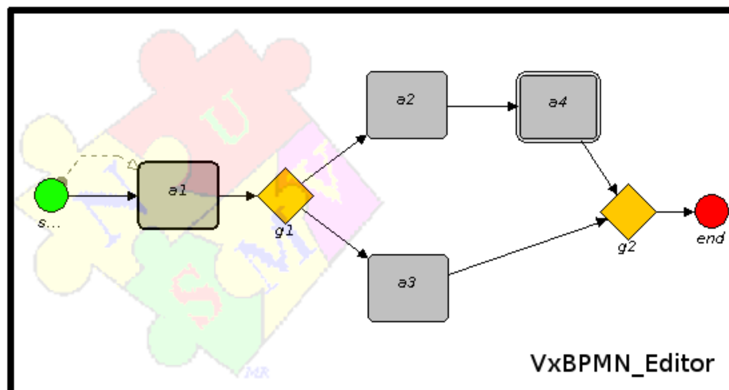
Internship Report

BUSINESS PROCESS MODEL CHECKING

using PDVI techniques & the NuSMV model checker tool

May 16, 2013



Andrés Tello, s2147149

Contents

1. Introduction **1**
1.1. Context 1

2. Implementation Phase **2**
2.1. Developing process 2
2.2. PDVI_NuSMV application 2
2.3. Detailed description 3

3. Conclusions **7**
3.1. Current state and future work 7

A. APPENDIX **8**
A.1. NuSMV model example 8
A.2. Kripke Structure and CTL XML schemas 8
A.3. System Requirements 12

1. Introduction

1.1. Context

Today Business Processes Management is used in many companies in the private and public sectors. Business Processes are translated into models which are then validated against some properties that the models have to meet. Model Checking is a technique for automatically verifying correctness of those models.

In the past, business process models were a representation of repetitive and rigid activities; nowadays those processes need to be flexible and adapt to the changing needs of the customers. Hence, the research group of Distributed Systems of the University of Groningen have proposed a technique to allow flexibility and re-usability in process modeling. Such approach is called PVDI: Process Variability through Declarative and Imperative techniques [2].

The aim of this internship is to build an application which using a existing model checker tool (e.g. NuSMV), validates models created using PVDI techniques.

2. Implementation Phase

In this chapter I explain in general terms the reasoning made before starting with the developing phase. In addition, I depict a list of the steps in which the developing process was decomposed. Then, I give an overview of how the implemented application works. Finally, I provide a detailed description of how each step was implemented, including the requirements and dependencies of the system.

2.1. Developing process

In the context of PVDI a process is defined as a directed graph consisting of activities, gates, and events. In addition, a *constraint* in PVDI is a CTL formula. Moreover, templates are used in PVDI as the basis for forming variants, and informally, a template is a process including constraints. Therefore, the processes' models created using PDVI techniques are a representation of state transition systems [7], and thereby, they also can be seen as kripke structures (KS) as proposed in [4]. Consequently those models can be evaluated using existing model checking tools such as NuSMV [1]. On the other hand, after studying the NuSMV documentation, it is clear that prior the evaluation, the models need to be translated to the NuSMV language. The *listing A.1* depicted in the *appendix A.1* shows an example of a model writen in NuSMV language.

After the reasoning made in the previous paragraph, the implementation was divided in the following steps.

1. Implement a representation of a Kripke Structure.
2. Implement a representation of the CTL formulas.
3. Implement a java procedure to translate the Kripke Structure and its associated CTL formulas to NuSMV language.
4. Integrate the NuSMV tool to the implemented java application.
5. Integrate the existing VxBPMN_Editor to the new java application

2.2. PDVI_NuSMV application

PDVI_NuSMV is a java application intended to validate a model created with the existing application VxBPMN_Editor, and using the Open Source NuSMV model checker tool. This application receives two xml files as an input. The first one is the Kripke Structure representation of the model, and the second is the representation of the CTL formulas. Then, an unmarshaling process is executed and the read xml documents are mapped to java objects. Once the mapping is executed, another process parses the inputs to NuSMV language. This process creates a file with extension .smv which serves

as an input for the NuSMV model checker tool. Then, the constraints expressed as CTL formulas in NuSMV language are validated and the output produced by NuSMV is shown in the user interface. Figure 2.1 depicts the PDVI_NuSMV application during execution. In the left side of the screen we can see the input model translated to NuSMV language, while the ouput produced by the NuSMV tool after validating the CTL constraints is shown in the right side of the screen. The application has only one command which executes all the tasks exaplained before. The user has to use the *open* command under the *File* menu and select the xml file of the KS.

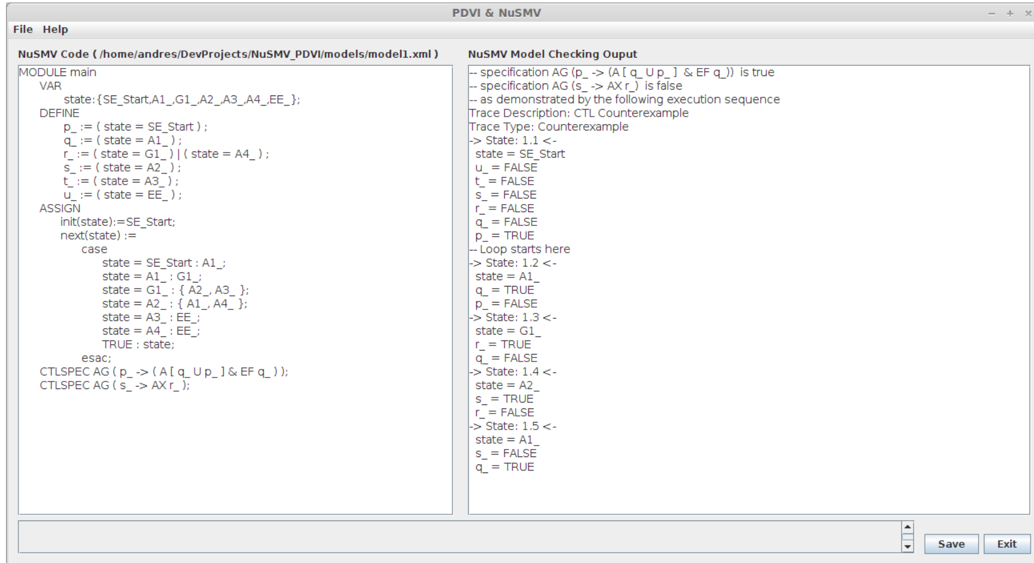


Figure 2.1.: Screenshot of the PDVI_NuSMV application

2.3. Detailed description

Kripke Structure and CTL formulas implementation. The first step is to think of a way of representing a Kripke Structure and its associated CTL formulas and offers both as inputs for the application. Since a Kripke Structure can be seen as a directed graph, then a straightforward approach is to represent it using XML language. In addition, to be consistent in the way the application is implemented, the same approach is used for representing the CTL formulas. The definition of the XML schemas used for representing both entities are shown in *appendix A.2*.

After choosing XML as the language for representing the KS and the CTL formulas, a way to map xml to java classes has to be defined. I did some research about which tools allow to bind xml documents to java objects. Since, I consider that a straightforward approach is to use JAXB (Java Architecture for XML Binding). JAXB make it easier to access XML documents from applications written in the Java programming language. Moreover, the existing VxBPMN_Editor application also uses JAXB for marshaling and unmarshaling xml documents. Therefore, due to the main goal of the PDVI_NuSMV application is to integrate NuSMV with the VxBPMN_Editor, the use of JAXB is self justified.

In order to the application read the input XML files correctly, both, the KS and CTL files has to have the same name. However, the one representing the KS has to have an .xml extension, while the CTL file has to have a .ctl extension (e.g., *my_model.xml* for the KS, and *my_model.ctl* for the CTL formulas). If the input files do not meet this requirements then an error message indicating that the

files could not be read is shown in the user interface.

The file which contains the implementation of this part is named *Process.java*, and it is contained in the package *rug.mc.util*.

Parsing the input to NuSMV language. NuSMV uses a propietary language to define the processes models. Therefore, once the input xml files are unmarshaled and their content is kept in memory, the next step is to translate those files to the input language of the NuSMV tool. A model writen in NuSMV language has a specific structure. In this particular case, to represent the Kripke Structure and CTL formulas in NuSMV language, it is necessary to use four clauses: VAR, DEFINE, ASSIGN and CTLSPEC. The four parts of the NuSMV model are depicted in the figure 2.1 (left size of the user interface). Since, the process for parsing the input to the NuSMV language was divided into four subprocesses which create each of those parts.

The VAR clause is necessary to define the variables that will be part of our model. However, the only variable created for every model is the *state variable*. This variable allows to declare all the possible states included in the Kripke Structure. As the name says, the DEFINE clause allows to define the atomic propositions of the process, as well as, in which states this variables will be true. The next part to define is the ASSIGN clause. It is allows to determine the initial state and its initial value. In addition, this clause also allows to specify the transitions from one state to another. Finally, the CTL constraints have to be set using the clause CTLSPEC. The output of this process is a string which contains all the generated NuSMV code and also a file with the extension .smv which represents a model in NuSMV convention.

The file which contains the implementation of these functionalities is named *Parser.java*, and it is contained in the package *rug.mc.parser*.

NuSMV integration. At this point, the translation from XML files, of the Kripke Structure and its corresponding CTL specifications, to the NuSMV input language is done. Then, the next step is to provide the generated model as input to the NuSMV tool for validation. Even though this seems to be a trivial task, many issues appeared during the implementation phase of this part.

The first thing to take into account is that NuSMV is a system written in the C programming language. Since, one approach is to create Java libraries out from C-header files using Java Native Access (JNA) and JNAerator. JNA provides Java programs easy access to native shared libraries without using the Java Native Interface. JNA's design aims to provide native access in a natural way with a minimum of effort. No boilerplate or generated glue code is required. [5]. JNAerator is a tool for the Java programming language which automatically generates the Java Native Access (JNA) code needed to call C and Objective-C libraries from Java code [6].

After some research with the aim of finding an existing API which performs this tasks, I found a project with Eclipse Public License - v 1.0 which provides a set of tools for the model checker NuSMV. One of the tools is an OSGi bundle containing a Java API to the NuSMV model checker itself, which allows to embed NuSMV in any Java application [3]. This OSGi bundle uses JNA and JNAerator in order to allow access to NuSMV from Java programs. Together with the jar files of the nusmv-tools, a library for windows, linux and osx systems is provided. In this particular case, the project was tested using a

Linux Mint 14 system of 64bits. However, the library *libnusmv.so* provided with the musmv-tools jar files, is a library for a 32bits system. Therefore, the application had to be set to run using a 32bits Java Virtual Machine. In addition, the library file *libnusmv.so* has to be in the */lib* directory.

The file which contains the implementation of these functionalities is named *NuSMVManager.java*, and it is contained in the package *rug.mc.util*.

VxBPMN_Editor integration. The last step of the developing process is to integrate the exiting application *VxBPMN_Editor* with the application under development. To implement this step, two main considerations has to be taken into account in order to properly integrates the *VxBPMN_Editor*.

First, due to every process has to be represented as a Finite State Machine, all the loops in a business model has to be unwinded to a depth of N. Figures 2.2 and 2.3 depict a model containing a loop and the same loop unwinded to a depth of two respectively.

Second, kripke structures do not know different gates. Any split in a kripke can be seen as an XOR-split. AND and OR gates need to be encoded differently. Figure 2.4 depicts a simple graph where we assume **G** is an AND split and merge gate. We would need to translate this to a kripke using only OR-splits while keeping the atomicity of each path, i.e., meaning **B** is always followed directly by **C**, and **D** by **E**), and being able to obtain the parallel AND information. The algorithm that I have to implement proposes to take all the paths and append them, then take all paths again, but move them alongside each other one step, then add this option to the above. Finally, we continue to do the above until all combinations have been processed. Figure 2.5 depicts the resulting kripke structure after applying the algorithm.

The file which contains the implementation of these functionalities is named *PathManager.java*, and it is contained in the package *rug.mc.util*. Due to the algorithm is not integrated with the rest of the application, this file has its own *Main()* function in order to be executed. This function just calls a method which receives a list of execution paths as input and combine them in the way that was explained in the previous paragraph.

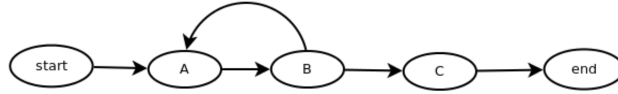


Figure 2.2.: Model with a loop

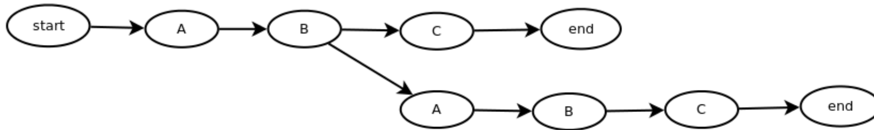


Figure 2.3.: Loop Unwinded to a depth of 2

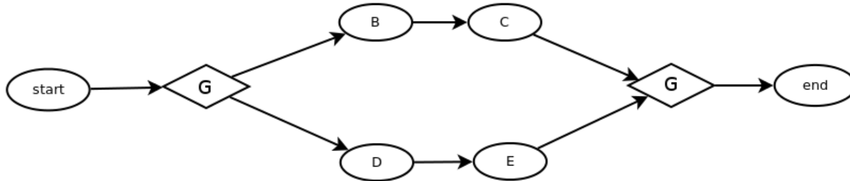


Figure 2.4.: Process with a split and merge AND gates

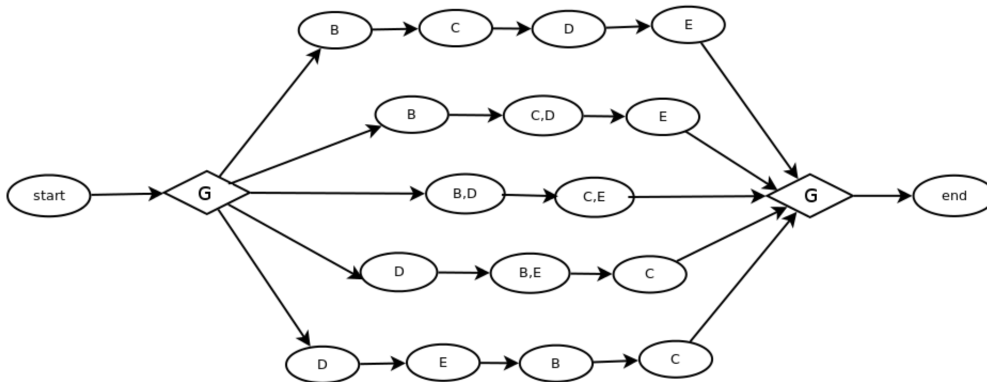


Figure 2.5.: Process after decomposing the AND gates

3. Conclusions

3.1. Current state and future work

As stated in chapter 2, the development process was divided in different steps in order to simplify and organize the work. At the end, most of the steps established in 2.1 where completed successfully. The application allows to read XML documents which represent a Kripke Structure and the CTL formulas associated to it. Those files are mapped to java objects and taken as an input for creating a model in NuSMV language. Then, the application invokes the NuSMV model checker tool in order to validate the model created in the previous step. The application receives the output of the NuSMV tool and shows those results to the end users. The last step is to integrate the application with the *VxBPMN_Editor*. Now, the algorithm for decomposing an AND- split and merge gate is implemented. However, a fully integrations with the *VxBPMN_Editor* was not possible due to time constraints. The algorithm was implemented receiving a list of *execution paths* as an input, which after the execution of the process are translated to all the possible combinations between them.

The main concern with the implementation of this algorithm is that it is an explosive algorithm. For instance, with three paths, each one with two states, the number of produced results is 37. Increasing the number of input paths only to five, the number of results increases to 2252 possible combinations. Since, in the worse cases, the computer's memory is not enough to finish the process. I tried to increase the heap size of the JVM; however, it only can increase up to 4GB, and even with this size a Stack Overflow exception can not be avoided.

On the other hand, in the current version of the *VxBPMN_Editor* application there is no chance to distinguish between different types of logical gates. Therefore, a real integration with the tool is not possible yet.

I consider that even though, the intended goal was not completely met, the PDVI_NuSMV application might be seen as the first prototype for the full integration of PDVI techniques and exiting and tested model checking tools such as NuSMV. The application might need to be improved in order to achieve a better performance. However, the implemented functionality is doing what it is expected to do.

The steps that still have to be done are the unwinding of loops in the model and the fully integration with the *VxBPMN_Editor*. In addition, the AND- split and merge decomposition could be improved. I think that it would be very useful to calculate the complexity of the algorithm. With that value it is possible to verify whether the algorithm is producing the correct number of results. Now we know that the number of results grows exponentially and at some point the computer's memory collapse. However, even though when the process ends normally, it is not possible to be sure if the number of results is accurate.

A. APPENDIX

A.1. NuSMV model example

Listing A.1: A simple NuSMV model example

```
MODULE main
  VAR
    state:{SE_Start,A1_,G1_,A2_,A3_,A4_,EE_};
  DEFINE
    p_ := ( state = SE_Start ) ;
    q_ := ( state = A1_ ) ;
    r_ := ( state = G1_ ) | ( state = A4_ ) ;
    s_ := ( state = A2_ ) ;
    t_ := ( state = A3_ ) ;
    u_ := ( state = EE_ ) ;
  ASSIGN
    init(state):=SE_Start;
    next(state) :=
      case
        state = SE_Start : A1_;
        state = A1_ : G1_;
        state = G1_ : { A2_, A3_ };
        state = A2_ : { A1_, A4_ };
        state = A3_ : EE_;
        state = A4_ : EE_;
        TRUE : state;
      esac;

CTLSPEC AG ( p_ -> ( A [ q_ U p_ ] & EF q_ ) );
CTLSPEC AG ( s_ -> AX r_ );
```

A.2. Kripke Structure and CTL XML schemas

Listing A.2: XML Schema of the Kripke Structure

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www.example.
    org/NewXMLSchema" xmlns:xsd="http://www.example.org/NewXMLSchema"
    elementFormDefault="qualified">

    <complexType name="AtomicProposition">
      <sequence>
        <element name="description" type="string"></element>
      </sequence>
```

```
    <attribute name="id" type="string"></attribute>
    <attribute name="name" type="string"></attribute>
  </complexType>

  <complexType name="Group">
    <attribute name="id" type="string"></attribute>
    <attribute name="name" type="string"></attribute>
    <attribute name="isFrozen" type="boolean" default="false"></attribute>
  </complexType>

  <complexType name="AtomicPropositions">
    <sequence>
      <element ref="xsd:AtomicProposition" maxOccurs="unbounded" minOccurs="1"></
          element>
    </sequence>
  </complexType>

  <complexType name="Label">
    <sequence>
      <element ref="xsd:StateId"></element>
      <element ref="xsd:apIDs"></element>
    </sequence>
    <attribute name="id" type="string"></attribute>
  </complexType>

  <complexType name="State">
    <sequence>
      <element ref="xsd:EventInfo" maxOccurs="1" minOccurs="0"></element>
    </sequence>
    <attribute name="id" type="string"></attribute>
    <attribute name="name" type="string"></attribute>
    <attribute name="type" type="xsd:StateType"></attribute>
    <attribute name="idGroup" type="string" use="optional"></attribute>
  </complexType>

  <complexType name="KripkeStructure">
    <sequence>
      <element ref="xsd:States"></element>
      <element ref="xsd:Relations"></element>
      <element ref="xsd:Labels"></element>
      <element ref="xsd:AtomicPropositions"></element>
    </sequence>
  </complexType>

  <simpleType name="StateType">
    <restriction base="string">
      <enumeration value="activity"></enumeration>
      <enumeration value="event"></enumeration>
      <enumeration value="gate"></enumeration>
    </restriction>
  </simpleType>

  <complexType name="Relation">
    <sequence>
```

```
      <element ref="xsd:CurrentState" maxOccurs="1" minOccurs="1"></element>
      <element ref="xsd:NextState" maxOccurs="1" minOccurs="1"></element>
    </sequence>
  </complexType>

  <complexType name="States">
    <sequence>
      <element name="state" type="xsd:State" maxOccurs="unbounded" minOccurs="1"></
          element>
    </sequence>
  </complexType>

  <complexType name="Relations">
    <sequence>
      <element ref="xsd:Relation" maxOccurs="unbounded" minOccurs="1"></element>
    </sequence>
  </complexType>

  <complexType name="Labels">
    <sequence>
      <element ref="xsd:Label" maxOccurs="unbounded" minOccurs="1"></element>
    </sequence>
  </complexType>

  <element name="States" type="xsd:States"></element>

  <element name="Relations" type="xsd:Relations"></element>
  <element name="Labels" type="xsd:Labels"></element>

  <element name="KripkeStructure" type="xsd:KripkeStructure"></element>


  <complexType name="EventInfo">
      <attribute name="isStart" type="boolean" default="false" use="optional"></
          attribute>
      <attribute name="isEnd" type="boolean" default="false" use="optional"></
          attribute>
      <attribute name="isGroupEvent" type="boolean" default="false" use="optional"></
          attribute>
  </complexType>

  <element name="AtomicPropositions" type="xsd:AtomicPropositions"></element>

  <element name="StateId" type="string"></element>

  <element name="LabelId" type="string"></element>

  <element name="AtomicProposition" type="xsd:AtomicProposition"></element>

  <element name="Relation" type="xsd:Relation"></element>

  <element name="EventInfo" type="xsd:EventInfo"></element>

  <element name="Group" type="xsd:Group"></element>
```

```
    <element name="Label" type="xsd:Label"></element>

    <element name="State" type="xsd:State"></element>

    <element name="apIDs" type="xsd:apIDs"></element>

    <complexType name="apIDs">
      <sequence>
        <element name="apID" type="string" maxOccurs="unbounded" minOccurs="1"></
            element>
      </sequence>
    </complexType>

    <element name="CurrentState" type="string"></element>

    <element name="NextState" type="string"></element>
</schema>
```

Listing A.3: XML Schema of the CTL formulas

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://www.example.org/CTL/
    " targetNamespace="http://www.example.org/CTL/">
    <complexType name="CTL">
      <sequence>
        <element ref="tns:CTL" maxOccurs="unbounded"
          minOccurs="0">
        </element>
      </sequence>
      <attribute name="id" type="string"></attribute>
      <attribute name="value" type="string" use="optional"></attribute>
      <attribute name="type" type="tns:CTLOperator"></attribute>
    </complexType>

    <complexType name="CTLFormula">
      <sequence>
        <element ref="tns:CTL" maxOccurs="unbounded" minOccurs="1"></element>
      </sequence>
    </complexType>

    <element name="CTL" type="tns:CTL"></element>
    <element name="CTLFormula" type="tns:CTLFormula"></element>

    <simpleType name="CTLOperator">
      <restriction base="string">
        <enumeration value="implies"></enumeration>
        <enumeration value="ap"></enumeration>
        <enumeration value="neg"></enumeration>
        <enumeration value="and"></enumeration>
        <enumeration value="or"></enumeration>
        <enumeration value="eq"></enumeration>
        <enumeration value="eg"></enumeration>
        <enumeration value="ex"></enumeration>
        <enumeration value="ef"></enumeration>
```

```
        <enumeration value="ag"></enumeration>
        <enumeration value="ax"></enumeration>
        <enumeration value="af"></enumeration>
        <enumeration value="eu"></enumeration>
        <enumeration value="au"></enumeration>
      </restriction>
    </simpleType>
</schema>
```

A.3. System Requirements

In this sections I describe the system requirements in order to run the application. The description is based on the developing environment where the application has been tested.

Java Virtual Machine:

jdk-7u17-linux-i586

NuSMV:

org.eclipselabs.nusmvtools.nusmv4j_0.1.0.v201204032116.jar
org.eclipselabs.nusmvtools.nusmv4j.source_0.1.0.v201204032116.jar
com.sun.jna_3.2.7.jar
com.ochafik.lang.jnaeratorruntime_0.9.7.jar
libnusmv.so – this file has to be included in the /lib directory.

User interface:

swing-layout-1.0.3.jar

Bibliography

[1] Roberto Cavada, Alessandro Cimatti, Gavin Keighren, Emanuele Olivetti, Marco Pistore, and Marco Roveri. Nusmv 2.5 user manual. `http://nusmv.fbk.eu/NuSMV/userman/v25/nusmv.pdf`, june 2011.

[2] H. Groefsema, P. Bulanov, and M. Aiello. Imperative versus declarative process variability: Why choose? Technical Report JBI 2011-12-6, University of Groningen, december 2012.

[3] Siamak Haschemi. textnusmv-tools. `https://code.google.com/a/eclipselabs.org/p/nusmv-tools/`.

[4] Johan van Benthem, Jan van Eijck, and Vera Stebletsova. Modal logic, transition systems and processes, 1994.

[5] Wikipedia. Java native access. `http://en.wikipedia.org/wiki/Java_Native_Access`, april 2013.

[6] Wikipedia. Jnaerator. `http://en.wikipedia.org/wiki/JNAerator`, march 2013.

[7] Wikipedia. State transition system. `http://en.wikipedia.org/wiki/State_transition_system`, february 2013.