly, we adopt a two-step approach. In the case of CD+IG+LV, first, CD+IG (without latent variables) is run. If the result is not satisfactory (i.e., RIG below a threshold), CD+IG+LV is run again with all latent variables involving $N$ terms ($N$ is a tunable parameter). The same applies for DD+IG+LV.



(a) (b)

**Fig. 3.** N-ary to binary transformation

The complexity of CD+IG+LV and DD+IG+LV is combinatorial on $N$, since one latent variable is generated for each subset of size $N$ of the variables in the log, and for each arithmetic operator. Thus these techniques are practical only for small values of $N$. Another limitation of CD+IG+LV and DD+IG+LV is that they only discover equalities or inequalities where each side involves a single type of arithmetic operator (only $+$ or $-$ or $*$ or $/$). Introducing latent variables combining multiple types of arithmetic operators would lead to a higher combinatorial explosion when $N > 2$.

## 3.4 Extension to N-ary Branching Points

Hitherto, we have assumed that every branching point is binary. The technique can be extended to N-ary branching points as follows. Given an N-ary branching point, we rewrite this point into a number of of binary branching points by leaving one branch intact, collapsing the remaining $N - 1$ branches into a separate branching point and so on recursively. For instance, the quaternary branching point in Figure 3(a) is rewritten into binary branching points in Figure 3(b). The transformed model has 2 new (black-filled) tasks ($I_1$ and $I_2$). These dummy ($\tau$) tasks are introduced purely for the purpose of the branching condition discovery.

Any of the above techniques (CD+IG, DD+IG or their extensions with latent variables) can be applied to each binary branching point using the extended log. In the example, this allows us to discover the 6 conditions $C(t) - t \in \{T'_1, T'_2, T'_3, T'_4, I_1, I_2\}$. Having discovered the conditions for each binary branching point, the branching condition $C(T_i)$ of the $i^{th}$ branch of the N-ary branching point is then defined as the conjunction of $C(T'_i)$ and each of the $C(I_j)$ where task $I_j$ is on the path from the first binary branching point to $T'_i$ in the rewritten model. In the example, this means that: $C(T_1) = C(T'_1)$, $C(T_2) = C'(I_1) \wedge C(T'_2)$, $C(T_3) = C'(I_1) \wedge C'(I_2) \wedge C(T'_3)$ and $C(T_4) = C'(I_1) \wedge C'(I_2) \wedge C(T'_4)$.

An N-ary branching point can be rewritten into binary ones in multiple ways depending on the order in which the N branches are refactored. Each rewriting leads to different branching conditions. Since we seek to maximize information gain, we perform the rewriting as follows. First we run CD+IG (or an extension) on each of the N original branches. We then select the branch for which the discovered condition has the highest Relative Information Gain (RIG). The RIG of a branching condition is the IG of the condition divided by the entropy of the observation instances of the branch in question and the union of the observation instances of all other branches. RIG is equal to 1 when the IG is equal to the entropy. This normalization of information gain relative to the entropy allows us to compare the gain of conditions in different branches (which may have different entropies). Having selected the branch with the highest RIG, we refactor this branch and apply the procedure recursively on the remaining branches.

## 4 Evaluation

The proposed techniques have been prototyped in Java using Daikon[5] for invariant detection and Weka[6] for decision tree learning. The prototype and the testbed presented below are available at `http://sep.cs.ut.ee/Main/BranchMiner`.

### 4.1 Testbed

We designed a battery of test cases covering different types of conditions. Daikon supports three primitive data types (integer, float and string) and sequences these primitive types. The testbed includes branching conditions with integers and strings. Strings are used to encode categorical (unordered) domains (i.e. enumerated types). Floats are not included in the testbed because Daikon handles integers and floats in the same way, and thus testing for both is redundant. We also left out sequences, because we consider they deserve a separate study.

The testbed includes conditions composed of atoms including variables with either categorical domain or numerical domain as follows. We defined 3 variables ($c1$, $c2$ and $c2p$) with categorical domains. Each domain includes 3 values: **C11**, **C12**. **C13** for $c1$ and **C21**, **C22**. **C23** for $c2$ and $c2p$. Since categorical domains are treated as unordered, we created atoms of the form variable-equals-constant and variable-equals-variable over the 3 variables. Thus two types of atoms were defined for categorical domains. We defined 4 variables ($x1$ to $x4$) over a numerical domain ($[1000, 15000]$)[7]. With these variables, we created atoms of the form variable-operator-variable and variable-operator-constant, where the operator can be $=$, $\leq$ and $\geq$. We did not produce atoms for operators $<$ and $>$ because these operators appear anyway in the negations of $\leq$ and $\geq$ and each test case includes a condition and its negation. Thus 3 types of atoms are defined over numerical variables. Test cases for $\geq$ and $=$ are omitted for the sake of brevity.

Given these atom types, we designed test cases covering 4 types of expressions: (i) single-atom; (ii) conjunctions of two atoms; (iii) disjunctions of two atoms; and (iv) disjunction of a conjunction and an atom. These test case design principles led us to 6 test cases for categorical domains and 15 for numerical domains of which only 5 are shown below for brevity. To test branching conditions with arithmetic operators, we introduced 2 additional variables ($x5$ and $x6$) and 3

[5] `http://groups.csail.mit.edu/pag/daikon/dist/`

[6] `http://www.cs.waikato.ac.nz/ml/weka/`

[7] Daikon was configured to discover invariants of upper and lower bound for numerical variables in the range of their corresponding domain, the default being $[-1 \ldots 2]$.

additional cases (one single-atom, one disjunctive and one conjunctive) containing atoms with $\leq$, $>$ and $\geq$. The test cases are presented in Tables 1 and 2.

For each test case, we generated an event log of 200 execution traces via simulation using CPNTools[8]. The simulation model is a Coloured Petri net with three transitions. The first transition randomly assigns a value to each of the 9 variables ($c1$, $c2$, $c2p$, $x1$-$x6$) according to a uniform distribution. The other two transitions correspond to the branches of a branching point. One of these two transitions is labelled with the branching condition corresponding to the test case and the other with its negation. In the case of conditions $x_i = c$ and $x_i = x_j$ where $x_i$ and $x_j$ are numeric variables, we adjusted the random assignment so that these conditions hold in 50% of the cases. If we simply used a uniform distribution for these variables the probability of $x_i = c$ would be too low to generate enough traces that take the corresponding branch.

## 4.2 Results

**Branching conditions without arithmetic operators.** Table 1 presents the original and the discovered conditions for the test cases without arithmetic operators. The table also shows the RIG (cf. Section 3.4) for each discovered condition. We observe that DD+IG discovered each of the original conditions in this category and the corresponding RIG is exactly one, indicating that the discovered conditions have perfect discriminative power. Meanwhile, CD+IG failed in case 11 with very low RIG, discovered alternative (equivalent) conditions in cases 3 and 6, and a similar (non-equivalent) condition in case 9. In these three latter cases, the original condition included a disjunction of atoms, which is equivalent to a conjunction of negations of the original atoms. This conjunction is discovered by the conjunctive approach with the caveat that the negated atoms involve the duals of the comparison operators of the original conditions. Thus, $c2 = c2p$ is discovered as $\neg(c2 \neq c2p)$ in case 6, while $c1 = \mathbf{C12}$ is discovered as $\neg(c1 \in \{\mathbf{C11}, \mathbf{C13}\})$. For case 9, atom $x1 \leq 8000$ was discovered as $\neg(x1 \geq 9000)$ because Daikon does not discover invariants of the form "$x < C$" but instead it finds invariants with $\leq$. CD+IG failed in case 11 because both the original expression and its dual contain a disjunction.

**Branching conditions with arithmetic operators.** Table 2 show the results of test cases for branching conditions with arithmetic operators. In all three cases, CD+IG+LV and DD+IG+LV succeeded to discover either the original condition, an alternative (equivalent) one or a similar (non-equivalent) one. In spite of having a RIG of 1.0, the solution to case 14 given by DD+IG+LV is unnecessarily elaborated, in addition to being non-equivalent to the original condition. It is clear that the presence of two disjoints comes from the partitioning induced by the decision tree used in Algorithm 4. Not shown in the table is that CD+IG and DD+IG (without latent variables) failed in all these cases, as expected. They returned conditions with very low RIG.

[8] `http://cpntools.org`

| Case | Original | CD+IG | RIG | DD+IG | RIG |
|---|---|---|---|---|---|
| 1 | $c1=\mathbf{C12}$ | $c1=\mathbf{C12}$ | 1.0 | $c1=\mathbf{C12}$ | 1.0 |
| 2 | $c1=\mathbf{C12} \wedge c2=\mathbf{C22}$ | $c1=\mathbf{C12} \wedge c2=\mathbf{C22}$ | 1.0 | $c1=\mathbf{C12} \wedge c2=\mathbf{C22}$ | 1.0 |
| 3 | $c1=\mathbf{C12} \vee c2=\mathbf{C22}$ | $\neg(c2 \in \{\mathbf{C23},\mathbf{C21}\} \wedge c1 \in \{\mathbf{C11},\mathbf{C13}\})$ | 1.0 | $c2=\mathbf{C22} \vee c1=\mathbf{C12}$ | 1.0 |
| 4 | $c2=c2p$ | $c2=c2p$ | 1.0 | $c2=c2p$ | 1.0 |
| 5 | $c2=c2p \wedge c1=\mathbf{C12}$ | $c2=c2p \wedge c1=\mathbf{C12}$ | 1.0 | $c1=\mathbf{C12} \wedge c2=c2p$ | 1.0 |
| 6 | $c2=c2p \vee c1=\mathbf{C12}$ | $\neg(c2 \neq c2p \wedge c1 \in \{\mathbf{C11},\mathbf{C13}\})$ | 1.0 | $c2=c2p \vee c1=\mathbf{C12}$ | 1.0 |
| 7 | $x1 \leq 8000$ | $x1 \leq 8000$ | 1.0 | $x1 \leq 8000$ | 1.0 |
| 8 | $x1 \leq 8000 \wedge x2 \leq 8000$ | $x2 \leq 8000 \wedge x1 \leq 8000$ | 1.0 | $x2 \leq 8000 \wedge x1 \leq 8000$ | 1.0 |
| 9 | $x1 \leq 8000 \vee x2 \leq 8000$ | $\neg(x2 \geq 9000 \wedge x1 \geq 9000)$ | 1.0 | $x2 \leq 8000 \vee x1 \leq 8000$ | 1.0 |
| 10 | $x1 \leq x2$ | $x1 \leq x2$ | 1.0 | $x1 \leq x2$ | 1.0 |
| 11 | $x1 \leq 8000 \vee c2=\mathbf{C22} \wedge x3 \leq x4$ | $\underline{\neg(\mathbf{x1} \geq \mathbf{9000})}$ | $\underline{\mathbf{0.39}}$ | $x1 \leq 8000 \vee c2=\mathbf{C22} \wedge x3 \leq x4$ | 1.0 |

**Table 1.** Test suite with no arithmetic operator.

| Case | Original | CD+IG+LV | RIG | DD+IG+LV | RIG |
|---|---|---|---|---|---|
| 12 | $x1 \leq x2 \wedge x3+x4 > 15000$ | $x1 \leq x2 \wedge x3+x4 \geq 16000$ | 1.0 | $x1 \leq x2 \wedge x3+x4 \geq 16000$ | 1.0 |
| 13 | $x1 \leq x2 \vee x3+x4 > 15000$ | $\neg(x1 > x2 \wedge x3+x4 \leq 15000)$ | 1.0 | $\neg(x1 > x2 \wedge x3+x4 \leq 15000)$ | 1.0 |
| 14 | $x5 * x6 \geq 49$ | $x5 * x6 \geq 49$ | 1.0 | $x5 * x6 \geq 50 \vee x5 * x6 \in [49 \ldots 56]$ | 1.0 |

**Table 2.** Test suite with arithmetic operators.

**Execution times.** The experiments were conducted on a laptop, using a Java VM 1.7 on a 64 bit operating system. To gather the execution times, we ran every test case five times and took the average time of these runs. For the Conjunctive approach, the times had an average of 2 s, with 0.1 s of standard deviation, and a maximum of 2.18 s. For the Disjunctive approach, the average execution times was 11.2 s, with 0.8 s of standard deviation, and a maximum of 35.8 s.

**Discussion.** The results show that the proposed techniques have increased levels of precision (pecentage of correctly discovered conditions) but also decreasing performance. The results also show that the RIG of a discovered condition is a useful indicator of whether or not this condition has sufficient discriminative power. Thus, the techniques can be applied in a "trial-and-error" manner. Given a branching point, we can first apply CD+IG. If the RIG is less than a given threshold, we can apply DD+IG. If the resulting RIG is still low, we can introduce latent variables with the aim of maximizing the RIG of the discovered condition.

## 5 Conclusion

We have shown that a combination of invariant detection and decision tree learning techniques allow us to discover a wide spectrum of branching conditions from business process execution logs, thereby allowing us to enhance the output of existing automated process discovery techniques. Specifically, we proposed three branching condition discovery techniques of increased level of complexity.

The proposed techniques have been validated on synthetically-generated logs covering branching conditions with varying structures. The test results show that the techniques discover non-trivial branching conditions in a few seconds (for the simpler technique) and in less than a minute for the more complex technique. In the future, we plan to apply the proposed techniques in practice on real-life event logs for example in the field of insurance where complex decisions are often involved when classifying claims, and similarly in the field of healthcare.

The approach to discover inequalities with more than two variables suffers from two key limitations. First, it is only possible to discover inequalities where each side has at most $N$ variables (with $N$ fixed) and where only one type of arithmetic operator appears in each side of the inequality. Second, the complexity of the approach increases combinatorially with $N$. Recent work [10] has put forward a technique to discover invariants consisting of equalities and inequalities among nonlinear polynomials. Adapting this technique for branching condition discovery is a direction for future work.

**Acknowledgments.** Work supported by EU's FP7 program (ACSI project).

## References

1. van der Aalst, W.M.P.: Process Mining - Discovery, Conformance and Enhancement of Business Processes. Springer (2011)
2. Rozinat, A., van der Aalst, W.M.P.: Decision mining in ProM. In: Proc. of BPM'2006, Springer-Verlag (2006) 420–425
3. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming **69** (2007) 35–45
4. de Leoni, M., van der Aalst, W.M.P.: Discovery of data-aware process models. In: Proc. of the 28th ACM symposium on Applied Computing (SAC'13), ACM (2013)
5. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically Discovering Likely Program Invariants to Support Program Evolution. IEEE Trans. Software Eng. **27** (2001) 99–123
6. Guo, P.J., Perkins, J.H., McCamant, S., Ernst, M.D.: Dynamic inference of abstract types. In: Proc. of ISSTA'2006, ACM (2006) 255–265
7. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: Proc. of ICSE'2008, IEEE (2008) 501–510
8. Lo, D., Maoz, S.: Scenario-based and value-based specification mining: better together. Autom. Softw. Eng. **19** (2012) 423–458
9. Kuzmina, N., Paul, J., Gamboa, R., Caldwell, J.: Extending dynamic constraint detection with disjunctive constraints. In: Proc.of WODA'2008, ACM (2008) 57–63
10. Nguyen, T., Kapur, D., Weimer, W., Forrest, S.: Using dynamic analysis to discover polynomial and array invariants. In: Proc. of ICSE'2012, IEEE (2012) 683–693
11. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: Proc. of POLP'2002, ACM (2002) 4–16
12. Shoham, S., Yahav, E., Fink, S., Pistoia, M.: Static specification mining using automata-based abstractions. In: Proc. of ISSTA'2007, ACM (2007) 174–184
13. Lo, D., Khoo, S.C., Han, J., Liu, C., eds.: Mining Software Specifications: Methodologies and applications. CRC Press (2011)