Testing *k*-Safe Petri Nets

Gregor v. Bochmann, Guy-Vincent Jourdan

School of Information Technology and Engineering (SITE)
University of Ottawa
800 King Edward Avenue, Ottawa, Ontario, Canada, K1N 6N5
{bochmann,gvj}@site.uottawa.ca

Abstract. Petri nets have been widely studied as tool for specification, modeling and analysis of concurrent systems. However, surprisingly little research has been done for testing systems that are specified with Petri nets. When a formal model is used, variations of Finite State Machines are often used for the automated generation of test cases. In this paper, we study automated conformance testing when the formal specification is given as a *k*-safe Petri net. We provide a general framework to perform these tests, and give a few algorithms for test case generation based on different assumptions. We provide two inefficient, but general algorithms for *k*-safe Petri net conformance testing. We also provide efficient algorithms for testing *k*-safe free-choice Petri nets under specific fault assumptions.

Keywords: Conformance testing, fault model, *k*-safe Petri nets, free-choice Petri nets, automatic test generation

1 Introduction

Software systems are notoriously incorrect, a problem that only gets worst when dealing with distributed systems. In order to help producing better systems, one common suggestion is to create a formal model of the specification of the system, and then test whether a given implementation *conforms* to the specification, for some suitable definition of conformance. Ideally the tests are automatically generated based on the formal specification. This kind of approach has been mostly researched using Finite State Machines (FSM) as the formal model (see e.g. [1] for a survey). More recently, the same questions have been asked for concurrent systems, for which FSM do not offer a good model. When modeling concurrent systems with FSMs for test generations, *multi-ports* FSM [2,3] and *Partial Order Input/Output Automata* [4] have been used. When modelling concurrent systems in general, a popular formalism among researchers are Petri Nets (see e.g. [5] for a survey). Surprisingly, little has been done in the area of conformance testing of systems specified as Petri Nets, even through Petri Nets have long been used in practice and have for example influenced the design of some UML schema and have been used as a basis for workflow modeling [6]. Other formalisms beside the already mentioned FSMs have been used in the context of testing distributed systems (for example for Labeled Transition

Systems [7] or Message Sequence Charts [8]), but in this context Petri Nets have mostly been used for fault diagnosis (see e.g. [9]). An interesting classification of testing criteria for Petri nets was provided by Zhu and He [10], but without testing algorithms.

In this paper, we investigate the question of automatically testing Petri Nets, to ensure that an implementation of a specification provided as a Petri Net is correct. We focus on *k-safe* Petri Nets, that is, Petri Net for which a place never holds more that *k* tokens at once, and on free-choice Petri Nets. This modeling paradigm is well suited for certain real-life applications, such as workflows [6] or control flows in networks of processors [11]. When used to model workflows, the traditional *transitions* of Petri Nets are seen as *tasks* of a workflow.

In this context, we provide a precise fault model, capturing what kind of changes could make a candidate implementation not conforming to the specification: some of the specified flow constraints between the tasks may be missing, or some unspecified flow constraints between tasks may be added. We provide a precise framework for the conformance question in Section 2.3. We then provide general testing algorithms for *k*-safe Petri Nets in Section 3. We first show that testing *k-safe* Petri Nets can be reformulated as a special case of state machine testing. This approach makes sense since state machine testing has been extensively studied. Unfortunately, the complexity of the transformation of a Petri Net into an state machines is exponential in the worst case. We then provide a more efficient testing algorithm, which is still exponential in the size of the Petri-Net (in the worst case). In Section 4, we study some particular cases of faults for *k*-safe free-choice Petri Nets, for which a test suite of polynomial length can be proposed. In Section 4.1, we provide an algorithm that generates a test suite that guarantees the detection of any number of missing flow constraints between the tasks in the implementation. Similarly, we discuss in Section 4.3 the testing of implementations with only additional *input flows* between tasks. We show in Section 4.3 that the problem is more difficult for additional *output* flows. We conclude in Section 5. The basic concepts and definitions are introduced in Section 2.

2 Basic Concepts and Assumptions

In this section, we give the definition of Petri Nets and we explain the assumptions that we make about testing environment.

2.1 Petri nets

Definition 1: Petri Nets, input/outputs, traces, executable sets, markings. A *Petri Net* is a 4-tuple $N=(P,T,F,M_0)$ where P is a finite set of places, T is a finite set of transitions (or *tasks* in our context), $F \subseteq (P\times T)\cup(T\times P)$ is a set of arcs (the *flows* in our context). A marking is a mapping from P to the natural numbers, indicating the number of tokens in each place. We write M_0 for the initial marking of the net.

For a task $t\in T$, we note $\bullet t=\{p\in P|(p,t)\in F\}$ the set of *inputs* of t, and $t\bullet=\{p\in P|(t,p)\in F\}$ the set of *outputs* of t. Similarly, for a place $p\in P$, we note

$\bullet p=\{t\in T|(t,p)\in F\}$ and $p\bullet=\{t\in T|(p,t)\in F\}$.
A task t is enabled for execution if all inputs of t contain at least one token. When a task is executed the marking changes as follows: The number of tokens in all inputs of t decreases by one, and the number of tokens in all outputs of t increases by one. A *trace* of a Petri Net is a sequence of tasks that can be executed starting from the initial marking in the order indicated, without executing any other tasks. An *executable set* of tasks is a multiset of tasks whose task elements can be sequenced into a trace. Clearly, for each trace there is a unique executable set, but several traces may correspond to the same executable set, in which case we say that the traces are *equivalent.* A *marking of a trace t* is the marking obtained after the execution of t from the initial marking. We say that t *marks* P if the place P contains at least one token in the marking of t.

Definition 2: Petri Net equivalence. Two Petri Nets $PN_1=(P_1,T_1,F_1,M_{10})$ and $PN_2=(P_2,T_2,F_2,M_{20})$ are said to be equivalent if $T_1 = T_2$ and they have the same set of traces.

Definition 3: Free-choice Petri Nets. A Petri Net $PN=(P,T,F,M_0)$ is *free-choice* if and only if for all places p in P, we have either $| p\bullet| < 2$ or $(\bullet(p\bullet)) = \{p\}$.

Definition 4: *k*-Safeness. A marking of a Petri Net is *k-safe* if the number of tokens in all places is at most *k*. A Petri Net is *k*-safe if the initial marking is *k*-safe and the marking of all traces is *k*-safe.

Proposition 1: *In a k-safe free-choice Petri Net $PN=(P,T,F,M_0)$, if for some task $t\in T$ and some place $p\in P$ such that $p\in\bullet t$, and $|\bullet t|>1$ there is no trace that marks $(\bullet t)\backslash p$ but not p then removing the input (p,t) from F defines a Petri Net PN' which is equivalent to PN.*

Proof: *Let $PN=(P,T,F,M_0)$ be a k-safe free-choice Petri Net, let $t\in T$ be a task of PN and $p\in P$ a place of PN such that p is in $\bullet t$, $|\bullet t|>1$ and there is no trace of PN that marks $(\bullet t)\backslash p$ but not p. Let PN' be the Petri Net obtained by removing (p,t) from F in PN. We show that PN' is equivalent to PN. Suppose they were not equivalent, that is, PN and PN' do not have the same set of traces. Since we have only removed a constraint from PN, clearly every trace of PN is also a trace of PN', therefore PN' must accept traces that are not accepted by PN. Let Tr be such a trace. Since the only difference between PN and PN' is fewer input flows on t in PN', necessarily t is in Tr. Two situations might occur: either Tr is not a trace of PN because an occurrence of t cannot fire in PN (i.e. $\bullet t$ is not marked at that point in PN), or another task t' can fire in PN' but not in PN. In the former case, because we have only removed (p,t), it mean that $(\bullet t)\backslash p$ is marked but $\bullet t$ is not, a contradiction with the hypothesis. In the latter case, if t' can fire in PN' but not in PN then $\bullet t'$ must be marked in PN' and not in PN. Again, the only difference being t not consuming a token in p in PN', it follows that t' can use that token in PN' but not in PN. In other words, $p\in\bullet t'$, but then $|p\bullet|>1$ and thus $\bullet(p\bullet) = \{p\}$ (PN is free-choice), a contradiction with $|\bullet t|>1$.*

2.2 Assumptions about the specification, implementation and testing environment

We assume that the specification of the system under test is provided in the form of a k-safe Petri Net. Moreover, we assume that there is a well identified initial marking. The goal of our test is to establish the conformance of the implementation to the specification, in the sense of trace equivalence (see Definition 2).

For the system under test, we make the assumption that it can be modeled as a k-safe Petri Net. In addition, we make the following assumptions: the number of reachable markings in the implementation is not larger than in the specification. This assumption corresponds to a similar restriction commonly made for conformance testing in respect to specifications in the form of state machines, where one assumes that the number of states of the implementation is not larger than the number of states of the specification. Without such an assumption, one would not be sure that the implementation is completely tested, since some of the (implementation) states might not have been visited. In the context of state machine testing, methods for weakening this assumptions have been described where it is allowed that the number of states of the implementation may exceed the number of states of the specification by a fixed number; similar considerations may also be considered for Petri nets, however, we do not address this question in this paper.

Regarding the testing environment, we assume that the system under test provides an interface which provides the following functions:

1. At any time, the tester can determine which tasks are enabled[1].
2. The tester can trigger an enabled task at will. Moreover, tasks will only be executed when triggered through the interface.
3. There is a reliable reset, which bring the system under test into what corresponds to the initial marking.

A good example of applications compatible with these assumptions are Workflow Engines. Workflow processes are usually specified by some specification language closely related to Petri Nets. Workflow engines are used to specify (and then enforce) the flow of tasks that are possible or required to perform for a given activity. In that setting, the set of possible tasks is well known, and a given task can be initiated if and only if the set of tasks that directly preceed it are finished. In a computer system, it typically means that at any time, the set of tasks that can be performed are enabled (accessible from the user interface) while the tasks that cannot be performed are not visible. It is thus possible to know what tasks are enabled, and choose one of them to be performed.

[1] We assume that the tester has access to the list of tasks that are enabled from the current marking, an assumption which is compatible with our example of workflow engines. Another option is to obtain such a list by attempting to fire all the tasks from the current marking, resetting the system and driving it to the current marking after each successful firing, This latter option is less efficient, but is still polynomial in the size of the net.

2.3 Fault Model

We assume that the difference between the system under test and the reference specification can be explained by certain types of modifications, called faults, as explained below. We do not make the single-fault assumption, thus the difference between the implementation and the specification may be due to several occurrences of these types of faults.

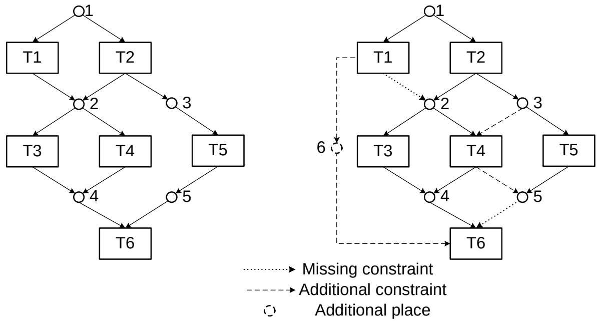

Figure 1: An example of a Petri Net specification (left) and a Petri Net implementation (right) with several types of faults

1. **Missing output flow**: a task does not produce the expected output, that is, does not put the expected token in a given place. In Figure 1, the specification (left) says that task T1 must produce an output into place 2. However, in the implementation (right), T1 fails do produce this output; this is a missing output flow fault.
2. **Missing input flow**: a task does not require the availability of a token in a given place to fire. In Figure 1, the specification (left) says that task T6 must take an input from place 5. However, in the implementation (right), T6 does not require this input; this is a missing input flow fault.
3. **Additional output flow**: a task produces an output into an existing place, that is, places a token into that place while the specification does not require such output. In Figure 1, the specification (left) says that task T4 does not produce an output into place 5. However, in the implementation (right), T4 produces this output; this is an additional output flow fault.
4. **Additional input flow**: a task requires the availability of a token in a given existing place while the specification does not require such a token. In Figure 1, the specification (left) says that task T4 does not require an input from place 3. However, in the implementation (right), T4 does require this input; this is an additional input fault.
5. **Additional place**: the implementation may contain an additional place which is connected with certain tasks through additional input and output flows. Note: The missing of a place in the implementation can be modeled by the missing of all its input and output flows.

We note that in principle, an implementation may also have missing or additional tasks, as compared with the specification. However, given the first test environment assumption described in Section 2.2, such a fault would be detected easily at the testing interface which displays the enabled tasks in any given state. The situation would be more complex if the tasks of the specification had labels and the observed traces would be sequences of the labels of the executed tasks; however, this situation is outside the scope of this paper.

In the rest of the paper, we are interested only in faults that actually have an impact on the observable behavior of the system, that is, create a non-equivalent Petri Net (in the sense of Definition 2). It may prevent a task to be executed when it should be executable, and/or make a task executable when it should not. It is clear that it is possible to have faults as defined here that create an equivalent system, either because they simply add onto existing constraints, or replace a situation by an equivalent one.

When considering faults of additional output flows, new tokens may “appear” anywhere in the system every time a task is executed. This raises the question of *k*-safeness of the faulty implementation. One may assume that the faulty implementation is no longer *k*-safe, and thus a place can now hold more than k tokens. Or one may assume that the implementation is still *k*-safe despite possible faults. Another approach is to assume that the implementation’s places cannot hold more than k tokens and thus additional tokens can “overwrite” others. Finally, and this is our working assumption, one may assume that violation of *k*-safeness in the system under test will raise an exception that we will catch, thus detecting the presence of a fault.

3 Testing *k*-safe Petri Nets: the general case

3.1 Using the testing techniques for finite state machines

In this section we consider the use of the testing techniques developed for state machines. We can transform any *k*-safe Petri Net into a corresponding state machine where each marking of the Petri Net corresponds to a state of the state machine, and each task of the Petri Net corresponds to a subset of the state transitions of the state machine. The state machine can be obtained by the classical marking graph construction method. Unfortunately, the number of markings may be exponentially larger than the size of the original Petri Net.

We note that the state machine obtained by the marking graph construction method is a deterministic labeled transition system (LTS) which is characterized by rendezvous interactions with its environment. In fact, the triggering of an LTS state transition corresponds to the execution of a task in the Petri net, and according to the first environmental testing assumption (see Section 2.2), this is done in rendezvous with the tester.

Therefore, we may apply here the testing methods that have been developed for LTS. Tan et al. [15] show how the various methods that have been developed for testing FSM with input/output behavior (see for instance [1,13]) can be adapted for testing of LTS, which interact through rendezvous. The paper deals with the general case of non-deterministic LTS and owes much to previous work on testing non-deterministic FSMs [14].

Algorithm 1: State machine testing

1. From the initial marking, enumerate every possible marking that can be reached. Call *E* this set of markings.
2. Create a finite state machine *A* having |*E*| states labeled by the elements of *E* and a transition between states if and only if in the Petri Net it is possible to go from the marking corresponding to the source state to the marking corresponding to the target state by firing one task. Label the state transition with this task. An example of this transformation is given in Figure 2.
3. Generate test cases for identifying the states in *A* using one of the existing methods for state machines.
4. For each *identified state* of the implementation,
   a. verify that no task is enabled that should not be enabled (by driving the implementation into that state and listing the tasks that are enabled);
   b. for each task not checked during state identification, verify that its execution leads to the correct next state.

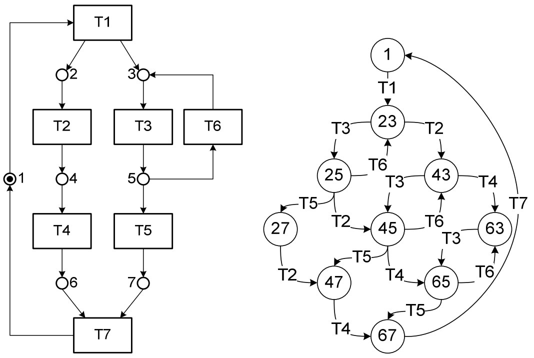

Figure 2: A simple Petri Net and the corresponding finite state machine

Algorithm 1 will produce an exhaustive verification of the implementation.

Proposition 2: *Under the assumption that the number of markings for the implementation is not larger than for the specification, Algorithm 1 detects every possible combination of faults (in the fault model) resulting in an implementation that is not equivalent to the specification.*

Proof: *If a combination of faults results in a non-equivalent implementation, this means that one of the following cases occurs:*

1. *A given marking of the specification cannot be reached in the implementation. This will be caught by Step 3 We note that the state identification methods for state machines normally assume that the state machine is minimized. We believe that the marking graph of most Petri nets considered here would be minimal. If this is not the case by assumption the number of implementation states does not exceed the number of states of the marking graph of the specification and we can*

use an extended state machine testing method that allows additional states compared with the minimized specification [13].

2. *From a given marking, a task that should be enabled is not, or it is enabled but executing it leads to the wrong marking. This will be caught by Step 3 and Step 4(b).*
3. *From a given marking, a task is enabled but shouldn't. This is typically not tested by checking sequences algorithms, but this problem will be detected by Step 5(a) of Algorithm 1***.** *Indeed, in our settings, we are able to list all tasks enabled in a given marking. The checking sequence algorithm will give us a means to know how to put the implementation into a state corresponding to a given marking of the specification, thus Step 4 will detect any additional transition.*

3.2 Using techniques specific to Petri Nets

In the general case, even if we cannot propose a polynomial algorithm, we can still look for a more efficient alternative than Algorithm 1. Indeed, every possible fault is in some senses localized on the net, in that the fault impacts a particular task, not a set of tasks. We still have to deal with the fact that in order to exhibit this fault, we may have to execute any set of tasks, but if we do so, this will indicate one faulty task. The idea is that, instead of testing every possible trace of the system (which is what Algorithm 1 indirectly does), it is enough to try every possible executable set. This is described in Algorithm 2.

Algorithm 2: Optimized generic testing

```
1. For all executable sets $S$
2.       reset the system and execute $S$ ;
3.       verify the status of every tasks of the net .
```

To illustrate the difference between Algorithm 1 and Algorithm 2, consider the (partial) Petri Net shown Figure 3. The following table shows the traces that will be tested by Algorithm 1, and an example of traces that could be selected by Algorithm 2, regarding this portion of the Petri Net.

To following proposition shows that Algorithm 2 will find every combination of faults:

Proposition 3: *Algorithm 2 detects every possible combination of faults (in the fault model) that results in an implementation that is not equivalent to the specification.*

Proof: *This results from the fact that in the type of Petri Nets we are considering, executing two traces that have the same executable set will necessarily lead to the same marking. In other words, if we execute the same multiset of tasks in any order (that can be executed), starting from the same marking, we always end up with the same marking.*

By definition, an implementation is faulty if and only if there is a trace tr which, when executed, leads to a task T that is in the wrong state (enabled when it shouldn't, or not enabled when it should). If there is such a fault, then there is such a trace, and *the algorithm will eventually test an executable set with the same multiset of tasks as that trace. Thus, because the multiset is the same, the same marking will be reached and the same state will be obtained for T. Since the same marking is expected, the same state is expected for T as well, and thus if the state is incorrect for tr, it will also be incorrect for the tested executable set, and the error will be detected.*

Algorithm 1	**Algorithm 2**
T1	*T1*
T1-T2	*T1-T2*
T1-T3	*T1-T3*
T1-T4	*T1-T4*
T1-T2-T3	*T1-T2-T3*
T1-T3-T2	
T1-T2-T4	*T1-T2-T4*
T1-T4-T2	
T1-T3-T4	*T1-T3-T4*
T1-T4-T3	
T1-T2-T3-T4	*T1-T2-T3-T4*
T1-T3-T2-T4	
T1-T2-T4-T3	
T1-T4-T2-T3	
T1-T3-T4-T2	
T1-T4-T3-T2	

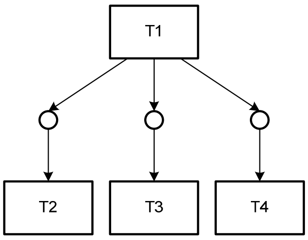

Figure 3: A simple net (partial) that leads to fewer executions when using Algorithm 2, compared to using Algorithm 1.

If violation to *k*-safeness of the implementation is not automatically detected, and additional tokens put into places simply overwrite the existing one, Algorithm 2 does not work, as shown in Figure 4 (with *k*=1), where the additional output from *T2* to *p* does violate the 1-safeness of the implementation, but the resulting error exhibits only if *T2* is executed after *T3*.

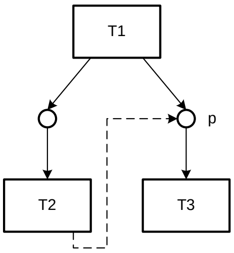

Figure 4: The additional output fault from T2 to p can be seen only if T2 is executed after T3

4 Testing *k*-Safe free-choice Petri nets under assumptions

In this section, we look at some particular cases for which we are able to provide better testing algorithms. We focus our attention on *k*-safe free-choice Petri nets, on which conflict and synchronization can both occur, but not at the same time. It is a well studied class of Petri nets because it is still quite large, yet more easy to analyze than general Petri nets (see e.g. [11]). In addition to considering free-choice nets, we will also make further assumptions on the possible combination of faults in the implementation.

4.1 Testing for missing flow faults

In some cases, it is possible to have more efficient testing algorithms. One such case is when the only possible faults are of type missing flow, that is, missing input flow or missing output flow. In this case, it is possible to check each input and output flow individually.

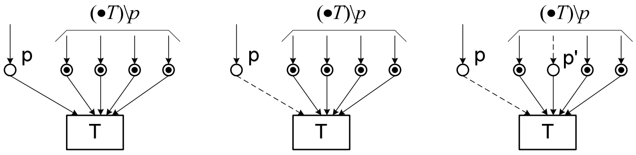

Figure 5: Missing input flow. When (•*T*)*p* are marked but not *p*, T is not enabled in the specification (left), but it is enabled in the implementation (center); however, some additional faults may interfere (right).

Intuitively, the principles are the following: for detecting a missing input flow, we note that a task *T* normally requires all of its inputs to be marked to be enabled. For example, in Figure 5, left, the task *T* is not enabled because even though places (•*T*)*p* are all marked, place *p* is not. However, if the input flow from *p* to *T* is missing in the implementation (center), then is the same situation, with places (•*T*)*p* marked but place *p* not marked, *T* becomes enabled. The testing algorithm will thus, for all tasks *T* and all places *p* in •*T*, mark all places in (•*T*)*p* and check if *T* is enabled. However, as shown in Figure 5 (right), in some case the missing input flow may not be detected;

this may happen when at least one place *p'* in (•*T*)\p was not successfully marked, because of some other (missing output) faults in the implementation.

The idea for testing for missing output flows is the following: if a task *T* outputs a token into a place *p*, and a task *T'* requires an input from *p* to be enabled, then marking all the places in (•*T'*) by executing *T* to mark *p* (among other tasks) will enable *T'* (Figure 6, left). If *T* is missing the output flow towards *p*, then *T'* will not be enabled after attempting to mark all the places in (•*T'*) because *p* will not actually be marked (Figure 6, center). Again, the situation can be complicated by another fault, such as a missing input flow between *p* and *T'*, in which case *T'* will be enabled despite the missing output (Figure 6, right).

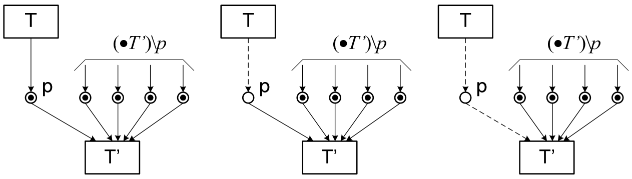

Figure 6: Missing output flow. Correct situation (left), missing output flow of T (center), and interference by a missing input flow to T'.

The problems of the interference between several faults, as indicated in the right of Figure 5 and Figure 6, will be addressed indirectly by the proof of Proposition 4 which will show that the verification for missing input might fail because of another missing output, and the verification for missing output might fail because of a missing input, but they cannot both fail at the same time.

Formally, the algorithm for testing for missing input flows is the following:

Algorithm 3: Testing for missing input flows

```
1. For all task $T$
2.     For all place $p$ in (•$T$)
3.         If there is a trace $S$ that marks (•$T$)\$p$ but not $p$
4.             Reset the system and execute $S$
5.             Verify NOT-ENABLED(T)
```

Line 3 is a reachability problem, which is PSPACE-complete for the kind of Petri Nets we are dealing with [12]. This means that the algorithm for determining a test suite for this purpose has a complexity that is PSPACE-complete. However, the length of the obtained test suite is clearly polynomial in respect to the size of the given Petri Net.

As we will see in the proof of Proposition 4, this algorithm is not sufficient on its own, since it can miss some missing input flows, when combined with missing output flows. It must be run in combination with Algorithm 4 which tests for missing output flows:

Algorithm 4: Testing for missing output flows

```
For all task $T$
    For all place $p$ in $(T\bullet)$
        For all task $T'$ in $(p\bullet)$
            If there is trace $S$ that contains $T$ and marks $(\bullet T')$ with a single token in $p$
                Reset the system and execute $S$
                Verify $ENABLED(T')$
```

As above, the complexity of the test selection algorithm is PSPACE-complete, however, the obtained test suite is polynomial in respect to the size of the given Petri Net. Proposition 4 shows that executing the tests generated by both algorithms is enough to detect all missing flow faults in the absence of other types of faults. To guarantee the detection of these faults, we must assume that the implementation only contains faults of these types.

Proposition 4: *Executing both Algorithm 3 and Algorithm 4 will detects any faulty implementation that has only missing input and/or output flow faults.*

Proof: *If an implementation is not faulty, then clearly neither algorithm will detect a fault. Assume that there is a task T with a missing input flow from a place p. If there is no trace S that marks* $(\bullet T)\backslash p$ *but not p, then Proposition 1 shows that the input flow is unnecessary and the resulting Petri Net is in fact equivalent to the original one. We therefore assume that there is such a trace S. If after executing S successfully* $(\bullet T)\backslash p$ *is indeed marked, then T will be enabled and the algorithm will detect the missing input constraint. If after executing S T is not enabled, that means that* $(\bullet T)\backslash p$ *is in fact not marked. The problem cannot be another missing input for T, since it would mean fewer constraints on S, not more, and wouldn't prevent T to be enabled. Thus, the only remaining option is that some task T' in* $\bullet((\bullet T)\backslash p)$ *did not mark the expected place in* $(\bullet T)\backslash p$ *when executing S, that is, T' has a missing output flow. In conclusion, Algorithm 3 detects missing input flows, except when the task missing the input constraint has another input flow which is not missing but for which the task that was to put the token has a missing output flow.*

Assume now that there is a task T with a missing output flow to a place p. If this output flow is not redundant, then there is a task T' that has p as an input flow and that will consume the token placed there by T. Such a T' will be found by Algorithm 4. Because of the missing output flow, normally T' will not be enabled after executing S and the algorithm will catch the missing flow. However, it is still possible for T' to be enabled, if it is missing the input flow from p, too. In conclusion, Algorithm 4 detects missing output flow, except when the missing output flow is to a place that has an input flow that is missing too.

To conclude this proof, we need to point out that each algorithm works, except in one situation; but the situation that defaults Algorithm 3 is different from the one that defaults Algorithm 4. In the case of Algorithm 4, we need a place that has lost both an input and an output flow, while in the case of Algorithm 3, we need a place that has lost an output flow but must have kept its input flow. Thus, by running both algorithms, we are guaranteed to detect all problems, Algorithm 4 catching the problems missed by Algorithm 3, and vice versa.

4.2 Testing for additional input flow faults

Testing for additional flow faults is more difficult than testing for missing flow faults because it may involve tasks that are independent of each other according to the specification. Moreover, the consequence of this type of faults can be intermittent, in the sense that an additional output flow fault may be cancelled by an additional input flow fault, leaving only a short window of opportunity (during the execution of the test trace) to detect the fault. In the case of additional input flow faults without other types of faults, we can still propose a polynomial algorithm, but we cannot check for additional output flow faults in polynomial time, even if no other types of faults are present.

An additional input flow fault can have two different effects: it may prevent a task from being executed when it should be executable according to the specification, because the task expects an input that was not specified, or it may prevent some other task from executing because the task with the additional input flow has unexpectedly consumed the token. Figure 7 illustrates the situation: task *T* has an additional input flow from place *p* (left). In the case where •*T* is marked, but *p* is not (center), *T* is not enabled, although it should. If *p* is marked too (right), then *T* can fire, but then *T*' cannot anymore, even though it should be enabled.

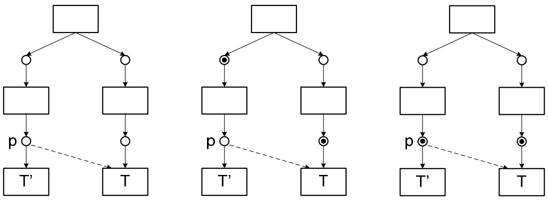

Figure 7: Additional input flow fault: T has an additional input flow from p (left). This may prevent T from firing (center), or, when T fires, T' becomes disabled (right).

A more general description is illustrated in Figure 8: in order to mark •*T*, some trace is executed (the dashed zone in the Figure). While producing this trace, some other places will also become marked (for example *p'*) while other places are unmarked (for example *p*). This normal situation is shown on the left. If *T* has an additional input constraint from *p* (center), then after executing the same trace, the faulty •*T* will not be enabled. If *T* has an additional input constraint from *p'* (right), then the faulty •*T* is still marked after generating the trace, so *T* is enabled, however, if it fires it will disable *T*'.

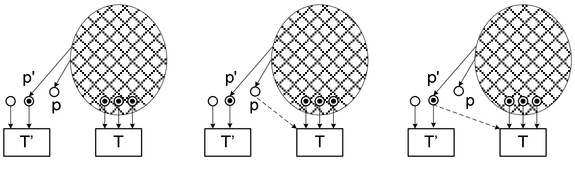

Figure 8: Additional input flow fault (see explanation above)

Consequently, in order to detect these faults, in the absence of any other type of faults, we use an algorithm that works in two phases: first, a trace is executed that should mark $\bullet T$ and verifies that T is indeed enabled. This shows that either T does not have an additional input constraint, or that the additional input place happens to be marked as well. So the second phase checks that the places that are marked by this trace (and that are not part of $\bullet T$) are not being unmarked by the firing of T. Algorithm 5 gives the details.

Algorithm 5: Testing for additional input flows

```
1.  For all task $T$
2.      Find a trace $S$ that marks $\bullet T$
3.      Reset the system and execute $S$
4.      Verify $ENABLED(T)$
5.      For all place $p$ not in $\bullet T$ which is marked by $S$
6.          If there is a trace $S'$ containing T and another task $T'$ in $p\bullet$ such that
                                $p$ should have a single token when $S'$ is fired
7.              Reset the system and verify that $S'$ can be executed
8.          Else if there is a trace $S''$ marking $\bullet T$ but not $p$
9.              Reset the system and execute $S''$
10.             Verify $ENABLED(T)$
```

Proposition 5: *Executing Algorithm 5 will detect any faulty implementation that has only additional input flow faults.*

Proof: *If a task T has an additional input from a place p and that fault it not caught at line 4, it necessarily means that p is marked by trace S, and expected to be so because we consider only additional input flow faults. Such a case will be dealt with by lines 5 through 10. If the fault has an impact (i.e. if it leads to a wrong behavior of the implementation), then there must be a task T' in p• and a trace containing both T and T' that is executable according to the specification but not in the implementation. Again, because we consider only additional input flow faults, any trace containing both T and T' and that ends up with no token in p will fail, since when the task consuming the last token in p is executed, the token in p has already been consumed as often as it has been set, that task will not be enabled. Lines 6 and 7 of the algorithm ensure that such a trace will be found and run, therefore a fault with an impact when p is marked will be caught. Finally, the fault might have no impact when p is marked, but if there is another way to enable T without marking p, via some trace S″ then T would not be enabled when S″ is executed. Line 9 and 10 of the algorithm address this case.*

4.3 Testing for additional output flow faults

The fault of an additional output flow to an existing place might enable a task when that task should not be enabled. Detecting this type of faults is more difficult than additional input flows, and we cannot do it in polynomial time even in the absence of other types of fault.

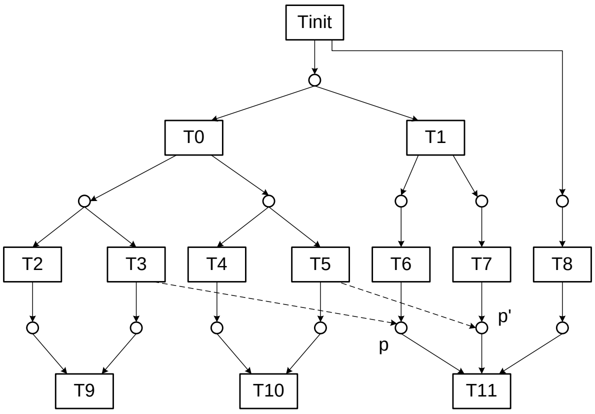

Figure 9: Additional output flow faults: the additional output flows from *T3* to *p* and from *T5* to *p'* can be detected only if these two tasks are included in the trace

When considering additional output flows, additional tokens may be placed anywhere in the system every time a task is executed. Thus, any trace may now mark any set of places anywhere in the net, so we cannot have any strategy beyond trying everything. This is illustrated in Figure 9, with two additional output flows, one from *T3* to *p* and one from *T5* to *p'*. Neither *T3* nor *T5* are prerequisite to *T11*, so to exhibit the problem requires executing tasks that are unrelated to the problem's location, namely *T3* and *T5*. Both *T3* and *T5* are branching from a choice, and of the 4 possible combinations of choices, there is only one combination that leads to the detection of the problem. For detecting an arbitrary set of additional output flow faults, it is therefore necessary to execute traces for all possible combination of choices.

Because of these difficulties, we cannot suggest a polynomial algorithm for these types of faults.

5. Conclusion

In this paper, we look at the question of conformance testing when the model is provided in the form of a *k*-safe Petri Net. We first provide a general framework for testing whether an implementation conforms to a specification which is given in the form of a *k*-safe Petri Nets. The types of errors that we consider in this paper include faults of missing or additional flows (inputs to, or outputs from tasks). We provide two general, but inefficient algorithms for testing these faults; they lead in general to test suites of exponential length. The first one is derived from methods originally developed for state machines, while the second one, slightly more efficient but still exponential, is specific to Petri Nets. We then identify special types of faults for which polynomial test suites can be provided when free-choice Petri Nets are considered

Acknowledgments: This work has been supported in part by grants from the Natural Sciences and Engineering Research Council of Canada.

References

[1] Lee D, Yannakakis M (1996) Principles and methods of testing finite state machines – a survey. *Proceedings of the IEEE*, 84(8):1089–1123.

[2] G. Luo, R. Dssouli, G. v. Bochmann, P. Ventakaram and A. Ghedamsi, Generating synchronizable test sequences based on finite state machines with distributed ports, *IFIP Sixth International Workshop on Protocol Test Systems*, Pau, France, September 1993, pp. 53-68.

[3] J. Chen, R. Hieron and H. Ural, Resolving observability problems in distributed test architecture, *FORTE 2005, LNCS 3731*, 2005, pp. 219-232.

[4] G. v. Bochmann, S. Haar, C. Jard and G.-V. Jourdan, Testing Systems Specified as Partial Order Input/Output Automata. *TestCom 2008, LNCS 5047*, pages 169-183, Springer 2008.

[5] T. Murata (1989) Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4): 541–580.

[6] W. van der Aalst, T. Weijters, L. Maruster, (2004) Workflow mining: discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9): 1128–1142.

[7] P. Bhateja, P. Gastin, and M. Mukund, A Fresh Look at Testing for Asynchronous Communication. *ATVA 2006, LNCS 4218*, pages 369-383, Springer 2006.

[8] P. Bhateja, P. Gastin, M. Mukund and K. Narayan Kumar, Local Testing of Message Sequence Charts Is Difficult. *Fundamentals of Computation Theory, 2007, LNCS 4639*, pages 76-87, Springer 2007.

[9] S. Haar (2008) Law and Partial Order. Nonsequential Behaviour and Probability in Asynchronous Systems. *Habilitation à diriger les recherches, INRIA. http://www.lsv.ens-cachan.fr/~haar/HDR.pdf.*

[10] H. Zhu and X. He, (2002) A methodology of testing high-level Petri nets. *Information and Software Technology*, 44(8): 473-489.

[11] J.Desel and J. Esparza (1995) Free choice Petri nets *Cambridge Tracts In Theoretical Computer Science* Vol. 40. ISBN:0-521-46519-2.

[12] A. Cheng, J. Esparza and J. Palsberg (1995). Complexity results for 1-safe nets. *Theoretical Computer Science* 147(1-2): 117-136.

[13] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou and A. Ghedamsi, Test selection based on finite state models, *IEEE Transactions on Software Engineering*, Vol.17, no.6, June 1991, pp. 591-603.

[14] G. Luo, A. Petrenko and G. v. Bochmann, Selecting test sequences for partially-specified nondeterministic finite state machines, *Proc. of the International Workshop on Protocol Test Systems (IWPTS'94)*, Tokyo, Japan, Nov. 1994, pp.95-110.

[15] Q. M. Tan, A. Petrenko and G. v. Bochmann, Checking experiments with labeled transition systems for trace equivalence, in *Proc. IFIP 10th Intern. Workshop on Testing of Communication Systems(IWTCS'97)*, Cheju Island, Korea, 1997.