# *Petrinets Testing*: A Petri Net Modeling and Testing Language Developed with MontiCore*

Daniel Petri, Roy Hermanns, and Haikun Huang

RWTH Aachen University, Aachen, Germany
{daniel.petri, roy.hermanns, haikun.huang}@rwth-aachen.de

**Abstract.** Ensuring reliability and security of software to improve the quality becomes an urgent issue. Petri nets are not only a graphical modeling tool but also a formal method with strict grammar and semantic definitions that can be used to model and describe the system effectively at the same time. Therefore model-based testing techniques with petri net can be used to develop tests of systems based on formal criteria to find errors at early stages in development. It is even possible to generate testcases automatically.
We define a textual modeling language, *Petrinets Testing*, based on the MontiCore Language Workbench [6]. Our language is an extension of the *petrinets4analysis* language, which is to analyze the given petri net model. The goal of the *Petrinets Testing* language is to verify the correctness of the given petri net model. Due to the powerful language generation by MontiCore based on a simple context-free grammar, only a small hand-written codebase is needed, enhancing maintainability and extensibility.
Our work is to create a modeling DSL (domain specific language) using the Monticore to automate the inspection process for petri nets. Moreover, The test cases used in the *Petrinets Testing* can be generated automatically or set manually. Finally a JUnit class is generated for each petri net that needs to be tested.

**Keywords:** Petri net · Model-based Testing · JUnit

## 1 Introduction

Petri nets are not only a graphical modeling tool but also a formal method with strict grammar and semantic definitions that can be used to model and describe the system effectively at the same time. Therefore model-based testing techniques with petri net can be used to develop tests of systems based on formal criteria to find errors at early stages in development.

Based on the MontiCore Language Workbench [1], we propose the textual modeling language *Petrinets Testing*. There are three components in our *Petrinets Testing* project, they are *Testcase Generator*, *Petrinets Testing language*, *JUnit*

* Practical Project for the lecture "Software Language Engineering" by Prof. Dr. B. Rumpe. The authors are grateful to Imke Drave for her supervision of the project.

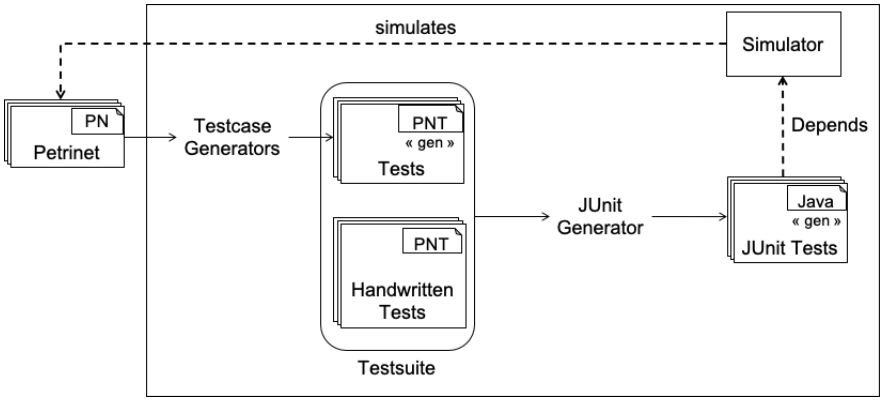


Fig. 1: The *Petrinets Testing* language Architecture

*Test Generator* and *Simulator*. The architecture of *Petrinets Testing* project is in Fig 1. The specific responsibilities of each part are as follows:

- *Testcase Generator*: Any petri net model can be used as an input for the *Petrinets Testing* project. First, we will use the *petrinets4analysis* language to parse the given petri net model. Through the parsed petri net model, the system can automatically generate test cases using the algorithm we set, which will be mainly introduced in chapter 4. Furthermore, we can also create test cases manually according to the grammar of *Petrinets Testing* language. Finally we will get a Petrinets Testing model.

- *Petrinets Testing language*: *Petrinets Testing* is a domain-specific language (DSL) that is easily usable also by domain experts without knowledge of programming. The grammar of the *Petrinets Testing* language is defined by the MontiCore Language Workbench, which produces Java classes for the abstract syntax tree (AST) as well as additional infrastructure by our defined keywords and syntactic sugar in the MontiCore Language Workbench. In *Petrinets Testing* language, we can define the initial markings, simulated transitions and expected markings of given petri net model. Each petri net model will have a corresponding *Petrinets Testing* model (an example is in Listing 2). Through the *Petrinets Testing* language, we can define multiple test cases in the same model.

- *JUnit Test Generator*: The petri net test needs to use JUnit, which is a simple framework to write repeatable tests, so the function of the JUnit Test Generator model is to convert the generated *Petrinets Testing* model into a JUnit class. In each of JUnit Class, we also add some auxiliary functions for the next simulation operation.

– *Simulator*: After the program has generated *Petrinets Testing* models for all petri nets, this module *Simulator* will automatically execute the corresponding JUnit class for petri net testing. In the process of running, the program will first need the parsed petri net model of *petrinets4analysis* language, so the *petrinets4analysis* language is also used in this process. When the results of each JUnit class running meet the expected conditions, the system will finally give a "pass" signal.

## 2 Background

We assume that the reader is familiar with the basic concept of a petri net. Nevertheless, we quickly introduce the fundamentals in this chapter. In addition, the *Petrinets Testing* language can automatically create the testcase of given petri net model by cause-effect-net[2], so we also introduce the concept of cause-effect-net in this chapter.

### 2.1 Petri Nets

Petri nets are a graphical and mathematical modeling tool which are useful for modeling systems with concurrent, asynchronous, distributed or parallel properties. They have been observed to have broad application in modeling finite state machines, parallel activities, dataflow computations, communication protocols, sychronization control, discrete event systems, and asynchronous circuits [4]. They were introduced by Carl Adam Petri in 1962 [5]. In early 1970's MIT was very active in the research of Petri nets. Since the late 1970's, European researchers have organized workshops and published conference proceedings on Petri nets [4].

The Petri nets can mathematically be described as follows:

Let $\mathbb{N}$ denote the set of nonnegative integers. A Petri net with inhibitor arcs is a 6-tuple $PN = (P, T, F, I, V, m_0)$, where:

– $P = \{p_1, p_2, ..., p_{|P|}\}$ is a set of places, where $|P|$ denotes the cardinality of set $P$,
– $T = \{t_1, t_2, ..., t_{|T|}\}$ is a set of transitions, where $|T|$ denotes the cardinality of set $T$ and $P \cap T = \emptyset$,
– $F \subseteq (P \times T) \cup (T \times P)$,
– $I \subseteq (P \times T)$,
– $V$ is a weight function: $F \to \mathbb{N}$, and
– $m_0$ is the the initial marking $P \to \mathbb{N}$.

Petri nets can be viewed as bipartite directed multigraphs. The set of nodes is divided into two disjoint sets: $P$ and $T$. An arc in $F$ connects a pair of nodes in $P \times T$ or $T \times P$. The mapping $V$ assigns non-negative integers to arcs. An arc assigned with number $k$ denotes $k$–parallel arcs between the same pair of nodes. A *marking* $m \in (\mathbb{N} \cup 0)^n$ is a mapping from every place $p$ to a non-negative

integer. If the number $k$ is assigned to the place $p$, we say that there are $k$ tokens on place $p$. The number of tokens on place $p$ under marking $m$ is represented by $m(p)$. The initial marking $m_0$ is the distribution of tokens between the places of the net before any action is taken. The input place of a transition $t$ , denoted as $t^-$ , is a set of places $\{p|p \in P, (p,t) \in F\}$ . The output place of a transition $t$ is a set of places defined as $\{p|p \in P, (t,p) \in F\}$ , and is denoted by $t^+$ . The functions $t^-(p) = i$ is defined as $(p,t) \in F$ and $V((p,t)) = i$. Similarly $t^+(p) = i$ is defined as $(t,p) \in F$ and $V(t,p) = j$.

A transition $t$ is said to be enabled given a marking $m$ iff $\forall p \in t^-, m(p) \geq t^-(p)$. An enabled transition is ready to *fire*. When a transition fires, it removes tokens from its input places and puts tokens in its output places. An enabled transition may or may not fire, and there may be more than one transition enabled in a given marking, but only one transition can fire at a time. If the current marking of the net is $m$, $t$ is the fired transition, and $m'$ is the marking after the firing of $t$, the relation between $m$ and $m'$ is $m'(p) = m(p) - t^-(p) + t^+(p)$, for every $p \in P$.

An inhibitor arc in $I$ is an arc connecting a place and a transition. The connected transition cannot fire if the input place along the inhibitor arc contains at least one token. A transition with inhibitor arcs is enabled iff $\forall p \in t^-, m(p) \geq t^-(p)$ and every input place along the inhibitor arcs contains no token. After the transition fires, no token is removed through the inhibitor arc. Inhibitor arcs give Petri nets the ability to test "zero". This extends the modeling power of Petri nets to the level of Turing machines.

### 2.2 Cause Effect Net

The cause-effect-net is evolved from cause-effect graphing, which is a method of a black box program testing. The cause-effect graphing [7] is used to describe the causal relationship between the input and output of the system, and the constraint relationship between the input and the input. The drawing process of the cause-effect graphing is the modeling process of the external characteristics of the tested system. According to the cause-effect graphing between the input and output of the system, the judgment table is also called the test case table, so as to plan the test case. The cause-effect-net also generates tables in the same way as cause-effect graphing. When we have the test case table, then we can generate the corresponding the *Petrinets Testing* model according to the *Petrinets Testing* language grammar.

There are four types of relationships in the Petri net that need to be considered, they are OR-, AND-, SINGLE- and ALTERNATE-relationship. According to these four types of relationships, we define the input places of transition as a cause or combination of causes, and the output places of transition as effects. The following are the definitions of the four relationships:

– OR-relationship:
  Alternative causes which belong to the same effect (see Fig 2)

– AND-relationship:
  Common causes which belong to one effect (see Fig 3)
– SINGLE-relationship:
  A cause which directly leads to one effect (see Fig 4)
– ALTERNATE-relationship:
  A single cause or a combination of causes may lead to different effects (see Fig 5)

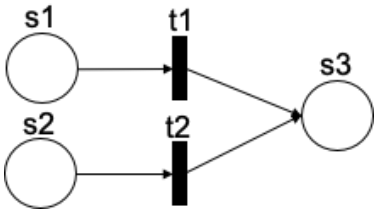


Fig. 2: OR-relationship

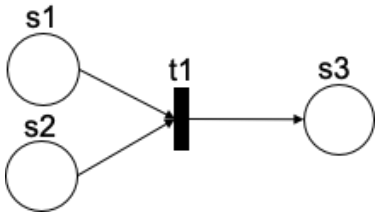


Fig. 3: AND-relationship

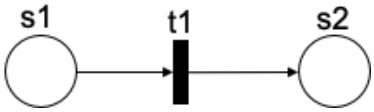


Fig. 4: SINGLE-relationship

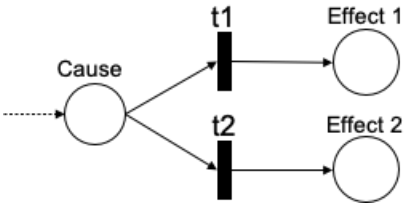


Fig. 5: ALTERNATE-relationship

The most important of the four types of relationship is the ALTERNATE-relationship, because this is a non-determinism relationship, which means the place has more than one successor transition. In our algorithm, we focus on finding the ALTERNATE-relationship in the all Petri net to determine the causes and effects in the cause-effect-net.

## 3 Language

The grammar is the foundation of the *Petrinets Testing* language which is defined on a syntactical level and on a semantic level, both aided by MontiCore generation capabilities.

### 3.1 Grammar Design

The syntactic foundation of the *Petrinets Testing* language is a context-free grammar which defines all valid keywords and permissible input values as well as syntactic sugar (e.g. punctuation and bracketing), and their static arrangement [6]. The grammar of the *Petrinets Testing* language (see Listing 1) is defined by the MontiCore Language Workbench, which produces Java classes for the abstract syntax tree (AST) as well as additional infrastructure by our defined keywords and syntactic sugar in the MontiCore Language Workbench. The desired modeling elements are

- Testcase features: orginal petri net, testcase body, testing step, testing expectation
- Testcase body elements: initial markings, inherit markings, define markings, markings with places
- Testing step elements: defined simulation
- Testing expectation elements: place binding with any or all conditional markings

The **PetriNetTest** grammar defines the structure of the *Petrinets Testing* language. Firstly, the grammar should know which petri net needs to be tested, so we define **Import**, which is to import the original petri net model that needs to be tested, which is parsed by the *petrinets4analysis* language[3]. The following part is about the all testcases of the original petri net model. The **PetriNetTest** grammar can contain one and more than one one-terminal **Testcase**. In each testcase body, we can define the **InitialMarking** of the petri net that needs to be tested. If we want to use the initial markings from the original petri net model, we can use the **InheritMarking** to inherit the original markings. Moreover, we can also make modifications to the inherited markings. If we want to define the new initial markings, we can use **DefineMarking** non-terminals to define the initial markings we need, which can be specified in their corresponding places.

In the testing step part, we can use **Simulation** non-terminals to specify which transitions are required for the *Petrinets Testing* language model to simulate, where the order of the transitions affects the final testing expectation.

In the testing expectation part, we can use **Expectation** to define the expected markings model will reach, which can be specified in their corresponding places. In order to realize the conditional expected markings, we define **BooleanCondition** interface utilizing MontiCore through an **astrule Condition** for the **Expectation** non-terminal. At this point we can define various combinations of expected markings with ”all” and ”any” keywords.

### 3.2 Well-formedness of Models

The grammar ensures syntactic correctness of the *Petrinets Testing* model definitions, but additionally, models must be well-formed. The context conditions (CoCos) in MontiCore will be verified the semantic correctness of the *Petrinets*

```
grammar PetrinetTests extends de.monticore.literals.MCCommonLiterals  {

    symbol scope PetriNetTest = "pntest" Name "{"
        Import
        (TestcaseBody | Testcase+)
    "}";

    symbol Testcase = "testcase" Name "{"
        TestcaseBody
    "}";

    Import = "use" "petrinet" Name;

    TestcaseBody = InitialMarking Expectation* TestStep*;
    TestStep = Simulation+ Expectation+;
    Simulation = "simulate" "{" (Name || ",")+ "}";

    InitialMarking = "initial" "marking" (InheritMarking | DefineMarking);
    InheritMarking = "inherited" RestSpecification?;
    DefineMarking = "{" (PlaceBinding || ",")* ("," RestSpecification)? "}";
    PlaceBinding = place:Name value:MarkingValue;
    RestSpecification = "rest" MarkingValue;
    MarkingValue = NatLiteral;

    Expectation = "expect" Condition;
    interface Condition;
    astrule Condition =
      method Optional<Boolean> verify(
        petrinet._ast.ASTPetrinet petrinet, petrinet.analysis.Marking marking) { };
    interface BooleanCondition extends Condition;
    Negation implements BooleanCondition = "not" Condition;
    Conjunction implements BooleanCondition = "all" "{" (Condition || ",")* "}";
    Disjunction implements BooleanCondition = "any" "{" (Condition || ",")* "}";

    MarkingCondition implements Condition =
      "marking" "{" (PlaceBinding || ",")* "}";
    EnabledCondition implements Condition =
      "enabled" "{" (Name || ",")* "}";
}
```

Listing 1: The *Petrinets Testing* MontiCore grammar

*Testing* model. A context condition (CoCo) is a predicate on a context-free grammar correct sentence where the context of a word is used to determine the total correctness, also called well-formedness[6].

The *Petrinets Testing* language uses the *petrinets4analysis* language to parse the petri net model, and the *Petrinets Testing* language will simulate the petri net with the defined initial markings and transitions, so a place that appears in the *Petrinets Testing* model must exist in the petri net model in the *petrinets4analysis* language. In addition, all transitions in the *Petrinets Testing* model must also exist in the petri net model in the *petrinets4analysis* language.

## 4 Automatic Test Case Generation

The *Petrinets Testing* language can automatically analyze the Petri net and automatically generate test cases of this petri net. We decided to use cause-

effect-net [2] to automatically create test cases. In this paper [2], the author introduced how to find Causes and Effects, and provided a method to create a casue-effect-net to generate a test case table. We adopt the method mentioned in the paper and implement it with a program to complete the automatic generation of the test case table about a given petri net.

### 4.1 Test case generation

We use the *petrinets4analysis* language to parse the given Petri net and obtain the entire Petri net, and then use the algorithm to transform the Petri net into the cause-effect-net and get the test case table.

- Step 1: Finding the causes and effects:
  First, finding all ALTERNATE-relationships in the given Petri net and recording the corresponding transitions. These transitions are the all non-determinism transition in the original Petri net and will be all transitions in the cause-effect-net.

- Step 2: Linking the causes and effects:
  Then analyzing all input places corresponding to these transitions using the OR-, AND-, SINGLE- and ALTERNATE-relationship and recording them. These associated input places are the causes. In addition, analyzing all output places corresponding to these transitions using the OR-, AND-, SINGLE- and ALTERNATE-relationship and recording them. These associated output places are the effects.

- Step 3: Generating cause-effect-net:
  According to the recorded transitions and the corresponding input and output places, a cause-effect-net of this Petri net can be created, which is similar to the generation of the Boolean graph.

- Step 4: Generating test case table:
  According to the generated cause-effect-net, each column in the test case table corresponds to a transition, and each row corresponds to the input and output places. When a transition has input and output places, the corresponding grid is marked as one.

- Step 5: Generating *Petrinets Testing* model:
  According to the generated test case table, the simple transition test case model can be generated by the *Petrinets Testing* language using PrettyPrinter. The test case model will be saved in the form of a file in the designated location to provide support for the next generating test code step.

### 4.2 Example

An specific example for the generate test case automatically from a Petri net, which is a simplified workflow of a cheap cookies machine (see Fig 6) with the following transitions:

First, coin is put into the cookies machine and will be checked whether is accepted. If the coin is accepted and the cookies button is pressed, the machine will receive the signal and give the cookies. At the same time, the amount of the cookies store and the number of cookies counter will be reduced. The machine will not accept a new signal until the money is cleared.

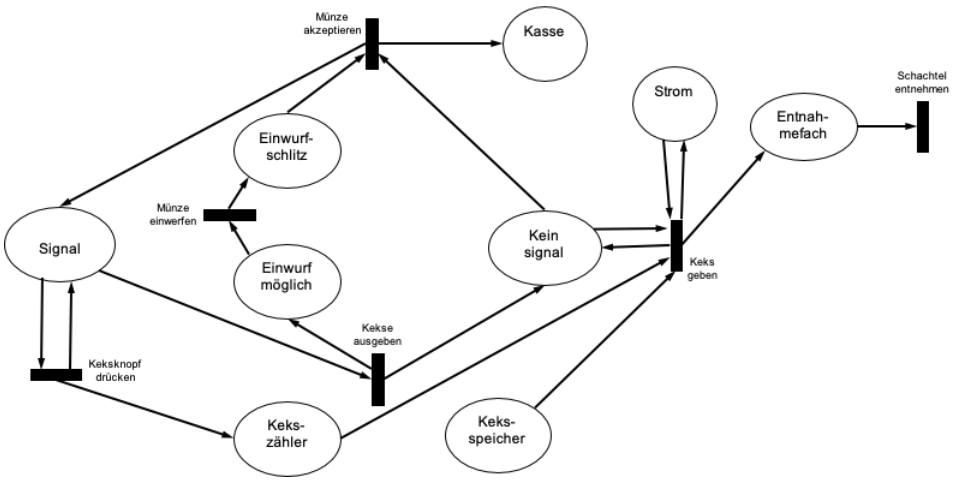


Fig. 6: Petri Net: Cheap Cookie Machine [3]

**Finding the causes and effects** Firstly, we traverse all transitions in order to search all ALTERNATE-relationships in this Petri net. So we can find two places and four transitions that they meet the ALTERNATE-relationship conditions. They are *Signal* places with *Keksknopf drücken* and *Kekse ausgeben* transitions, and *Keinsignal* with *Münze akzeptieren* and *Keks geben* transitions. Then we record these four transitions into a temporary table.

**Linking the causes and effects** Secondly, we analyze all input and output places about these four transitions using the OR-, AND-, SINGLE- and ALTERNATE-relationship and record them into a temporary table. The following is the temporary table about recorded transitions and input and output places:

1. Transition: *Keks drücken*

  - Input places: *Signal*
  - Output places: *Signal*, *Kekszähler*
2. Transition: *Kekse ausgeben*
  - Input places: *Signal*
  - Output places: *Einwurfmöglich*, *Keinsignal*
3. Transition: *Keks geben*
  - Input places: *Kekszähler*, *Keksspeicher*, *Strom*, *Keinsignal*
  - Output places: *Keinsignal*, *Entnahmefach*, *Strom*
4. Transition: *Münze akzeptieren*
  - Input places: *Keinsignal*, *Einwurfschlitz*
  - Output places: *Kasse*

**Generating cause-effect-net** According to the temporary table above, we can create a cause-effect-net (see Fig 7) of the original Petri net. At the same time, we mark transition *Keks drücken* as 1, taransition *Kekse ausgeben* as 2, transition *Keks geben* as 3 and transition *Münze akzeptieren* as 4 for the convenience of generating test case table.

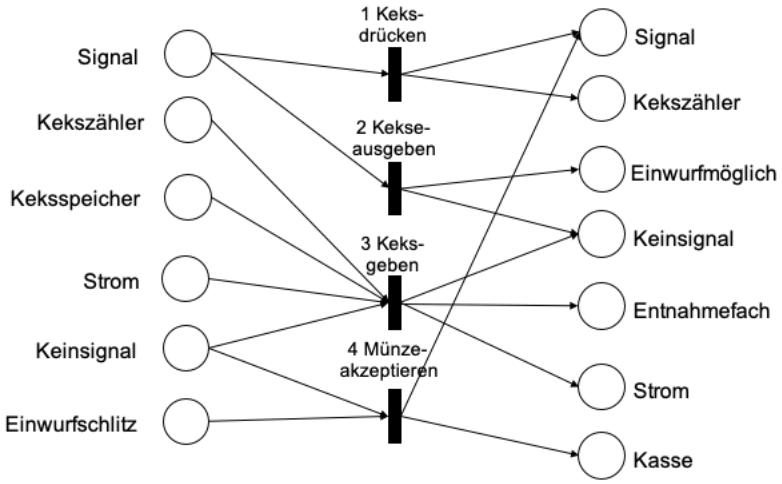


Fig. 7: cause-effect-net of Cheap Cookie Machine model

**Generating test case table** According to the cause-effect-net above, we can create a test case table (see Table 1). In this table, we use the previously marked numbers 1,2,3,4 to represent transitions. In addition, Causes (Input places) and Effects (Output places) are distinguished. For example, we mainly focus on the third column in the table and the number 3 on the header stands for transition *Keks geben*. Because *Kekszähler*, *Keksspeicher*, *Strom*, *Keinsignal* are

the input places of transition *Keks geben*, all corresponding positions in Causes are marked as 1. Similarly, *Keinsignal*, *Entnahmefach*, *Strom* are the output places of transition *Keks geben*, so all corresponding positions in Effects are marked as 1. Other transitions are marked by analogy. Finally we get a complete test case table about Petri net "Cheap Cookies Machine".

Table 1: Test case table of Cheap Cookie Machine model

| **TestCase** | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| **Causes:** | | | | |
| Signal | 1 | 1 | 0 | 0 |
| Kekszähler | 0 | 0 | 1 | 0 |
| Keksspeicher | 0 | 0 | 1 | 0 |
| Strom | 0 | 0 | 1 | 0 |
| Keinsignal | 0 | 0 | 1 | 1 |
| Einwurfschlitz | 0 | 0 | 0 | 1 |
| | | | | |
| **Effects:** | | | | |
| Signal | 1 | 0 | 0 | 1 |
| Kekszähler | 1 | 0 | 0 | 0 |
| Einwurfmöglich | 0 | 1 | 0 | 0 |
| Keinsignal | 0 | 1 | 1 | 0 |
| Entnahmefach | 0 | 0 | 1 | 0 |
| Strom | 0 | 0 | 1 | 0 |
| Kasse | 0 | 0 | 0 | 1 |

**Generating *Petrinets Testing* model** According to the test case table above, the program can automatically generate a *Petrinets Testing* model about transition test. Each column in the test case table will generate a testcase in the *Petrinets Testing* model. The corresponding Causes in the table will be used as *initial marking* in the model. Moreover, the corresponding Effects in the table will be used as *expect marking* in the model and the transition will be used as *simulate* in the model. Then the program uses the PrettyPrinter of the *Petrinets Testing* language to write the model into a file (see Listing 2).

## 5 Implementation

The base of the *Petrinets Testing* language is the MontiCore Language Workbench [6] 6.1.0 which is a fully featured framework for the development of domain-specific languages. Based on the grammar that we present above, it is able to generate, among many other tools, a parser for models that conform to the grammar which also translates the model to a powerful and easy to manipulate data structure, the Abstract Syntax Tree (AST). As code is generated even before

```
pntest CookieMachine_AutoTest {
    use petrinet CookieMachine_modified
    testcase KeksGeben_TransitionTest {
        initial marking {
            KeksZaehler 1,
            Keksspeicher 1,
            KeinSignal 1,
            Strom 1
        }
        simulate {
            KeksGeben
        }
        expect marking {
            KeinSignal 1,
            Entnahmefach 1,
            Strom 1
        }
    }
    ...
}
```

Listing 2: The *Petrinets Testing* model of Cheap Cookie Machine model

compile time it also provides a development infrastructure that is specifically suited to work with our DSL and is easy to extend with more features, including code generation. This also means that only the parts that specifically concern the functionality of the language models had to be implemented.

We use unit testing to evaluate our implementation, aiming for a high degree of test coverage.

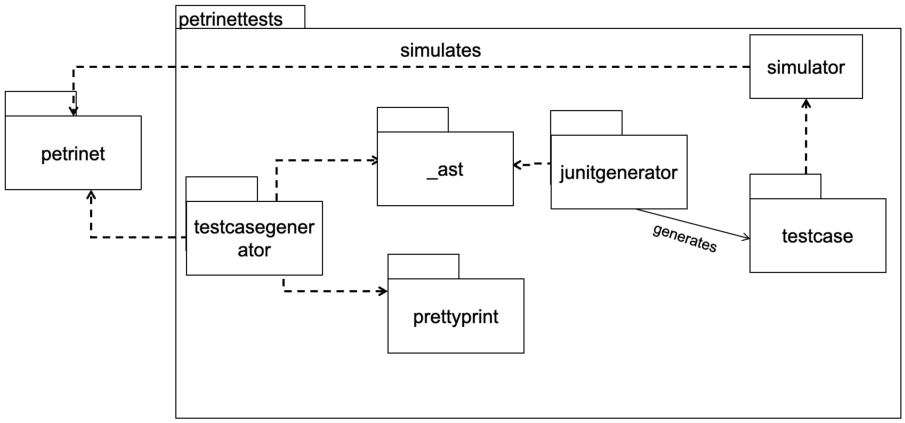


Fig. 8: The *Petrinets Testing* language Architecture

Figure 8 shows the top-level architecture based on packages as well as their interactions. **petrinet** here stands for the *petrinet4analysis* language. In the following, we describe the main roles of the individual packages.

- **testcasegenerator** contains the classes that implement the automatic generation of test cases based on models provided as *petrinet4analysis* AST. The mechanism that is implemented here is explained in detail in 4.
- **prettyprint** Using the PrettyPrinter, it is possible to inspect a *Petrinet Testing* model by printing it to a string. It makes use of the IndentPrinter provided by MontiCore, as well as the Visitor pattern. Its main use case is the export of ASTs that have been generated by the **testcasegenerator** package.
- **junitgenerator** provides functionality of generating executable testcases from *Petrinet Testing* models. This makes use of the MontiCore support for generation. Several templates are defined in the *FreeMarker* template language and passed to MontiCore along with the AST to generate from. This results in Java code that implements a JUnit test case.
- **simulator** could be seen as a run-time environment for generated test cases. The generated test cases require on this package which is able to perform operations on *petrinet4analysis* ASTs needed to perform the individual test steps. This includes setting a marking on a petri net and updating the marking based on transitions.

**Tool** In additon to that, we provide a command line interface as a user interface to our toolset. It supports both the generation of *Petrinet Testing* models from *petrinet4analysis* models using the **testcasegenerator** package, as well as the generation of JUnit tests from *Petrinet Testing* models. The tool is accessible through the command `java -jar petrinet-tests.jar`, where the file petrinet-tests.jar has to be the fully assembled jar file. The `--help` flag provides user information about the tool.

**Running tests** We provide a separate maven module *petrinet-tests-runner* that is able to parse models and execute all defined tests automatically. Both *petrinet4analysis* and *Petrinet Testing* models can be placed in to the module's `src/main/resources` directory to be considered. The generated files will be stored in the directory `target/generated-test-sources`.

This process is already included in the standard maven build process and can therefore be triggered by running `mvn install`.

**Workflow** A workflow that combines several use cases of our toolset could therefore look as follows. First, the user defines the model they wish to test as a *petrinet4analysis* model. Along with that, they can define hand-written test cases and write them using *Petrinet Testing* models. Then, using the PetrinetTestsTool, they can trigger the automatic generation of testcases described in section 4. This will generate additional *Petrinet Testing* model files. Both the *petrinet4analyis* and the *Petrinet Testing* models can be placed into the folder mentioned above. By running the maven goal install, the run of the runner module described above is triggered. It reads all defined models and generates executable JUnit tests that

execute the defined test cases. As the tests are directly run, the user is able to read the test results directly from the console.

### 5.1 Adaptations to *petrinet4analysis*

In order to be compatible with *petrinet4analysis*, a few adaptions to that implementation were required which are described in this section.

The *Petrinets Testing* language is supplemented by the grammar in *petrinets4analysis* language [3]. The *Petrinets Testing* language uses MontiCore 6.1.0 [1] to establish the project, but the *petrinets4analysis* language used the version of MontiCore was 5.0.3 [6]. In order to better use the grammar in *petrinets4analysis* language and seamlessly connect with our project, we decided to upgrade the MontiCore version used by *petrinets4analysis* language.

The *petrinets4analysis* language used MontiCore 5.0.3 to create the grammar language and implement the project. There are some differences between MontiCore 5 and MontiCore 6. In MontiCore, all grammars are dependent on the most basic grammar class, which is *de.monticore.literals.Literals* in MontiCore 5.0.3 but *de.monticore.literals.MCCommonLiterals* in MontiCore 6.1.0. There are all differences of *petrinets4analysis* language in MontiCore 5.0.3 and MontiCore 6.1.0 in Table 2

Table 2: Differences of *petrinets4analysis* language in MontiCore 5.0.3 and MontiCore 6.1.0

| | MontiCore 5.0.3 | MontiCore 6.1.0 |
|---|---|---|
| MC4 | de.monticore.literals.Literals | de.monticore.literals.MCCommonLiterals |
| | IntLiteral | NatLiteral |
| Java | Scope | PetrinetArtifactScope |
| | GlobalScope | PetrinetGlobalScope |
| | PetrinetSymbolTableCreator | PetrinetSymbolTableCreatorDelegator |
| | ResolvedSeveralEntriesException | ResolvedSeveralEntriesForSymbolException |

Note: MC4 is the filename extension of grammar file and Java is the filename extension of java file.

In the *petrinets4analysis* language upgrade, one of the difficulties is the definition of the number type. According to *de.monticore.literals.Literals* in MontiCore 5.0.3, the *petrinets4analysis* language used *IntLiteral* to define number type, but in *de.monticore.literals.MCCommonLiterals* of MontiCore 6.1.0, the *IntLiteral* is no longer used. Therefore we analyzed the role of *IntLiteral* in the *petrinets4analysis* language project, and finally we decided to replace it with *NatLiteral* in MontiCore 6.1.0. The return of getValue method of *IntLiteral* in MontiCore 5.0.3 is Optinal class in Java 8, but the return of getValue method of *NatLiteral* in MontiCore 6.1.0 is normal Int class, so we changed the corresponding algorithm logic in the place about *NatLiteral* in MontiCore 6.1.0.

In addition, another thing worth paying attention to during the upgrade is scope. The naming method of Scope and GlobalScope are different in the automatically generated code of MontiCore 5.0.3 and MontiCore 6.1.0. For example, Scope in the *petrinets4analysis* language of MontiCore 6.1.0 is PetrinetArtifactScope and GlobalScope is PetrinetGlobalScope. Moreover, we also made corresponding changes to the remaining related code about symbol table and exception handling.

### 5.2 Extensibility and Future Work

Many aspects of our design consider the need for future extensibility.

Many more advanced techniques of generated test cases exist that take into account different coverage criteria. In the future, they can be implemented by adding new algorithms to the **testcasegeneration** package.

Also, by swapping out the implementation of the **simulator** package, it would be possible to not only perform tests on *petrinet4analysis* models but to provide an actual system to test.

## 6 Conclusion

Petri nets provide a versatile tool for modeling processes. We further extended the petri net description language *petrinets4analysis* to comply to the current version of the MontiCore Language Workbench and on that basis developed a specific language, *Petrinets Testing* to model test cases on such petri nets. In addition we added capabilities for automated test case generation and provided an implementation for a generator utilizing cause effect nets.

The *Petrinets Testing* language offers a simple and concise yet powerful way of specifying deterministic test cases. Each of which is tested by simulating the referenced petrinet by firing the specified transitions in the correct order and asserting that the defined properties hold. By only focusing on necessary language constructs this enables others to create handwritten tests in order to check specific properties on a petrinet.

With the provided petri net simulation-interfaces we enable future developers to plug in their own implementations to be simulated as described in the test cases. Together with interfaces for automatic test case generators this creates an extensible framework which can be used to automatically apply the desired tests on the actual processes, modelled by the petri net.

The MontiCore Language Workbench [6] played a major role in developing this framework. It enabled us to focus on language design and tool-functionality by automatically generating all necessary steps to parse, verify and transform a language. Only a minimal additional codebase was needed to connect the two languages to work in unison. Leveraging these capabilities, other developers will also benefit from the consistent and easy to understand interface , which in turn further adds to the extensibility of the overall framework.

## References

1. Monticore - language workbench and development tool framework. https://github.com/MontiCore/monticore (2020)
2. Desel, J., Oberweis, A., Zimmer, T., Zimmermann, G.: Validation of information system models: Petri nets and test case generation. Proceedings of the IEEE International Conference on Systems, Man and Cybernetics **4**, 3401–3406 (1997). https://doi.org/10.1109/icsmc.1997.633178
3. Hein, P., Fiestelmann, M., Kinderreich, S.: petrinets4analysis : A Petri Net Modeling and Analysis Language Developed with MontiCore
4. Murata, T.: Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE **77**(4), 541–580 (1989)
5. Petri, C.A.: Kommunikation mit automaten (1962)
6. Rumpe, B., Hölldobler, K.: Monticore 5 language workbench. edition 2017 (2017)
7. Srivastava, P.R., Patel, P., Chatrola, S.: Cause effect graph to decision table generation. ACM SIGSOFT Software Engineering Notes **34**(2), 1–4 (2009)