Generating Benchmarks by Random Stepwise Refinement of Petri Nets

Kees M. van Hee and Zheng Liu

Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{k.m.v.hee, z.liu3}@tue.nl

Abstract. The quality of algorithms is often determined by benchmarking, i.e., testing the algorithm on a predetermined data set. In contrast to traditional benchmarking, with fixed data set, we present a way to generate random sets of test data. In this paper we present random classes of Petri nets and a method to generate finite samples from such a class. The classes may contain infinitely many Petri nets, each net with its own probability to be generated. This generation method is based on stepwise application of construction rules such as refinement rules. Each random class of Petri nets has a probability distribution for each of its characteristics. We illustrate the approach by estimating this distribution for some simple characteristics.
Keywords: Petri nets, benchmarking, random graphs

1 Introduction

In computer science research we often lack a method to evaluate the *quality* of an algorithm in an analytical way. The quality may concern the *efficiency* of the algorithm (how fast is a solution found) or the *effectiveness* (how good is the solution found). Although it is sometimes possible to give bounds for the worst case behavior, it is seldom possible to determine the average-case behavior analytically. What we normally do is fixing a set of test data as *benchmark* and then we test the algorithm on that set, which is called *benchmarking*. Generally a benchmark is either pre-defined or generated. Both have disadvantages [15]. Pre-defined datasets are designed to be representative examples in a particular domain. Researchers evaluate new algorithms with respect to such a fixed benchmark, but how good are they if applied to other test data? In order to increase the quality of benchmarking we will consider methods to generate *random benchmarks*, which are *samples* from an infinite set of tests, each having its own probability of being selected for a sample. Note that it is impossible to have an infinite set of tests each having the same probability, so the probabilities of the tests have to be non-uniform. Due to the fast increasing computing power we are able to generate samples that are so big that all non-selected tests together have a probability below some chosen bound. This allows us to obtain *statistical* statements of the quality of an algorithm.

In this paper we focus on Petri nets and algorithms to compute their characteristic properties. Hence we define *random classes* of Petri nets. Such a class contains Petri nets with some structural property, like free choice nets. There may be infinite number of nets in one class. Each net in a class has its own probability of being selected for a benchmark. Petri nets are used to model complex processes [11], for example computational processes in a computer system or business processes within or between enterprises. These models are often produced by a stepwise refinement process (top-down approach) or by gluing together existing components (bottom-up approach). The *generation method* uses two kinds of construction rules, *refinement rules* and *bridge rules*. The refinement rules were firstly studied by Berthelot in [5] and Murata in [18] as reduction rules, in this paper we use them in the inverse direction to expand Petri nets by refinement. The bridge rules connect a pair of nodes in a Petri net, so we can use them to glue components. In most cases these construction rules preserve some property which means that if the initial Petri net has that property, then all elements of the class have the same property. The generation method determines, in a random way, (1) which construction rule will be applied in the current Petri net, (2) to which part of the net and (3)if we continue or stop. So the construction rules have weights. Rephrasing what we are doing, we define a *graph grammar* with weights on the *construction rules*, such that each graph of the graph language has a certain probability of occurring.

Although our generation method for benchmarks can be applied to all kinds of algorithms on Petri nets, we use it here to determine *characteristics* of the random classes of Petri nets. Simple examples of such characteristics are the number of nodes, and the average fanin and fanout of nodes. More complicated ones are the occurrence (or the number) of deadlocks or livelocks. Each characteristic of a Petri net is expressed by a real number. In some cases we consider only 0 and 1 and we consider them as 'false' and 'true'. Since every Petri net in a class has a certain probability of being selected in a benchmark, we may consider every characteristic as a random variable having a *probability distribution* over the class. For the given examples, we can speak of the *expectation* and *variance* of the number of nodes in a class, or the *probability* of having a deadlock. We have developed a software tool in the form of a plugin for ProM (cf [16]) to realize our approach.

The rest of this paper is organized as follows. Section 2 introduces the necessary preliminaries. In Section 3 we represent our construction rules and discuss our methodology of generating Petri nets. Section 4 presents some characteristics. The software tool is introduced in Section 5 with characteristic examples. Related work is discussed in Section 6. Finally Section 7 concludes this paper and discusses some of our future researches on identifying class parameters.

2 Preliminaries

Let S be a set. With $|S|$ we denote the number of elements in S. The empty set, e.g., the set without any elements is denoted by $\emptyset$. Two sets S and R are disjoint if $S \cap R = \emptyset$. We denote the set of all natural numbers as $\mathbb{N} = \{0, 1, 2, \cdots\}$. A sequence σ of length $l \in \mathbb{N}$ over S is a function $\sigma : \{1, \cdots, l\} \rightarrow S$. We denote a sequence by $\sigma =< \sigma(1), \sigma(2), \cdots, \sigma(l) >$, such that $\forall i (1 \leq i < l) : \sigma(i) \in \bullet\sigma(i+1)$. We write here this as $n_1 \xrightarrow{\sigma} n_l$. We denote the length of a sequence by $|\sigma|$. The set of all finite sequences over S is denoted as Σ. Let $\nu, \gamma \in \Sigma$ be two sequences. *Concatenation*, denoted by $\sigma = \nu \circ \gamma$, is defined as $\sigma : \{1, \cdots, |\nu| + |\gamma|\} \rightarrow S$, such that for $1 \leq i \leq |\nu| : \sigma(i) = \nu(i)$, and for $|\nu| + 1 \leq i \leq |\nu| + |\gamma| : \sigma(i) = \gamma(i - |\nu|)$. The Parikh vector of a sequence σ, denoted by $\overrightarrow{\sigma}$, is a bag representing the number of occurrences of each element in σ. A *bag m (multiset)* over S is a function $m : S \rightarrow \mathbb{N}$. For $s \in S$, $m(s)$ denotes the number of occurrences of s in m. We denote a bay by square brackets. e.g., in a bag $[a, b^2, c]$, element a occurs once, element b twice, and element c once. All other elements have a multiplicity of 0. $\prec$ is the prefix operator, such that $\sigma^{'} \prec \sigma$ if and only if $\exists \sigma^{''} : \sigma = \sigma_{'} \circ \sigma_{''}$.

A Petri net is a tuple $N = (P, T, F)$ where P is the set of *places*, T is the set of *transitions*, P and T are disjoint, and $F \subseteq (P \times T) \cup (T \times P)$ is the set of arcs. An element of $P \cup T$ is called a node. We call an element of $P \cup T \cup F$ is an element of N. A *path* in N is a sequence σ over the set $P \cup T$. Graphically, we denote places by circles, transitions by squares, and arcs as arrows between places and transitions. The *state* of a Petri net, called a *marking* is a bag over the places P of N. A marking is graphically represented by placing *tokens* in each place. A marked Petri net is a pair (N, m_0), where N is a Petri net and m_0 is a marking of N. A transition $t \in T$ is enabled in (N, m_0), denoted by $(N : m_0 \xrightarrow{t})$ if $\bullet t \leq m_0$. An enabled transition in (N, m_0) can *fire* resulting in a new marking $m^{'} = m_0 - \bullet t + t\bullet$, denoted by $(N : m_0 \xrightarrow{t} m^{'})$.

A special class of Petri nets are *workflow nets.* A workflow net is a 5-tuple $W = (P, T, F, i, f)$ where (P, T, F) is a Petri net, $i \in P$ is the initial place, such that $\bullet i = \emptyset$, $f \in P$ is the final place, such that $f\bullet = \emptyset$, in graph of W each node $n \in P \cup T$ is on a directed path from i to f. If for a workflow net W, we have $\forall p \in P \setminus \{i, f\}, |p\bullet| = |\bullet p| = 1, |i\bullet| = |\bullet f| = 1$, the workflow net is a *T-net*, also called a *marked graph* workflow net. If $\forall t \in T, |t\bullet| = |\bullet t| = 1$, then it is a *S-net*, also called a *state machine* workflow net. In a workflow net, two places $p, s \in P$, if either $p\bullet = s\bullet$ or $p\bullet \cap s\bullet = \emptyset$, then this workflow net is a *free-choice* workflow net. A *firing sequence* or a *trace* is a sequence σ over T such that all the transitions of σ can fire in that order starting from the initial marking. A trace for a workflow net is complete if it leads to the final marking with only f marked.

A workflow net is k sound, for $k \in \mathbb{N}$ if for each marking m that is reachable from an initial marking m_0 with only k tokens in the initial place i, the final marking with only k tokens in the final place f can be reached. A workflow net is *generalized* sound if it is k-sound for all $k \geq 1$ ([14]). Note that *1-sound* is usually called *sound* ([1]). Soundness can be considered as a general sanity check for workflow nets.

3 Net Generation

In this section we firstly define the construction rules. The rules enable us to generate all Petri nets, here we focus upon workflow nets. Moreover, we show that different subclasses of workflow nets can be generated by different construction rules. Finally, we discuss our method of randomly generating Petri nets.

3.1 Construction Rules

Based upon how they can modify the structure of Petri nets, the construction rules are divided into two classes, *refinement* rules and *bridge* rules. The refinement rules ([5], [8], [12], and [18]) were firstly studied by Berthelot and Murata as abstraction rules to reduce Petri nets, here we use them in the inverse direction to expand Petri nets. The bridge rules connect a pair of nodes in Petri nets, so we can use them to glue components. Let N be the original net and R be one of the construction rules. If we apply R to N then we get the generated net $N^{'}$. If we use R in the opposite direction, then we can get N again by reducing $N^{'}$. In such a case we say $(N, N^{'}) \in \varphi_R$. Based upon such a relationship, we define the rules as follows.

Refinement rules. Figure 1 is an example of the refinement rules.

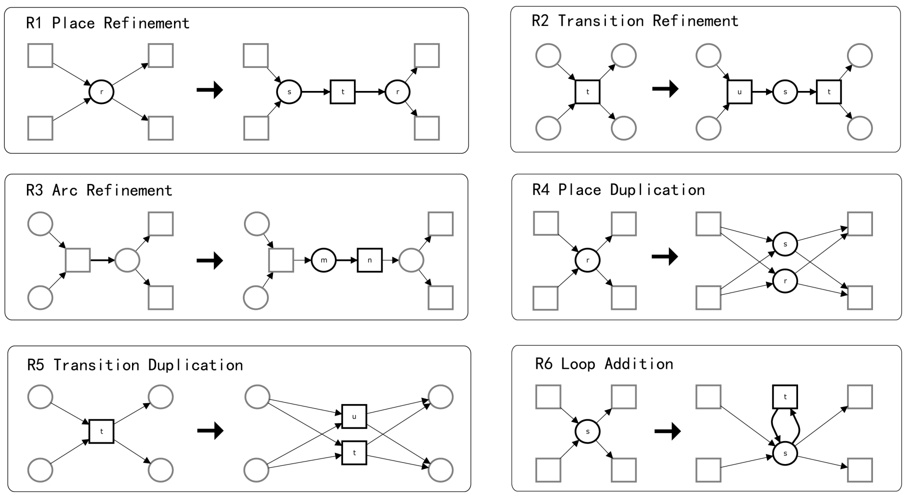


Fig. 1. Refinement Rules

Definition 1 (Place refinement rule R1). *Let $N = (P, T, F)$ and $N^{'} = (P^{'}, T^{'}, F^{'})$ be two Petri nets. We say $(N, N^{'}) \in \varphi_{R_1}$ if and only if there exist places $s, r \in P^{'}, s \neq r$ and a transition $t \in T^{'}$ such that: $\bullet t = \{s\}, t\bullet = \{r\}$, $s\bullet = \{t\}, \bullet s \neq \emptyset$, $\bullet s \not\subseteq \bullet r$. The net N satisfies: $P = P^{'} \setminus \{s\}$, $T = T^{'} \setminus \{t\}$, $F = (F^{'} \cap ((P \times T) \cup (T \times P))) \cup (\bullet s \times t\bullet)$.*

Definition 2 (Transition refinement rule R2). *Let $N = (P, T, F)$ and $N^{'} = (P^{'}, T^{'}, F^{'})$ be two Petri nets. We say $(N, N^{'}) \in \varphi_{R_2}$ if and only if there exist a place $s \in P^{'}$ and transitions $t, u \in T^{'}, t \neq u$ such that: $\bullet s = \{u\}, s\bullet = \{t\}$, $\bullet t = \{s\}, t\bullet \neq \emptyset$, $u\bullet \not\subseteq t\bullet$. The net N satisfies: $P = P^{'} \setminus \{s\}$, $T = T^{'} \setminus \{u\}$, $F = (F^{'} \cap ((P \times T) \cup (T \times P))) \cup (\bullet u \times s\bullet)$.*

Definition 3 (Arc refinement rule R3). *Let $N = (P, T, F)$ and $N^{'} = (P^{'}, T^{'}, F^{'})$ be two Petri nets. We say $(N, N^{'}) \in \varphi_{R_3}$ if and only if there exist two nodes $m, n \in P^{'} \cup T^{'}$, such that: $| \bullet m| = 1$, $m\bullet = \{n\}$, $|n \bullet | = 1$, $\bullet n = \{m\}$, $(\bullet m \times n\bullet) \cap F^{'} = \emptyset$. The net N satisfies: $P \cup T = (P^{'} \cup T^{'}) \setminus \{m, n\}$, $F = (F^{'} \cap ((P \times T) \cup (T \times P))) \cup (\bullet m \times n\bullet)$.*

Note that in Figure 1, R3 shows only one instance of the arc refinement rule. We can also refine any arc with a place as its source node and a transition as its target node. This case is not shown in Fig 1.

Definition 4 (Place duplication rule R4). *Let $N = (P, T, F)$ and $N^{'} = (P^{'}, T^{'}, F^{'})$ be two Petri nets. We say $(N, N^{'}) \in \varphi_{R_4}$ if and only if there exist two places $s, r \in P^{'}, s \neq r$ such that: $\bullet s = \bullet r$, $s\bullet = r\bullet$. The net N satisfies: $P = P^{'} \setminus \{s\}$, $T = T^{'}$, $F = F^{'} \cap ((P \times T) \cup (T \times P))$.*

Definition 5 (Transition duplication rule R5). *Let $N = (P, T, F)$ and $N^{'} = (P^{'}, T^{'}, F^{'})$ be two Petri nets. We say $(N, N^{'}) \in \varphi_{R_5}$ if and only if there exist two transitions $t, u \in T^{'}, t \neq u$ such that: $\bullet t = \bullet u$, $t\bullet = u\bullet$. The net $P = P^{'}$, $T = T^{'} \setminus \{u\}$, $F = F^{'} \cap ((P \times T) \cup (T \times P))$.*

Definition 6 (Loop addition rule R6). *Let $N = (P, T, F)$ and $N^{'} = (P^{'}, T^{'}, F^{'})$ be two Petri nets. We say $(N, N^{'}) \in \varphi_{R_6}$ if and only if there exists a place $s \in P^{'}$ and a transition $t \in T^{'}$ such that: $\bullet t = \{s\}$, $t\bullet = \{s\}$. The net N satisfies: $P = P^{'}$, $T = T^{'} \setminus \{t\}$, $F = F^{'} \cap ((P \times T) \cup (T \times P))$.*

Bridge rules. Figure 2 is an example of the bridge rules.

Definition 7 (Place bridge rule R7). *Let $N = (P, T, F)$ and $N^{'} = (P^{'}, T^{'}, F^{'})$ be two Petri nets. We say $(N, N^{'}) \in \varphi_{R_7}$ if and only if there exist one place $s \in P^{'}$ and two transitions $u, t \in T^{'}$ such that: $\bullet s = \{u\}$, $s\bullet = \{t\}$. The net N satisfies: $P = P^{'} \setminus \{s\}$, $T = T^{'}$, $F = F^{'} \cap ((P \times T) \cup (T \times P))$.*

Definition 8 (Transition bridge rule R8). *Let $N = (P, T, F)$ and $N^{'} = (P^{'}, T^{'}, F^{'})$ be two Petri nets. We say $(N, N^{'}) \in \varphi_{R_8}$ if and only if there exist one transition $t \in T^{'}$ and two places $s, r \in P^{'}$ such that: $\bullet t = \{s\}$, $t\bullet = \{r\}$. The net N satisfies: $P = P^{'}$, $T = T^{'} \setminus \{t\}$, $F = F^{'} \cap ((P \times T) \cup (T \times P))$.*

Definition 9 (Arc bridge rule R9). *Let $N = (P, T, F)$ and $N^{'} = (P^{'}, T^{'}, F^{'})$ be two Petri nets. We say $(N, N^{'}) \in \varphi_{R_9}$ if and only if there exist two nodes $s, r \in P^{'} \cup T^{'}$, such that $(s, r) \in F^{'}$. The net N satisfies: $P = P^{'}$, $T = T^{'}$, $F = F^{'} \setminus \{(s, r)\}$.*

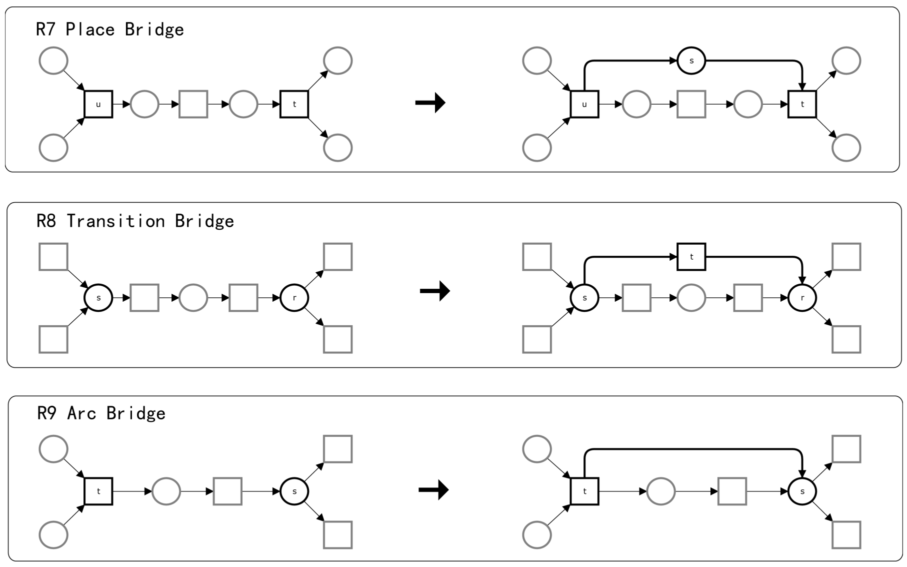


Fig. 2. Bridge Rules

Note that in Figure 2, R9 merely shows one instance of the arc bridge rule. We can also bridge a place and a transition with an arc. This case is not shown in Figure 2.

In R8, if s and r are the same place, it becomes R6. Thus the loop addition rule R6 is a special case of the transition bridge rule R8. All the refinement rules (R1,...,R6) preserve *liveness* and *boundedness* properties of Petri nets (with respect to a given marking) (proofs can be found in [12]). Although without formal proof, it is still easy to observe that all the bridge rules (R7,...,R9) generally do not preserve those properties. In fact, the bridge rules can break any good structures. For instance, the transition bridge rule can model the *goto* structure in programming languages, and the danger of such a structure is discussed in [9].

3.2 Structural Classes of Generated Nets

In this part we study different classes of the generated Petri nets based on their structures. We focus on workflow nets and their subclasses, namely, *Jackson nets*, *state machine* workflow nets, *marked graph* workflow nets, and *free-choice* workflow nets. These workflow nets without Jackson nets are defined in Section 2, so firstly we define a Jackson net as follows. For a formal description and analysis of Jackson nets see [12].

Definition 10 (Jackson net). *A Jackson net is a workflow net that can be generated, from one single place, by applying the rules* $R1$*,* $R2$*,* $R4$*,* $R5$*, and* $R6$ *recursively. However, only the rule* $R1$ *can be applied in the first step, and no rules can be applied to the initial place and the final place after the first step.*

In order to generate all workflow nets (starting with two places), it is sufficient to use the arc refinement rule R3 and the bridge rules R7, R8, R9 only. To prove that, we give the following definition.

Definition 11 (Copy of a net). *Let N be a workflow net. $N^{'}$ is a copy of N if and only if there is a bijection φ such that $\varphi(P) = P^{'}$, $\varphi(T) = T^{'}$, $\varphi(F) = F^{'}$ and $\forall(x, y) \in F : \varphi((x, y)) = (\varphi(x), \varphi(y))$.*

A path taken from a Petri net defines a new Petri net, so copying a path can be treated as copying its corresponding Petri net.

Lemma 1. *Given two nodes, m_1, $m_2 \in P \cup T$, and a path σ, such that $m_1 = \sigma(1)$, $m_2 = \sigma(|\sigma|)$, and $\forall i \in dom(\sigma) : \overrightarrow{\sigma}(i) = 1$. Then we can copy σ by firstly applying a bridge rule on (m_1, m_2) and afterwards repeating the arc refinement rule.*

Lemma 2. *We have a path σ in a Petri net, if there is some node n such that $\overrightarrow{\sigma}(n) > 1$, then there is a path $\sigma^{'} = \sigma_1 \circ \sigma_3$ if and only if $\sigma = \sigma_1 \circ \sigma_2 \circ \sigma_3$ and $n \xrightarrow{\sigma_2} n$.*

Theorem 1. *Let N be a workflow net. Then N can be copied into $N^{'}$ by only using the bridge rules and the arc refinement rule, starting with only two places.*

Proof. Initially $P^{'} = \{i^{'}, f^{'}\}$, $T^{'} = \emptyset$, $F^{'} = \emptyset$, $i^{'} = \varphi(i)$, and $f^{'} = \varphi(f)$. We select an arbitrary node n from N such that $n \in P \cup T$ and $n \notin P^{'} \cup T^{'}$. In N, from n we find two paths, $n_1 \xrightarrow{\sigma_1} n$ and $n \xrightarrow{\sigma_2} n_2$, such that $n_1, n_2 \in P^{'} \cup T^{'}$ and all other nodes on σ_1 and σ_2 are not in $P^{'} \cup T^{'}$. This is always possible by the definition of a workflow net. By Lemma 2 we can reduce σ_1 and σ_2 by removing all internal loops along the paths.

Consider case (1): suppose σ_1 and σ_2 have no common internal nodes, then we may apply Lemma 1 by adding the path $\sigma_1 \circ \sigma_2$ from n_1 to n_2, and add $n^{'} = \varphi(n)$ to the copy.

Consider case (2): suppose σ_1 and σ_2 have common internal nodes, e.g., $\exists i, j : \sigma_1(i) = \sigma_2(j) \neq n$, then there is a loop. We then look for the first node m on σ_1 which is also on σ_2. Then $\exists \sigma_3, \sigma_4, \sigma_5, \sigma_6 : n_1 \xrightarrow{\sigma_3} m \xrightarrow{\sigma_4} n \wedge n \xrightarrow{\sigma_5} m \xrightarrow{\sigma_6} n_2$ where $\sigma_1 = \sigma_3 \circ \sigma_4$ and $\sigma_2 = \sigma_5 \circ \sigma_6$. Now we have found $n_1 \xrightarrow{\sigma_3} m$ and $m \xrightarrow{\sigma_6} n_2$, and σ_3 and σ_6 have no internal nodes in common as m is the first common node on σ_1. We replace n with m, then we apply Lemma 1 by adding the path $\sigma_3 \circ \sigma_6$ from n_1 to n_2, and we add node $m^{'} = \varphi(m)$ to the copy.

In each step, therefore, we add at least one node. We repeat this procedure until all the nodes of N are copied into $N^{'}$. Finally, we add all arcs $F \setminus \varphi^{-1}(F^{'})$ by arc bridge rule. Now $N^{'}$ is a copy of N. □

Now let us consider which rules allow us to generate the subclasses of workflow nets. Based upon Theorem 1, it is easy to prove that we only need the transition bridge rule and the arc refinement rule to generate all state machine workflow nets, and we need the place bridge rule and the arc refinement rule to generate all marked graph workflow nets. They yield Corollary 1 and Corollary 2. As defined in Definition 10, Jackson nets are generated by the rules $R1$, $R2$, $R4$, $R5$, and $R6$. For a free-choice workflow net, the rules R1, R2, R4, and R5 can always preserve its properties.

Corollary 1. *If N is a state machine workflow net, then N can be copied into $N^{'}$ by only using the transition bridge rule with the arc refinement rule.*

Corollary 2. *If N is a marked graph workflow net, then N can be copied into $N^{'}$ by only using the place bridge rule with the arc refinement rule.*

Theorem 2. *The place refinement rule, the transition refinement rule, the place duplication rule, and the transition duplication rule preserve the free-choice workflow net property.*

[12] proves that each Jackson net is a sound net. From [6] we can derive that each generated state machine workflow net is sound as well. All free-choice workflow nets are sound. The generated marked graph workflow nets cannot be sound as the rules applied do not preserve the properties.

3.3 Generation of Nets

Our approach of generating Petri nets has two determinants, construction rules and probabilities of the rules. We have discussed the necessary rules to generate different classes of workflow nets. In this part, we focus on the probability-based rule selection.

Given two nets N and $N^{'}$, we say that N generates $N^{'}$ if and only if $N^{'}$ can be obtained from N by applying zero or more times a rule from our construction rules. To generate a workflow net we adopt a stepwise refinement technique starting with one single place. In the first step we apply the place refinement rule to generate the initial place and the final place. For any subsequent steps, we select a rule from all the rules whose conditions hold (we say those rules are *enabled*). For the initial and the final places there are some restrictions such that the place refinement rule, the place duplication rule, the loop addition rule cannot be applied to the initial and the final places, and for the transition bridge and the arc bridge rules, the initial place cannot be the target place (i.e., r in Definition 8 and 9), and the final place cannot be the source place (i.e., s in Definition 8 and 9). Figure 3 depicts an example of net refinement.

In order to select a rule from all the enabled rules, we attach a *rule weight* to each rule. A rule weight is a random number ranging from 0 (least likely) to 100 (most likely). Therefore, a rule is randomly selected based on the rule weight. Similarly, we also attach an *element weight* (also ranging from 0 to 100) to each of the elements (each of the places, transitions, and arcs) in a net. Hence each rule can randomly select an element or a pair of them based on the element weight to expand a net. Because we always start with an initial net to generate another net, for all the elements in the initial net, we give each of them a default element weight. When we apply rules, new elements are added into the net. In

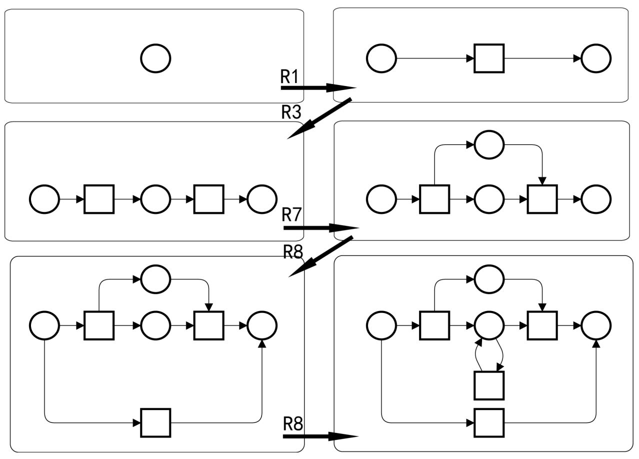


Fig. 3. An Example of Workflow Net Refinement

order to determine weights of the newly added elements, we define three *type parameters* (ranging from 0 to 1) for each element type, namely, *place parameter*, *transition parameter*, and *arc parameter*. Each rule can have some or all of the type parameters based on what kinds of elements it can introduce. For instance, as the place refinement rule can introduce two places, a transition, and two arcs, the rule has all the type parameters. While, the place bridge rule can add a place and two arcs, so it only has the place parameter and the arc parameter. All the refinement rules (R1,...,R6) only refine one element, so the weight of the newly added elements are determined by the multiplication of the weight of the refined element and the type parameters. For example, if we refine a place having a weight of 50, and we let the place parameter be 0.2, the transition parameter be 0.5, and the arc parameter be 0.1, then any newly added place has a weight of (0.2 x 50 =) 10, any newly added transition has a weight of (0.5 x 50 =) 25, and any newly added arc has a weight of (0.1 x 50 =) 5. On the other hand, all the bridge rules bridge a pair of nodes, so the weight of the newly added elements are determined by the multiplication of the sum of the bridged nodes and the type parameters. For instance, if we bridge two transitions having a weight of 50 and 70, respectively, then any newly added place has a weight of ((50 + 70) x 0.2 =) 24, and any newly added arc has a weight of ((50 + 70) x 0.1) = 12.

Therefore, such a mechanism makes net generation random. We are currently considering other mechanisms besides this one as well.

4 Characteristics of Net

Characteristics of Petri nets in different classes are decidable with the generated benchmarks. Such characteristics we consider currently consist of *length of the shortest path* (number of transitions in a path is the length of this path), *number* *of nodes*, *average number of fanin and fanout of nodes*, *deadlock and livelock*, and *coverage*. We use a real number to express each characteristic, in some cases we use 1 for *true* and 0 for *false*. Each Petri net in a class has a certain probability of being selected in a benchmark, and each characteristic can be considered as a random variable having a probability distribution over that class. For instance, with the given samples from a class, we can speak of the expectation and variance of the number of nodes in the class, or the probability of having a deadlock. Therefore, we can estimate the characteristics of any new nets from the same class with ceratin confidence. In this section, we focus on the characteristic *coverage* and give two examples to illustrate how it can help in testing.

We restrict ourselves to a Petri net with an initial node such as a workflow net. Each transition in the net is attached with a label. The goal of the coverage test is to find all the transitions with a certain label. The coverage is measured by *trace coverage* and *transition coverage*. We start a path with the initial node, at each choice point we select one of the enabled transitions at random. As soon as we find a transition with the label that we look for, we stop and start a new trace from the initial node again. Therefore, the trace coverage is the number of traces that we need to find all the transitions with the label, and the transition coverage is the number of transitions fired to find all the transitions with the label. We show two applications of how to use the coverage in testing.

Finding labels. We may attach labels to transitions in a workflow net. In this application we need to find all the transitions with a special label. We start in the initial state. A transition is randomly fired when there are two or more enabled transitions. After a transition has fired, we test whether it has the special label. If so we remove this label and we start a new trace from the initial state again. Otherwise we continue until we reach the final state and then we start from the initial state again. We stop if all the transitions with the label have been found. The coverage returns the number of traces followed and the number of transitions fired in order to find all the labeled transitions. Algorithm 1 describes this application. In this algorithm all transitions have a label either 0 or 1. We are looking for the transitions with label 1.

As labeling can be associated with different concepts, this algorithm can be used in model-based software testing as done in [7]. In this application we use a workflow net to model a software system, where each transition represents a software component. A software component either behaves correctly or has an error that can only be detected by firing the transition. Only transitions with an error are labeled.

Finding causal pairs. In this example we test how long (measured in coverage) it takes to cover all the *causal pairs*, i.e., a possible pair of consecutive transitions, in a workflow net. For process miners, i.e., the alpha algorithm [2], we need *complete logs*, i.e., log with a set of complete traces. This means that we need so many complete traces that every causal pair has occurred. We can get all the possible causal pairs for a given net by static analysis, i.e., by using the technique of reachability graph. Equipped with this result, we use Algorithm 2 to get the coverage for each given sound workflow net. This means that we have

Algorithm 1: Finding Labels

```
   input  : a sound $N = (P, T, F, i, f)$
   output: $pathCoverage, transitionCoverage$
1  var $A, B : A \subseteq T, B \subseteq T$
   var $pathCoverage, transitionCoverage : int$
   var $stop : boolean$
   var $m : P \longrightarrow \mathbb{N}$
   begin
2      $A := \emptyset; B := \emptyset; pathCoverage := 0; transitionCoverage := 0; stop :=$
       $false; m_0 := [i];$
       while $|A| < |T|$ do
3          $pathCoverage := pathCoverage + 1; m^{'} := m_0;$
4          repeat
5              $B := \{t \in T | (N : m^{'} \xrightarrow{t})\};$
6              $(N : m^{'} \xrightarrow{t} m^{''})$, for some $t \in B$;
7              $m^{'} := m^{''}$; $transitionCoverage := transitionCoverage + 1;$
               $A := A \cup \{t\};$
8              if $transitionLabel(t) \neq 0$ then
9                  $transitionLabel(t) := 0$; $stop := true$;
10             endif
11         until $m^{'} = [f]$ $or\ stop = true$
12     end
13 end
```

an estimate of the length of a log in order to be complete for random classes of Petri nets.

Finally, we show the result of an empirical study we did. We used the tool we developed (see Section 5) to randomly generated 3000 well-structured workflow nets. Those nets were used as benchmarks to get the results in Table 1.

5 Tool Implementation

We realized a supporting tool for our methodology. The tool is developed as a plugin for the ProM framework [10] in Java, the distribution can be downloaded from [16]. The tool has the following relevant features:

Graphical user interface. Figure 4 is a snapshot of the tool interface.This graphical user interface is easy to use. For example, it lists all the rules visually so that it is very intuitive for the user to select a particular rule. The generated nets can be visualized using the viewer provided by ProM. Testing results are output in a table for the user.

Selection of net class. The interface contains all the classes of the workflow nets that the user can generate. Once a particular class has been selected, all the rules which are not allowed in such a class are disabled automatically by giving their probabilities a value of zero.

Algorithm 2: Finding Causal Pairs

```
input  : a sound $N = (P, T, F, i, f)$, a set $C$ of all the causal pairs in $N$
output: $pathCoverage, transitionCoverage$
1  var $A, B : A \subseteq C, B \subseteq T$
   var $pathCoverage, transitionCoverage : int$
   var $u : u \in T$
   var $m : P \longrightarrow \mathbb{N}$
   begin
2      $A := \emptyset; B := \emptyset; pathCoverage := 0; transitionCoverage := 0; u := null; m_0 := [i];$
       while $A \neq C$ do
3          $pathCoverage := pathCoverage + 1; m' := m_0;$
4          $B := \{t \in T | (N : m' \xrightarrow{t})\};$
5          $(N : m' \xrightarrow{t} m'')$, for some $t \in B$;
6          $u := t; m' := m''; transitionCoverage := transitionCoverage + 1;$
7          repeat
8              $B := \{t \in T | (N : m' \xrightarrow{t})\};$
9              $(N : m' \xrightarrow{t} m'')$, for some $t \in B$;
10             $transitionCoverage := transitionCoverage + 1;$
11             $A := A \cup \{(u, t)\}; u := t; m' := m'';$
12         until $m' = [f]$
13     end
14 end
```

Random weight of rules. The user can change the weight of any enabled rule using the sliding bar under the rule image, the probability of the rule is calculated and displayed next to the sliding bar.

Table 1. Mean number, standard deviation, 25th percentile (Q1), Median (Q2), 75th percentile (Q3) of the characteristics of 3000 well-structured workflow nets.

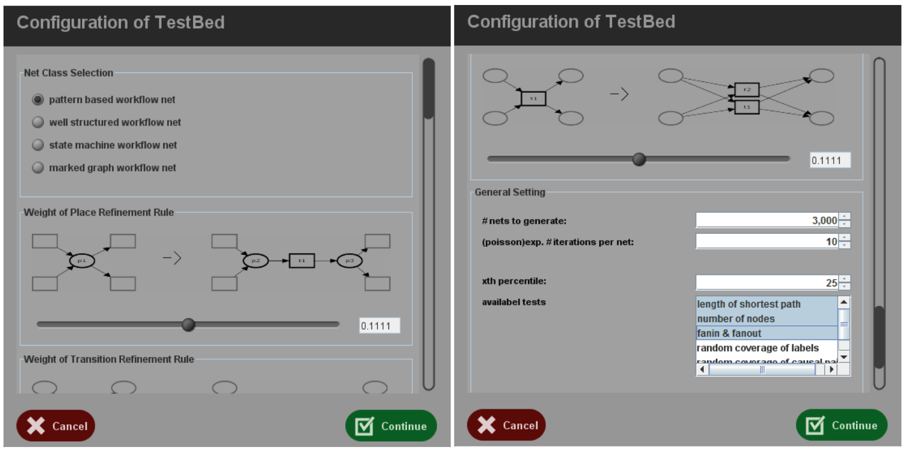
Characteristics		Avg.	Std.dev.	Q1	Q2	Q3
length of the shortest path		2.525	1.882	1.000	2.000	4.000
number of nodes	place	7.186	2.408	6.000	7.000	9.000
	transition	8.649	2.797	7.000	8.000	10.000
avg. fanin/out of nodes	place fanin	1.575	0.491	1.250	1.500	1.800
	place fanout	1.578	0.494	1.250	1.500	1.800
	transition fanin	1.283	0.332	1.000	1.200	1.400
	transition fanout	1.282	0.335	1.000	1.200	1.400
soundness		1.000	0.000	1.000	1.000	1.000
coverage	path coverage	15.032	10.125	8.400	12.100	18.500
	transition coverage	45.655	31.909	24.000	38.350	57.975

Specification of sample size and net size. The user can specify the number of nets to generate and the number of times to apply rules to generate a net. The sizes of all nets have a poisson distribution, the user only needs to input the average number, then the tool is able to calculate the size of each individual net.

Selection of characteristics. The user can select the characteristics introduced in Section 4 to investigate the nets.

Reusability of samples. The tool saves all the generated nets in PNML format in a dedicated folder on disk. This enables the user to reuse or share any benchmarks.

[13] proposed a systematic approach to design software in coloured Petri nets and transfer the CPN models into Java code, we practised the approach in the design and implementation of this tool. The tool currently runs on a single processor thus it is not possible to generate a very large size of benchmarks. In order to overcome this limit, we are considering to distribute the tool over multiple processors (e.g., over grid) in the future.


Configuration of TestBed

Net Class Selection

- [✓ pattern based workflow net]
- well structured workflow net
- state machine workflow net
- marked graph workflow net

Weight of Place Refinement Rule: 0.1111

Weight of Transition Refinement Rule

Configuration of TestBed

0.1111

General Setting

# nets to generate:	3,000
(poisson)exp. # iterations per net:	10
xth percentile:	25
availabel tests	length of shortest path number of nodes fanin & fanout random coverage of labels

Fig. 4. Interface Snapshot

6 Related Work

In [4] the authors proposed a method to generate Petri net benchmarks. They start with a Petri net containing two places and two transitions with all nodes connecting to each other, and make use of the refinement rules given by Murata in [18]. Whereas in our approach we use not only the Murata rules, but other rules as well, this makes our rules more extensive and more general. We focus on generating workflow nets, and start with a much simpler net with only one single place. They use a weighted random selection of rules to control the number of transitions and places. Such a probability mechanism is also realized in our approach, and we investigate more characteristics of nets besides the number of nodes.

In [3] the authors developed a tool to generate process models in Petri nets and log of business processes. They use a set of workflow patterns in [17] and add probabilities to the patterns. In our approach we do this for graph grammars. Both can get a probability distribution on the set of graphs.

7 Conclusion

In this paper, we defined a graph grammar with weights on the construction rules, such that each graph of the graph language has a ceratin probability of occurring. We consider graphs in the form of Petri nets, and the generated Petri nets can be used as benchmarks. By extensive studies, we have a set of construction rules to generate all Petri nets with different starting nets using a stepwise refinement approach. Based upon structures, the nets can be classified into different classes. Each Petri net in a class has a certain probability to be selected in a benchmark. Moreover, we distribute weights to the rules and all elements in a net, this makes the generation random. We have determined a number of characteristics used in the random classes of Petri nets, and we are able to get probability distributions of each characteristic over the classes. We focused upon an interesting characteristic called coverage, and presented two possible applications of it. A tool has been developed in ProM to realize our methodology.

In future research, we would like to identify the probability parameters based upon a concrete set of process models in a statistical way. If we have the parameter we can do better benchmarking. Also we would like to distribute our tool over grid so that we can generate a large number of benchmarks.

References

1. W. M. P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 2001.
2. W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16:2004, 2003.
3. Process Mining Group at University of Padua. Process Log Generator. http://www.processmining.it/sw/plg, 2010.
4. G. Bergmann, A. Horvath, I. Rath, and D. Varro. A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation. In *Proceedings of the 4th International Conference on Graph Transformation, ICGT'08*, pages 396–410, Leicester, UK, 2008.
5. G. Berthelot. Transformations and Decompositions of Nets. In *Lecture Notes in Computer Science: Advances in Petri Nets 1986 Part I: Petri Nets, central models and their properties . / W. Brauer, W. Reisig, and G. Rozenberg (Eds.)*, volume 254, pages 360–376. Springer Verlag, 1987.

6. I. Corro Ramos, A. Di Bucchianico, Hakobyan L., and K.M. van Hee. Synthesis and Reduction of State Machine Workflow Nets. Technical report CS 06-18, Edinhoven University of Technology, 2006.
7. I. Corro Ramos, A. Di Bucchianico, Hakobyan L., and K.M. van Hee. Model Driven Testing Based On Test History. In *Transactions on Petri Nets and Other Models of Concurrency I*, volume 1, pages 134–151, 2008.
8. J. Desel. Reduction and Design of Well-behaved Concurrent Systems. In *Proceedings on Theories of concurrency : unification and extension, CONCUR '90*, pages 166–181, New York, NY, USA, 1990. Springer-Verlag New York, Inc.
9. E. W. Dijkstra. Go To Statement Considered Harmful. *Communications of the ACM*, 11(3):147–148, March 1968.
10. B. F. van Dongen, A. K. A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, and W. M. P. van der Aalst. The ProM Framework: A New Era in Process Mining Tool Support. In *Lecture Notes in Computer Science: Applications and Theory of Petri Nets 2005: 26th International Conference, ICATPN 2005, Miami, USA, June 20-25, 2005. / Gianfranco Ciardo, Philippe Darondeau (Eds.)*, volume 3536, pages 444–454. Springer Verlag, jun 2005.
11. C. Girault and R. Valk, editors. *Petri Nets for System Engineering: A Guide to Modelling, Verification, and Applications*. Springer-Verlag, 2002.
12. K. M. van Hee, J. Hidders, G. Houben, J. Paredaens, and P. Thiran. On the Relationship between Workflow Models and Document Types. *Information Systems*, 34(1):178–208, 2008.
13. K. M. van Hee and Zheng Liu. From Service-Oriented Architecture via Coloured Petri Nets to Java Code. In *Proceedings of the Fifth International Workshop on Modellig of Objects, Compomens and Agents, MOCA'09*, pages 129–149, Hamburg, Germany, 2009.
14. K. M. van Hee, N. Sidorova, and M. Voorhoeve. Generalised Soundness of Workflow Nets Is Decidable. In *Proceedings of the 25th International Conference, ICATPN'04*, Bologna, Italy, 2004.
15. A. Lim, W. C. Oon, and W. B. Zhu. Towards Definitive Benchmarking of Algorithm Performance. In *Proceedings of the 11th European Conference on Information Systems, ECIS 2003*, Naples, Italy, 2003.
16. Architecture of Information System Group at indhoven University of Technology. Prom Nightly Builds. http://prom.win.tue.nl/tools/prom/nightly/, 2010.
17. N. Russell, A. H. M. ter Hofstede, W. M. P. van der Aalst, and M. Mulyar. Workflow Controlflow Patterns: A Revised View. Technical report, 2006.
18. I. Suzuki and T. Murata. A Method for Stepwise Refinement and Abstraction of Petri Nets. *Journal of Computer and System Sciences*, 27(1):51–76, 1983.