STSimulator V271007
A Library to Simulate Symbolic Transition Systems

Lars Frantzen

Instituto di Scienza e Tecnologie della Informazione “Alessandro Faedo”
Consiglio Nazionale delle Ricerche, Pisa – Italy

Institute for Computing and Information Sciences
Radboud University Nijmegen – The Netherlands
lf@cs.ru.nl

October 27, 2007
Abstract

Symbolic Transition Systems (STSS) are transition systems with an explicit notion of data, and data-dependent control flow. They are a well-studied formalism in the domain of modeling and testing reactive systems. The STSIMULATOR Java library allows to model and simulate STSSs. This manual describes the functionality of the library, and its usage.


## Contents

1. **Introduction**  
   1.1 What this Library is not  
   1.2 Contact  

2. **Mandatory and Optional Software**  
   2.1 The STSimulator library (mandatory)  
   2.2 The treeSolver (mandatory)  
   2.3 The Quick Sequence Diagram Editor (optional)  
   2.4 The dot Tool (optional)  

3. **Modeling a Symbolic Transition System**  
   3.1 Creating the `SymbolicTransitionSystem` Object  
   3.2 Defining the Types  
      3.2.1 Simple Types  
      3.2.2 Complex Types  
   3.3 Type Instances  
      3.3.1 Simple Type Instances  
      3.3.2 Complex Type Instances  
   3.4 Defining the Location Variables  
   3.5 Setting the Services and Ports  
   3.6 Defining the Operations  
   3.7 Defining the Messages  
      3.7.1 Defining the Interaction Variables  
      3.7.2 Constructing the Messages  
   3.8 Defining the Switches  
      3.8.1 Defining the Switch Restrictions  
      3.8.2 Defining the Update Mappings  
      3.8.3 Constructing the Switches  
   3.9 Defining the Locations  
   3.10 Connecting Locations and Switches  
   3.11 Setting the Initial Location  
   3.12 Checking the Consistency of the STS  
   3.13 Attaching Parsers to the Switches
4 Simulating a Symbolic Transition System

4.1 Preliminary Steps

4.1.1 Creating a Logger

4.1.2 Setting up the Tools

4.1.3 Creating the Socket Connections

4.2 Creating the STSimulator Object

4.2.1 Initializing the Quick Sequence Diagram Editor

4.3 Simulating the STS

4.3.1 Getting the Status

4.3.2 Getting the Current Switches

4.3.3 Computing Solutions for Switches

4.3.4 Processing Messages

4.3.5 Processing Quiescence

4.3.6 Getting the Coverage Status

4.4 Cleaning Up

5 Implementing On-The-Fly Testing Algorithms

A The Dumont Grammar in BNF

B GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007
There, he finds Yori, a program written by Lora. After Tron breaks Yori out of her reporting routine, the two programs make their way to the I/O tower and confront Dumont, the keeper of the tower. He grants Tron access to the port, and Tron receives the critical instructions he needs from Alan in order to destroy the Master Control Program.

© Wikipedia

Chapter 1

Introduction

The STSimulator Java library allows to model and simulate Symbolic Transition Systems (STSs). Several types of STSs have been defined in the literature. We focus here on the variants used for testing purposes, see for instance [3, 4, 11]. Especially [3] served as the formal foundation for this library. In that setting, an STS consists of the following structural ingredients:

- data types
- variables
- messages (also known as gates)

The following simple types are supported by the simulator:

- (positive) integers\(^1\)
- booleans
- enumerations
- strings

There is an experimental data type representing doubles with a fixed precision. Such doubles can be mapped to integers, which allows constraint solving over finite domains.

\(^1\) The restriction to positive integers comes from GNU Prolog, which is used for constraint solving. GNU Prolog can only deal with positive integers. Future versions of this library will probably use a different constraint solver, which also allows negative integers.
Simple types can be combined to *complex types*. This is similar to complex types known from XML Schema, or *records* from other programming languages. Though, it is not possible to define recursive complex types (like *lists*) in this manner. Complex types can, for instance, be used to model *objects* of *classes* in object oriented systems.

Variables serve the same purpose as in iterative programming languages, they are the memory of the STS, and can change their value. They are more precisely referred to as *location variables*. Every message is either an input sent to the modeled system, or on output received from the system (there can be also unobservable messages which just update variables). Messages can have parameters to communicate data values. Such parameters are also modeled via variables, so called *interaction variables*.

As an additional feature, this library assigns messages to *operations*, operations to *ports*, and ports to *services*. By so doing it is possible to model interfaces known from component-based systems like Web Services. In its current state of the library, this is just a means for an additional structuring of the messages, and has no significant impact on the algorithms of the library.

Furthermore, an STS consists of the following behavioral ingredients:

- *locations* (also known as *states*)
- *switches* (also known as *transitions*)

Locations are a way to structure the STS via representative conditions in which the modeled system is supposed to be. From a formal point of view, they are not really needed, but they aid in the structuring and understanding of the model. Switches are the places where communication takes place. Each switch is directed, and connects two locations. Moreover, a switch

- communicates a *message*,
- is constrained by a *switch restriction* (also known as a *guard*), and
- can update the values of the location variables via an *update mapping*.

To define switch restrictions, a simple language called *Dumont* can be used. For integers and boolean expressions it offers the most important operators known from common programming languages. For enumerations and strings the only supported operator is (in)equality.

Once an STS is defined, it can be used for several purposes. This library focuses on features needed for on-the-fly testing algorithms, as for instance the one described in [2]. The main task here is to *simulate* the STS. To do so, the simulator keeps track of the set of states in which the STS can currently be. Due to nondeterminism, there may be several of such states in the set. Such a state is determined by a location, and a valuation of the location variables. States are called *instantiated locations* in the simulator. Initially the STS is set into its *initial states*, which are determined by an initial location, and an initial valuation of the location variables.
CHAPTER 1. INTRODUCTION

The main simulation procedure consists now in giving (input or output) messages to the simulator, which computes the next set of possible instantiated locations. To do so, it checks at which current instantiated location there is a switch with the given message. If also the corresponding switch restriction is true, the switch can fire, leading to a new instantiated location.

It is also possible to let the simulator on its own construct a message for a given switch, such that the switch can fire. To do so, the simulator has to find a solution for the switch restriction. It uses the GNU Prolog constraint solver [5] to compute this.

Finding solutions to switch restrictions is also needed for a special observation which can be simulated — quiescence. This has the effect, that out of the set of the current instantiated locations all those are removed which can fire an output message, or on unobservable message. Quiescence is a crucial concept in several testing relations, like ioco [12]. For a more detailed and formal description of the simulation approach, please see [2].

1.1 What this Library is not

Other interesting tasks to be done with an STS, which are not supported by the library, are for instance:

- all kinds of advanced static analysis, like:
  - deadlock/lifelock analysis
  - symbolic reachability (e.g. regarding coverage criteria)
- composition of STSs
- generation of symbolic test suites (in contrast to on-the-fly testing)

The library is a prototype, and under continuous development. We are busy with improving and extending the feature set also in these directions. Check the webpage [8] for updates. If you want to be kept informed via email, please contact us.

1.2 Contact

Do not hesitate to contact Lars Frantzen (lf@cs.ru.nl) if there are questions, comments, remarks, etc.

>>>) Join the Development! <<<

We are aware that this manual, the sources, and the Javadoc, are far from being complete and optimal. It is quite some work to write all this stuff. Someone interested in joining the development is most welcome!
Overview

Chapter 2 explains which software is needed to work with the library. Chapter 3 shows how to model an STS with the library. The simulation of an STS is explained in Chap. 4. The way on-the-fly testing algorithms can be implemented with the library is explained in Chap. 5.
Chapter 2

Mandatory and Optional Software

This chapter points to the mandatory and optional software packages relevant for the STSimulator. It tells where to get the software, and gives installation hints.

2.1 The STSimulator library (mandatory)

The library itself can be downloaded at [8]. The .zip archive contains the following:

- the STSimulator.jar Java library
- the javadoc directory with the API documentation
- the manualXXXYYZZ.pdf manual (this file)
- the COPYING license file

We assume the reader is familiar with Java libraries, and Javadoc API documentations. Just adding those to a project should do the job in all modern Java IDEs. The sources are part of the STSimulator.jar archive. The library is licensed under the GPLv3, which can be also found in Appendix B.

2.2 The treeSolver (mandatory)

To do its job, the STSimulator needs access to the GNU Prolog constraint solver via a socket connection. To facilitate this, we have wrapped the solver in an executable offering such a socket connection. It is called TREE SOLVER and can be downloaded at [9]. The site provides executables for Windows and Linux. It can also be compiled on other systems, see the GNU Prolog site [5] for supported platforms. If there are problems compiling, please contact us.

It is not necessary to understand how the TREE SOLVER works to use the library, it just needs to be running. It is invoked with a single argument for the port to be used. The following command starts the solver on the localhost and port 60002:
$ treeSolver 60002

treeSolver V180607
By Lars Frantzen (lars@frantzen.info)
A socket interface to the constraint solver of GNU Prolog
GNU Prolog is copyright (C) Daniel Diaz
Listening on localhost port 60002

That is all. To use the library, a socket connection has to be established to the TReESOLVER
to construct the simulator object. This will be explained in Chap. 4.

2.3 The Quick Sequence Diagram Editor (optional)

An STS models an interface of a system. The simulation of an STS can be seen as an
interaction between a user of the system, and the system itself. One common way to
visualize such a message-exchange is via an UML sequence diagram [10]. The QUICK
SEQUENCE DIAGRAM EDITOR is a tool which allows to display such sequence diagrams in
realtime while the simulation takes place. Whenever the library simulates a message, this
message can be sent to the QUICK SEQUENCE DIAGRAM EDITOR via a socket connection,
which updates its displayed sequence diagram accordingly.

Note, though, that this only works for passive systems, meaning that an output message
is always the response to a preceding input message. Hence, the system never actively sends
an output message without being requested beforehand. This is due to some restrictions
of the input language of the QUICK SEQUENCE DIAGRAM EDITOR, and may change in
the future.

The QUICK SEQUENCE DIAGRAM EDITOR can be downloaded at [7]. Please refer to
its documentation on how to enable its realtime diagram server.

2.4 The dot Tool (optional)

Once an STS is modeled via the library, it exists as a Java object. Such an object can
be visualized as a graph, where the nodes correspond to the locations, and the edges
correspond to the switches. The library can produce a textual representation of the STS,
which is understood by the DOT tool. This tool transforms this textual representation in
such a graph. It also takes care of a proper graph layout.

The DOT tool is part of the GRAPHVIZ toolsuite, which can be downloaded at [6].
Please refer to its documentation on how to generate the graphs.
Chapter 3

Modeling a Symbolic Transition System

This chapter explains the steps needed to model an STS via the library. It is more meant as a pragmatic guide than as a complete reference. Please refer to the Javadoc for further details.

3.1 Creating the SymbolicTransitionSystem Object

First of all we create an (empty) object of the class SymbolicTransitionSystem as follows:
SymbolicTransitionSystem sts = new SymbolicTransitionSystem();

3.2 Defining the Types

All type-related classes are in the stsimulator.types package. We do not explicitly mention the package in the remainder. Each type subclasses the Type class.

3.2.1 Simple Types

The library supports the following simple types:
- positive integers via the ST_PosInt class
- booleans via the ST_Boolean class
- strings via the ST_String class
- enumerations the Enumeration class

All simple types subclass the SimpleType class. To refer later to a specific type, an object representing the type has to be created. For integers, booleans, and strings, just calling the empty constructor is sufficient, for instance:
SimpleType thePosInt = new ST_PosInt();
SimpleType theBoolean = new ST_Boolean();
SimpleType theString = new ST_String();

An enumeration represents an array of strings. The constructor of an enumeration type additionally demands this array, for instance:

String[] productnames = new String[2];
productnames[0] = "FOO";
productnames[1] = "BAR";
Enumeration product = new Enumeration(new QName("product"), productnames);

Mind that we deal here with the stsimulator.types.Enumeration class, not Java enumerations.

3.2.2 Complex Types

A complex type represents an array of types, either simple or complex. But it is not possible to define recursive types like lists in this manner. A complex type has a name, and fields. The fields is a two-dimensional array of (name,type) pairs. Continuing the example:

Object[][] fields = new Object[3][2];
fields[0][0] = "product";
fields[0][1] = product;
fields[1][0] = "quantity";
fields[1][1] = thePosInt;
fields[2][0] = "agent";
fields[2][1] = theString;
ComplexType quote = new ComplexType(new QName("quote"), fields);

We have defined here a complex type called quote consisting of three field elements: a product of enumeration type product, a quantity of type thePosInt, and an agent of type theString. We could call these field elements also the elements, or attributes of the complex type.

Adding the Types to the STS

After all necessary type-objects have been defined, they are added to the STS via the addType method. For instance:

sts.addType(thePosInt);
sts.addType(theBoolean);
sts.addType(theString);
sts.addType(product);
sts.addType(quote);
3.3 Type Instances

To represent values of a specific types, type instances are used. For each type mentioned above, a specific type instance class is declared. All these classes subclass the TypeInstance class. Type instances are for example needed when defining the initial values of interaction variables.

3.3.1 Simple Type Instances

The simple type instances are represented as follows:

- positive integer values via the ST_PosIntInstance class
- boolean values via the ST_BooleanInstance class
- string values via the ST_StringInstance class
- enumeration values the EnumerationInstance class

An instance of an integer, boolean, or string simple type is constructed by giving the corresponding primitive Java value, for instance:

```java
SimpleTypeInstance anInt = new ST_IntInstance(42);
SimpleTypeInstance aBoolean = new ST_BooleanInstance(true);
SimpleTypeInstance aString = new ST_StringInstance("a string");
```

An enumeration type instances is a single element out of the string array a given enumeration type represents. An enumeration type instance refers to its corresponding enumeration type when being constructed. For instance:

```java
EnumerationInstance theFoo = new EnumerationInstance(product, "FOO");
```

3.3.2 Complex Type Instances

A complex type instance is an array of type instances. The types of the type instances must match the types of the fields dictated by the complex type the instance is meant for. The following defines a type instance for the complex type quote defined above:

```java
Object[] fieldvalues = new Object[3];
fieldvalues[0] = theFoo;
fieldvalues[1] = anInt;
fieldvalues[2] = aString;
ComplexTypeInstance aQuote = new ComplexTypeInstance(fieldvalues);
```
3.4 Defining the Location Variables

Location variables are the global variables of the STS. They have a name, type, and an initial value. Their values can be checked in switch restrictions to determine if a switch can fire, and new values can be assigned to them via an update mapping after a switch has fired. They are represented by objects of the class LocationVariable. Its constructor is:

```java
public LocationVariable(String name,
                        Type type,
                        TypeInstance value)
```

We have seen in the previous section how to define type instances. They are needed here to define the initial values of the location variables.

In some cases it is convenient to have a means to automatically set predefined initial values. Every type allows to do this via the `generateInitialInstance()` method. For the simple types, these values are:

- the 0 for positive integers
- `false` for booleans
- the empty string "" for strings
- the first array element for enumerations

Complex types delegate the construction of initial instances to the individual types of the fields. For instance, the initial instance for the `quote` complex type is `[FOO, 0, ""]`. The following code defines a location variable of type `quote`, and a location variable of type `theBoolean`. Both are constructed with the initial value generated automatically:

```java
LocationVariable quoteIssued = new LocationVariable("quoteIssued", quote,
                                                     quote.generateInitialInstance());
LocationVariable orderSucceeded = new LocationVariable("orderSucceeded",
                                                        theBoolean, theBoolean.generateInitialInstance());
```

Adding the Location Variables to the STS

After all necessary location variables have been defined, they are added to the STS via the `addVariable` method. For instance:

```java
sts.addVariable(quoteIssued);
sts.addVariable(orderSucceeded);
```
3.5 Setting the Services and Ports

One of the original motivations for this library was to model Web Services. Due to that, concepts from that domain show up at some points. Especially the Web Service Description Language [1] served as a guideline on how to structure systems.

At the current state of the library, only a single service with a single port are supported. In other words, setting these parameters has no further relevance for the functionality of the library. Future versions may be able to deal with several ports, but there are still open theoretical and practical questions on how to do that. Pragmatically, setting some arbitrary values here is sufficient in this version, for instance:

```java
Service supplierService = new Service("SupplierService");
Port supplierPort = new Port("SupplierPort");

supplierService.addPort(supplierPort);

sts.addService(supplierService);
```

3.6 Defining the Operations

Operations are an additional means to group messages. By so doing it is possible to model operation calls known from programming languages, or WSDL, via messages. In the current state, we adopt the WSDL model. Each operation has one of four possible kinds:

- **request response**
  an input message followed by an output message

- **solicit response**
  an output message followed by an input message

- **one-way**
  an input message

- **notification**
  an output message

For instance, a (remote) procedure call would correspond here to a request-response operation. For the functionality of the simulator the operations have no further relevance. They come into play when the simulator is used, for instance, to test real systems. To construct an operation, a name and its kind are needed. The kinds are defined in the Java `OperationKind` enumeration. For instance:
Operation requestQuote = new Operation("requestQuote",
        OperationKind.REQUESTRESPONSE);
Operation makePayment = new Operation("makePayment", OperationKind.ONEWAY);

Adding the Operations to the STS

After all necessary operations have been defined, they are added to the STS via the
addOperation method. For instance:
sts.addOperation(requestQuote);
sts.addOperation(makePayment);

3.7 Defining the Messages

When a switch can fire, it communicates via sending or receiving a message. There are
three kinds of messages:

- **input message**
  a message sent to the modeled system

- **output message**
  a message received from the modeled system

- **unobservable message**
  just serves to switch the location, and may update the values of the location variables

The kinds are defined in the Java MessageKind enumeration. The constructor for a message is:

```
public Message(String name,
        MessageKind kind,
        Operation op,
        Port port,
        ArrayList<InteractionVariable> type)
```

It can be seen here, that a message always relates to an operation, and a port. It is
important, that the right kind and number of messages relate to an operation, i.e.:

- **request response** – an input message and an output message
- **solicit response** – an input message and an output message
- **one-way** – an input message
- **notification** – an output message

The list of parameters of a message is called the type in the constructor, which is an
ArrayList of interaction variables.
3.7.1 Defining the Interaction Variables

An interaction variable is similar to a location variable, it just does not need an initial value to be set when being constructed. This is due to the fact that they are used to model parameters of messages. Given, for instance, an input message called pay having a first parameter called reference of type ST_PosInt, and a second parameter called address of type ST_String. These parameters are modeled via two interaction variables with the given names and types. For instance:

```java
InteractionVariable reference = new InteractionVariable("reference", thePosInt);
InteractionVariable address = new InteractionVariable("address", theString);
```

Adding the Interaction Variables to the STS

After all necessary interaction variables have been defined, they are added to the STS via the addVariable method. For instance:

```java
sts.addVariable(reference);
sts.addVariable(address);
```

3.7.2 Constructing the Messages

What remains to be done is putting the interaction variables in an ArrayList to represent the type. Then the messages can be constructed. For instance:

```java
ArrayList<InteractionVariable> type_pay = new ArrayList<InteractionVariable>();
type_pay.add(reference);
type_pay.add(address);
Message pay = new Message("pay", MessageKind.INPUT, makePayment, supplierPort, type_pay);
```

We relate here the pay message to the makePayment operation. Since the makePayment operation is of kind oneway, just a single input message has to be assigned to it – it is now fully defined.

Constructing an Unobservable Message

An unobservable message does not have any interaction variables. It also does not relate to an operation. It only relates to a port, to which the unobservable message belongs. As indicated before, the port has no further meaning to the simulator, just pass here the one port created before. There is a static method generateUnobservableMessageForPort in the Message class which can be used to create an unobservable message. For instance:

```java
Message unobservable = Message.generateUnobservableMessageForPort(supplierPort);
```
Adding the Messages to the STS

After all necessary messages have been defined, they are added to the STS via the `addMessage` method. For instance:

```java
sts.addMessage(unobservable);
sts.addMessage(pay);
```

3.8 Defining the Switches

A `switch` is sometimes also called a `transition`. Switches are the places where communication takes place. Each switch is directed, and connects two locations. Moreover, a switch

- communicates a `message`,
- is constrained by a `switch restriction` (also known as a `guard`), and
- can update the values of the location variables via an `update mapping`.

The switch constructor is:

```java
public Switch(Message message,
             String switchRestriction,
             String updateMapping)
```

Note that there is another constructor which allows to also fix the target location.

3.8.1 Defining the Switch Restrictions

To define switch restrictions, a simple language called `Dumont` is used. For integers and boolean expressions it offers the most important operators known from common programming languages. For enumerations and strings the only supported operator is (in)equality.

The Dumont Language

There are four kinds of `literals`:

- **integer literal** – a numberstring, e.g. 3423432
- **boolean literal** – true or false
- **string literal** – a string between double quotes, e.g. "Hello World"
- **enumeration literal** – a string of the form `element@enumname`, where `enumname` is the name of an enumeration type, and `element` is one of its elements, e.g. FOO@product
CHAPTER 3. MODELING A SYMBOLIC TRANSITION SYSTEM

An identifier is a name of a (location or interaction) variable. If a variable has a complex type, then you can refer to the fields by the common dot-notation, e.g. \texttt{quoteIssued.product}.

Each literal and identifier is also an expressions, which has a corresponding type, i.e. one of \texttt{integer-exp}, \texttt{boolean-exp}, \texttt{string-exp}, \texttt{complex type-exp}, or \texttt{enumeration-exp}. One can combine literals and identifiers via operations, yielding a more complex expression, which again has a type. Next we mention the operations currently supported by the parser. First we mention the operations which lead to an expression of type \texttt{boolean-exp}:

- \texttt{==} compares \texttt{boolean-exp}, \texttt{integer-exp}, \texttt{string-exp} and \texttt{enumeration-exp} for equality
- \texttt{!=} compares \texttt{boolean-exp}, \texttt{integer-exp}, \texttt{string-exp} and \texttt{enumeration-exp} for inequality
- \texttt{<} compares \texttt{integer-exp} for being less than
- \texttt{<=} compares \texttt{integer-exp} for being less than or equal
- \texttt{>} compares \texttt{integer-exp} for being greater than
- \texttt{>=} compares \texttt{integer-exp} for being greater than or equal
- \texttt{&&} is the logical and of two \texttt{boolean-exp}
- \texttt{||} is the logical or of two \texttt{boolean-exp}
- \texttt{!} is the logical not of a \texttt{boolean-exp}

Next we mention the operations which lead to expressions of type \texttt{integer-exp}:

- \texttt{++} increments (+1) an \texttt{integer-exp}
- \texttt{--} decrements (-1) an \texttt{integer-exp}
- \texttt{+} adds two \texttt{integer-exp}
- \texttt{-} subtracts two \texttt{integer-exp}
- \texttt{*} multiplies two \texttt{integer-exp}
- \texttt{\%} is the remainder operator for \texttt{integer-exp}
- \texttt{/} divides two \texttt{integer-exp}

A switch restriction is an expression of type \texttt{boolean-exp}.

Sometimes one wants to express an empty switch restriction, meaning an expression which has the logical value \texttt{true}. This cannot be done by giving an empty string, nor by writing just the boolean literal \texttt{true}, one has to write \texttt{true == true} here. One also cannot just mention the name of a variable of type boolean, like \texttt{p}, or \texttt{!p}, it is necessary to write \texttt{p == true}, or \texttt{p == false}. This is due to some internal design decisions, and may change in further versions.
3.8.2 Defining the Update Mappings

There is another operator we have not mentioned yet, it is the assignment operator =. Here we assign a value to a location variable. An update mapping can consist of several such assignments, which all must be terminated by a ; (semicolon). It is also possible to have no assignment at all by giving the empty string "", meaning that all location variables remain unchanged. The BNF of the Dumont language can be found in Appendix A.

3.8.3 Constructing the Switches

Assuming we want to construct a switch for the pay message. In its switch restriction and update mapping we can refer to all location variables, and to the interaction variables of the pay message’s type (i.e., reference and address). The following switch restriction says that the parameter reference shall be greater than 41 added to the field value quantity of location variable quoteIssued:

String sr = "reference > quoteIssued.quantity + 41";

The following update mapping sets the location variable orderSucceeded to true:

String um = "orderSucceeded = true;";

Now we can construct a switch as follows:

Switch aSwitch = new Switch(pay, sr, um);

Note that when the switch carries an unobservable message, the switch restriction and update mapping are only allowed to deal with location variables (since an unobservable message does not have any interaction variables).

Adding the Switches to the STS

After all necessary switches have been defined, they are added to the STS via the addSwitch method. For instance:

sts.addSwitch(aSwitch);

3.9 Defining the Locations

A location has a name, and keeps track of its outgoing switches. One constructor just sets the name, for instance:

Location l1 = new Location("1");

Another constructor allows to additionally set the set of outgoing switches via a HashSet. For instance:
HashSet<Switch> os = new HashSet<Switch>();
    os.add(aSwitch);
    Location l1 = new Location("1", os);

    A single outgoing switch can also be added to the location via the addOutgoingSwitch method, see next section.

**Adding the Locations to the STS**

After all necessary locations have been defined, they are added to the STS via the addLocation method. For instance:

    sts.addLocation(l1);

### 3.10 Connecting Locations and Switches

A single outgoing switch can be added to a location via the addOutgoingSwitch method. The target of a switch can be set via the setTarget method. For instance:

    l1.addOutgoingSwitch(aSwitch);
    aSwitch.setTarget(l1);

### 3.11 Setting the Initial Location

We have already fixed the initial values of the location variables. To determine the initial state (called the initial instantiated location in the terminology of this library) of the STS, it is necessary to also set an initial location. This is done via the setCurrentLocation method. For instance:

    sts.setCurrentLocation(l1);

### 3.12 Checking the Consistency of the STS

The SymbolicTransitionSystem class can do some very basic consistency checks of the STS via the checkConsistency method. Please refer to the Javadoc for details. Note that it is necessary for the coming steps to have modeled a consistent STS. For instance:

    System.out.println("Consistency result = " + sts.checkConsistency());

A consistent STS returns a 0 here.
3.13 Attaching Parsers to the Switches

When we have created the switches, we passed a switch restriction and an update mapping as a string to the constructor. To transforms these strings into real parsers, the `attachParsersToSwitches` method has to called. For instance:

```java
sts.attachParsersToSwitches();
```

Here three exceptions may be thrown:

- `SymbolicTransitionSystem.NonConsistentSTSException`
  - the STS is not consistent

- `SymbolicTransitionSystem.SwitchParseException`
  - the switch restriction or update mapping does not conform to the Dumont grammar

- `DumontParser.IllegalArgumentException`
  - a static type check has failed

3.14 Displaying the STS via dot

To display the modeled STS as a graph, the DOT tool can be used. To generate the input understood by DOT, the `asDotDigraph` method from the `SymbolicTransitionSystem` can be used, for instance:

```java
System.out.println(sts.asDotDigraph());
```

For the running example of this chapter, we would get:

```dot
digraph STS {
1 -> 1 [label = "makePayment?<reference:ST_PosInt,address:ST_String>
[reference > quoteIssued.quantity + 41]
orderSucceeded = true;"];
}
```

Giving this to DOT creates the following graph:

![Graph](image)

We have modeled just one location with a self-looping switch. The first line on the switch says:

`makePayment?<reference:ST_PosInt,address:ST_String>`
Firstly there the operation name `makePayment`. The question mark `?` indicates, that the input message of the operation is meant (which we have called `pay`, the name is omitted in the graph). An output message would be marked with an exclamation mark `!`. The rest of the line repeats the type of the message. The second line goes like this:

```plaintext
[reference > quoteIssued.quantity + 41]
```

Here the switch restriction is given in square brackets. Finally, the third line gives the update mapping:

```plaintext
orderSucceeded = true;
```
Chapter 4

Simulating a Symbolic Transition System

This chapter explains the features available for simulating an STS. We focus here on the most relevant features, there are more which can be looked up in the Javadoc.

4.1 Preliminary Steps

4.1.1 Creating a Logger

The simulator needs a Java Logger to which it passes logging messages. For instance:

```java
Logger logger = Logger.getLogger("STSimulator_Logger");
```

4.1.2 Setting up the Tools

We assume from now on that the treeSolver is running on port 60002. Without a running treeSolver no simulation can be done. We also assume that the Quick Sequence Diagram Editor is running on port 60001. The editor is not mandatory, one can simply skip the steps which relate to it in the remainder.

4.1.3 Creating the Socket Connections

Before creating the simulator object, it is necessary to set up socket connections to the tools. For the treeSolver this could look like this:

```java
String solverHost = "localhost";
int solverPort = 60002;
Socket solverSocket = new Socket(solverHost, solverPort);
```

Note that here some exceptions may be thrown. Note also, that the treeSolver sends a welcome message through the socket. Remove that one from the stream before proceeding, for instance with:
new BufferedReader(new InputStreamReader(
solverSocket.getInputStream())).readLine();

The same way a socket to the Quick Sequence Diagram Editor can be set up (no welcome message sent in this case).

4.2 Creating the STSimulator Object

Now we can create the simulator object by passing a SymbolicTransitionSystem object (see last chapter), the socket to the TreeSolver, and the logger:

    sim = new STSimulator(sts, solverSocket, logger);

Also here check the possible exceptions. The STS is now set in its initial states, which are determined by the initial values of the location variables, and the initial location.

4.2.1 Initializing the Quick Sequence Diagram Editor

In case the Quick Sequence Diagram Editor shall be used to visualize the simulation, pass its socket to the simulator via the setDisplaySocket method, for instance:

    sim.setDisplaySocket(qsdeSocket,"User");

The string parameter sets the name for the classifier which represents the user of the simulated system.

4.3 Simulating the STS

The simulator keeps track of the current possible states in which the STS can be. Such a state is called an instantiated location in the terminology of the simulator. Doing simulation can change the current set of instantiated locations. If this set gets empty, an EmptyInstantiatedLocationsException is thrown, and the simulation ends. This can, for instance, be an indicator that a tested system has shown a failure.

4.3.1 Getting the Status

To get a textual representation of the current set of instantiated locations the getStatus() method can be used:

    System.out.println(sim.getStatus());

Continuing the example of the last chapter, the initial output would be:

    1 instantiated location(s):
    Location 1: orderSucceeded(false),quoteIssued(FOO@product,0,}
Thus, the STS is in one current instantiated location, which corresponds to location 1 and the initial values of the two location variables orderSucceeded (with value false), and quoteIssued (with field values [FOO,0,""]). The set of current instantiated location objects can also be retrieved via the getInstantiedLocations method.

4.3.2 Getting the Current Switches

To generate a simulation step, i.e. processing a message, it can be crucial to know which switches are currently possible. By possible we mean that there is a current instantiated location from which the switch departs. Being possible does not necessarily mean that the switch can really fire – that depends on the satisfiability of the switch restriction. In the next section we will see how to find solutions for switch restrictions.

To retrieve from the simulator the set of all currently possible input switches, together with the instantiated locations from which they depart, the getCurrentInputSwitches method can be used. For instance:

```java
HashSet<InstantiatedSwitch> cis = sim.getCurrentInputSwitches();
```

An InstantiatedSwitch simply encapsulates a switch together with an instantiated location. In the running example, cis will be a singleton, only consisting of the one switch aSwitch we have defined, together with the initial instantiated location.

What is outside the scope of the simulator is the algorithm deciding the switch which should be simulated next. This can be done randomly, based on some heuristics, or whatsoever. For the example, we just pick the first (and only) switch available:

```java
InstantiatedSwitch is = (InstantiatedSwitch) cis.iterator().next();
```

Note that we have is.getSwitch() == aSwitch. The same way the current instantiated output switches can be retrieved via the getCurrentOutputSwitches method.

4.3.3 Computing Solutions for Switches

If a switch can fire depends on its switch restriction. A switch restriction restricts the values of the location variables, and of the interaction variables representing the message parameters of the switch's message. These parameter values can come from elsewhere (for instance by observing a real system). They can also be generated by the simulator itself. Both cases are relevant when, for instance, testing a real system. There, the output parameter values are observed at the tested system, whereas the input parameter values are generated by the simulator.

To generate parameter values, the simulator has to find a solution for the switch restriction. To do so, it uses the TREEsolver. It offers the findSolution method, which gets as its parameter an instantiated switch. It returns an instantiated message, which is a message together with a valuation of its interaction variables. For instance:

```java
InstantiatedMessage im = sim.findSolution(is);
```
Also here, some exceptions may be thrown. For our example, printing out the im object would yield:

\[
\text{makePayment?(reference(42), address(STSimulator))}
\]

What happened here? The simulator had to find values for the parameters \text{reference} and \text{address}. These values were restricted by the switch restriction

\[
\text{[reference > quoteIssued.quantity + 41]}
\]

Since in the initial instantiated location (which is part of the instantiated switch we gave to the \text{findSolution} method) the value of \text{quoteIssued.quantity} is 0, the switch restriction can be substituted to \text{[reference > 0 + 41]}. This is now solved by the \text{treeSolver}, which returns the first solution found: \text{reference = 42}. The \text{address} parameter was not restricted in any way by the switch restriction. Thus, every string would be fine. In such a case the simulator chooses the predefined string \text{STSimulator}.

Note that there are many solutions to the switch restriction. The \text{reference} parameter might as well be 43, or any other value greater than 41. Nevertheless the \text{treeSolver} will always return the first solution found (if any), which is 42 in this case. This is not always satisfactory, sometimes a more sophisticated choice would do better for certain purposes (e.g. random choice). This is not supported in the current version of the simulator. We hope to extent it in future versions in this direction.

### 4.3.4 Processing Messages

Having constructed an instantiated message, it can be given to the simulator to process it. This is done via the \text{processInstantiatedMessage} method. For instance:

\[
\text{sim.processInstantiatedMessage(im)};
\]

Also here exceptions may be thrown. Of special interest is the:

\text{STSimulator.EmptyInstantiatedLocationsException}

It indicates, that the instantiated message could not be processed from any current instantiated location. The current set becomes empty, and the simulation to a halt. If, instead, the instantiated message can be processed from one or several current instantiated locations, the new set of current instantiated locations is computed. Note that for the example, if the \text{Quick Sequence Diagram Editor} has been initialized, it has displayed the simulated message:
CHAPTER 4. SIMULATING A SYMBOLIC TRANSITION SYSTEM

The new set of instantiated locations can again be displayed via the `getStatus()` method. For the example we get:

1 instantiated location(s):
Location 1: orderSucceeded(true),quoteIssued(FO0@product,0,

We see here that the value of the `orderSucceeded` location variable has changed its value from `false` to `true`. This is due to the update mapping `orderSucceeded = true;` of the switch we have processed.

### 4.3.5 Processing Quiescence

A special observation which can be simulated is *quiescence*. This has the effect, that out of the set of the current instantiated locations all those are removed which can fire an output message, or on unobservable message. More precisely, this means that all those instantiated locations are removed from which a switch departs carrying an output- or unobservable message. Furthermore, the switch must have a satisfiable switch restriction (i.e., there is at least one solution for it). Quiescence is simulated via the `processQuiescence` method.

As a parameter the method gets the port at which the quiescence appears. In the current state of the simulator this has no further meaning, just pass the one port modeled before.

For instance:

```
sim.processQuiescence(supplierPort);
```

For the example, the only current instantiated location will remain, since there is only a single input switch departing from it.

### 4.3.6 Getting the Coverage Status

The method `getCoverageStatus` gives a string summarizing for each location and each switch how many times it has been visited during the simulation. It also tells the percentage of achieved location- and switch coverage. For the example it yields:

**Coverage Status**

--------

Location 1: 2 visit(s).
100.0 % location coverage.
Switch makePayment? [reference > quoteIssued.quantity + 41]
  orderSucceeded = true;: 1 visit(s).
100.0 % switch coverage.

To retrieve just the coverage percentage as a double, the two methods getLocationCoverage
and getSwitchCoverage can be used.

4.4 Cleaning Up

When the simulation shall be ended, the socket connections should be closed. The TREE SOLVER
additionaly expects the string bye. on its stream. For instance:

new PrintWriter(solverSocket.getOutputStream(), true).println("bye.");
solverSocket.close();
sdeSocket.close();
Chapter 5

Implementing On-The-Fly Testing Algorithms

To be written.
BIBLIOGRAPHY

Bibliography


Appendix A
The Dumont Grammar in BNF

SwitchRestriction ::= BooleanExpression <EOF>
UpdateMapping ::= ( VarAssignment )* <EOF>
VarAssignment ::= Id "=" TermExpression ";"
BooleanExpression ::= LogicalOrExpression ( "&&" LogicalOrExpression )*| TermExpression TermEqualityExpression
| "(" BooleanExpression ")"
LogicalOrExpression ::= LogicalNotExpression ( "||" LogicalNotExpression )*| TermExpression TermEqualityExpression
| "=" TermExpression
| "=!" TermExpression
| "<=" TermExpression
| "<" TermExpression
| ">=" TermExpression
| ">" TermExpression
TermEqualityExpression ::= "==" TermExpression
| "!=" TermExpression
| "<=" TermExpression
| "<" TermExpression
| ">=" TermExpression
| ">" TermExpression
TermExpression ::= TermAddExpression
TermAddExpression ::= TermSubtractExpression ( "+" TermSubtractExpression )*| TermExpression TermEqualityExpression
| "=" TermExpression
| "!=" TermExpression
| "<=" TermExpression
| "<" TermExpression
| ">=" TermExpression
| ">" TermExpression
TermSubtractExpression ::= TermMultExpression ( "-" TermMultExpression )*| TermExpression TermEqualityExpression
| "=" TermExpression
| "!=" TermExpression
| "<=" TermExpression
| "<" TermExpression
| ">=" TermExpression
| ">" TermExpression
TermMultExpression ::= TermDivExpression ( "*" TermDivExpression )*| TermExpression TermEqualityExpression
| "=" TermExpression
| "!=" TermExpression
| "<=" TermExpression
| "<" TermExpression
| ">=" TermExpression
| ">" TermExpression
TermDivExpression ::= TermModExpression ( "/" TermModExpression )*| TermExpression TermEqualityExpression
| "=" TermExpression
| "!=" TermExpression
| "<=" TermExpression
| "<" TermExpression
| ">=" TermExpression
| ">" TermExpression
TermModExpression ::= TermUnaryExpression ( "%" TermUnaryExpression )*| TermExpression TermEqualityExpression
| "=" TermExpression
| "!=" TermExpression
| "<=" TermExpression
| "<" TermExpression
| ">=" TermExpression
| ">" TermExpression
TermUnaryExpression ::= "++" TermUnaryExpression
| "--" TermUnaryExpression
| "(" TermExpression ")"
| PrimaryTermExpression
PrimaryTermExpression ::= Literal
| Id
| Name ::= <IDENTIFIER> ( "." <IDENTIFIER> )*
| Id ::= Name
Literal ::= ( <INTEGER_LITERAL> )
| ( <BOOLEAN_LITERAL> )
APPENDIX A. THE DUMONT GRAMMAR IN BNF

     | ( <STRING_LITERAL> )
     | ( <ENUMERATION_LITERAL> )
Appendix B

GNU GENERAL PUBLIC LICENSE
Version 3, 29 June 2007


Everyone is permitted to copy and distribute verbatim copies of this
license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of
works.

The licenses for most software and other practical works are designed to take away your
freedom to share and change the works. By contrast, the GNU General Public License is
intended to guarantee your freedom to share and change all versions of a program--to make
sure it remains free software for all its users. We, the Free Software Foundation, use the
GNU General Public License for most of our software; it applies also to any other work
released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General
Public Licenses are designed to make sure that you have the freedom to distribute copies
of free software (and charge for them if you wish), that you receive source code or can get
it if you want it, that you can change the software or use pieces of it in new free programs,
and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking
you to surrender the rights. Therefore, you have certain responsibilities if you distribute
copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you
must pass on to the recipients the same freedoms that you received. You must make sure
that they, too, receive or can get the source code. And you must show them these terms
so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert
copyright on the software, and (2) offer you this License giving you legal permission to
copy, distribute and/or modify it.

For the developers’ and authors’ protection, the GPL clearly explains that there is no warranty for this free software. For both users’ and authors’ sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users’ freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

“This License” refers to version 3 of the GNU General Public License.

“Copyright” also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

“The Program” refers to any copyrightable work licensed under this License. Each licensee is addressed as “you”. “Licensees” and “recipients” may be individuals or organizations.

To “modify” a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a “modified version” of the earlier work or a work “based on” the earlier work.

A “covered work” means either the unmodified Program or a work based on the Program.

To “propagate” a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.
To “convey” a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays “Appropriate Legal Notices” to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.


The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work’s System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.
2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.


No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work’s users, your or third parties’ legal rights to forbid circumvention of technological measures.


You may convey verbatim copies of the Program’s source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.
APPENDIX B. GNU GENERAL PUBLIC LICENSE
VERSION 3, 29 JUNE 2007

5. Conveying Modified Source Versions.
   You may convey a work based on the Program, or the modifications to produce it
from the Program, in the form of source code under the terms of section 4, provided
that you also meet all of these conditions:

   (a) The work must carry prominent notices stating that you modified it, and giving
       a relevant date.

   (b) The work must carry prominent notices stating that it is released under this
       License and any conditions added under section 7. This requirement modifies
       the requirement in section 4 to “keep intact all notices”.

   (c) You must license the entire work, as a whole, under this License to anyone who
       comes into possession of a copy. This License will therefore apply, along with
       any applicable section 7 additional terms, to the whole of the work, and all its
       parts, regardless of how they are packaged. This License gives no permission to
       license the work in any other way, but it does not invalidate such permission if
       you have separately received it.

   (d) If the work has interactive user interfaces, each must display Appropriate Legal
       Notices; however, if the Program has interactive interfaces that do not display
       Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which
are not by their nature extensions of the covered work, and which are not combined
with it such as to form a larger program, in or on a volume of a storage or distribution
medium, is called an “aggregate” if the compilation and its resulting copyright are
not used to limit the access or legal rights of the compilation’s users beyond what
the individual works permit. Inclusion of a covered work in an aggregate does not
cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.
   You may convey a covered work in object code form under the terms of sections 4
and 5, provided that you also convey the machine-readable Corresponding Source
under the terms of this License, in one of these ways:

   (a) Convey the object code in, or embodied in, a physical product (including a
       physical distribution medium), accompanied by the Corresponding Source fixed
       on a durable physical medium customarily used for software interchange.

   (b) Convey the object code in, or embodied in, a physical product (including a
       physical distribution medium), accompanied by a written offer, valid for at least
       three years and valid for as long as you offer spare parts or customer support
       for that product model, to give anyone who possesses the object code either
       (1) a copy of the Corresponding Source for all the software in the product that
           is covered by this License, on a durable physical medium customarily used for
software interchange, for a price no more than your reasonable cost of physically
performing this conveying of source, or (2) access to copy the Corresponding
Source from a network server at no charge.

(c) Convey individual copies of the object code with a copy of the written offer to
provide the Corresponding Source. This alternative is allowed only occasionally
and noncommercially, and only if you received the object code with such an
offer, in accord with subsection 6b.

(d) Convey the object code by offering access from a designated place (gratis or
for a charge), and offer equivalent access to the Corresponding Source in the
same way through the same place at no further charge. You need not require
recipients to copy the Corresponding Source along with the object code. If the
place to copy the object code is a network server, the Corresponding Source
may be on a different server (operated by you or a third party) that supports
equivalent copying facilities, provided you maintain clear directions next to the
object code saying where to find the Corresponding Source. Regardless of what
server hosts the Corresponding Source, you remain obligated to ensure that it
is available for as long as needed to satisfy these requirements.

(e) Convey the object code using peer-to-peer transmission, provided you inform
other peers where the object code and Corresponding Source of the work are
being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the
Corresponding Source as a System Library, need not be included in conveying the
object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible
personal property which is normally used for personal, family, or household purposes,
or (2) anything designed or sold for incorporation into a dwelling. In determining
whether a product is a consumer product, doubtful cases shall be resolved in favor
of coverage. For a particular product received by a particular user, “normally used”
refers to a typical or common use of that class of product, regardless of the status of
the particular user or of the way in which the particular user actually uses, or expects
or is expected to use, the product. A product is a consumer product regardless of
whether the product has substantial commercial, industrial or non-consumer uses,
unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authoriz-
er keys, or other information required to install and execute modified versions
of a covered work in that User Product from a modified version of its Corresponding
Source. The information must suffice to ensure that the continued functioning of
the modified object code is in no case prevented or interfered with solely because
modification has been made.

If you convey an object code work under this section in, or with, or specifically for
use in, a User Product, and the conveying occurs as part of a transaction in which
the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

“Additional permissions” are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

(a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

(b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

(c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
(d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

(e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

(f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered “further restrictions” within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have
been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.


You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party’s predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor’s “contributor version”.

A contributor’s “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of
this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor’s essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a “patent license” is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To “grant” such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient’s use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE
THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS
How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

<one line to give the program’s name and a brief idea of what it does.>
Copyright (C) <year><name of author>
This program is free software: you can redistribute it and/or modify it un-
der the terms of the GNU General Public License as published by the Free
Software Foundation, either version 3 of the License, or (at your option)
any later version.

This program is distributed in the hope that it will be useful, but WITH-
OUT ANY WARRANTY; without even the implied warranty of MER-
CHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See
the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along
with this program. If not, see http://www.gnu.org/licenses/.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when
it starts in an interactive mode:

<program>Copyright (C) <year><name of author>

This program comes with ABSOLUTELY NO WARRANTY; for details
type show w. This is free software, and you are welcome to redistribute it
under certain conditions; type show c for details.

The hypothetical commands show w and show c should show the appropriate parts of
the General Public License. Of course, your program’s commands might be different;
for a GUI interface, you would use an “about box”.

You should also get your employer (if you work as a programmer) or school, if any, to
sign a “copyright disclaimer” for the program, if necessary. For more information on
this, and how to apply and follow the GNU GPL, see http://www.gnu.org/licenses/.

The GNU General Public License does not permit incorporating your program into
proprietary programs. If your program is a subroutine library, you may consider it
more useful to permit linking proprietary applications with the library. If this is what
you want to do, use the GNU Lesser General Public License instead of this License.