The JRELIABILITY Tutorials

Michael Glaß

October/29/2008

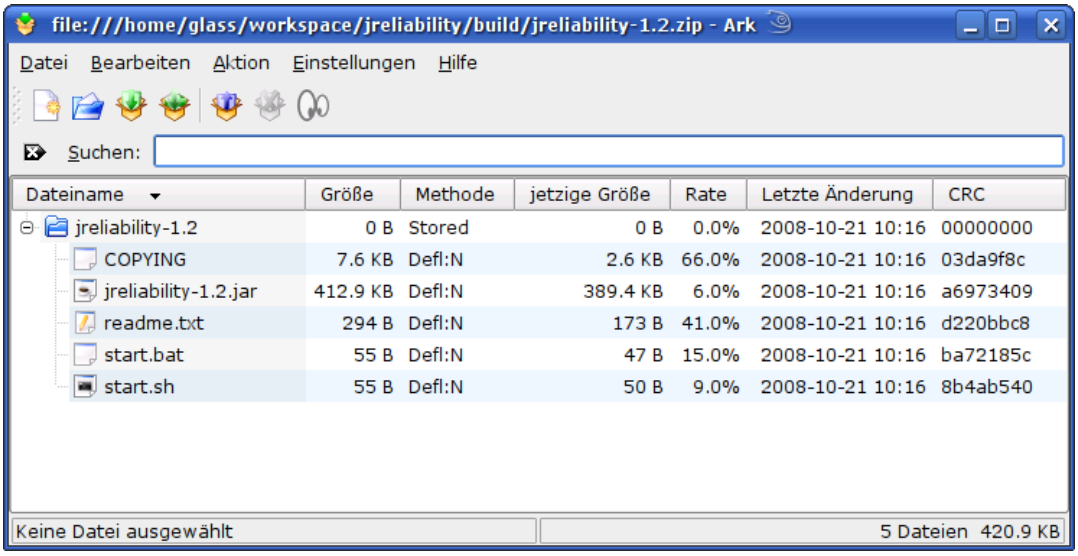
This tutorial consists of five sections: The first section is an installation guide for JRELIABILITY. The second section outlines the structure of JRELIABILITY covering modeling and evaluation. The third part gives a very brief introduction of *Binary Decision Diagrams* (BDDs) that are used to encode the *Boolean functions* that are the main data structure for the modeling. Concluding, two examples show how to model and evaluate a *Boiler* that controls a hot water tank using BDDs as well as a *Triple Modular Redundancy* (TMR) structure where modeling using linear constraints is introduced.

© JReliability.org

1 Installing JRELIABILITY

To install JRELIABILITY, the first step is to ensure that at least Java 5 Runtime Environment (JRE) is installed on your system. (You can use `java -version` on your command line (console) to check the version of your JRE.) If this is not the case, visit `http://java.sun.com`, download, and install the latest JRE for your operating system.

Next, you have to download the JRELIABILITY jar from `http://www.jreliability.org`. Download the latest release (currently, the zip-file **jreliability-1.2.zip**). Unzip all files.



To use JRELIABILITY as a library for your project, copy the included **jreliability-1.2.jar** file to your standard lib folder of your project or include the path to the file in your *classpath*.

A small test example can be executed for a quick start: On Windows systems, you can start the test example of JRELIABILITY with the **start.bat** file (or double-click the **jreliability-1.2.jar** file). On UNIX systems, use **start.sh**. Alternatively, you can use the command:

```
java -jar jreliability-1.2.jar
```

To install the JRELIABILITY tutorials, download the latest release (currently, the zip-file **jreliability-tutorials-1.2.zip**). Unzip all files.

To launch the *Boiler* example use:

```
java -jar jreliability-tutorials-1.2.jar
```

or

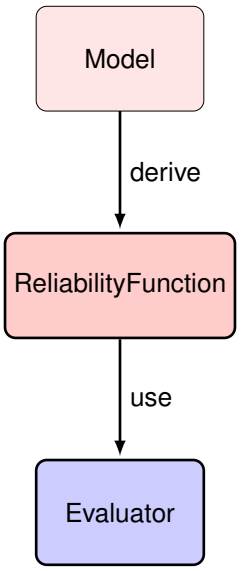
```
java -cp jreliability-tutorials-1.2.jar org.jreliability.tutorial.boiler.BoilerTester
```

To launch the *TMR* example use:

```
java -cp jreliability-tutorials-1.2.jar org.jreliability.tutorial.tmr.TMRTester
```

2 Understanding JRELIABILITY

The basic idea of the JRELIABILITY library is that there is some kind of *Model* of the system that describes its behavior in case of occurrences like *failures, defects* or also the *repairing* of system components. Based on this model, the *ReliabilityFunction* of the overall system can be generated. In the last step, a set of *evaluators* derives reliability-related measures based on the either the *Model* and the *ReliabilityFunction* or, if possible, on the *ReliabilityFunction* only.

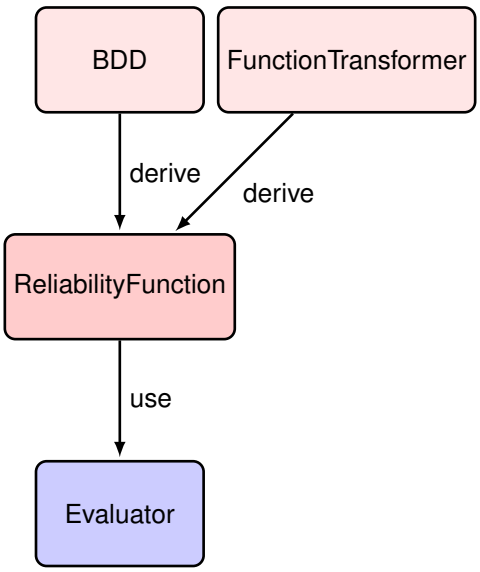


2.1 Modeling

The behavior of a system is typically represented by *fault-trees*, *Reliability Block Diagrams* (RBDs), *automatons*, *Markov-chains* or *Boolean functions*. All these techniques have in common that there exists, either implicitly or explicitly, a data-structure that allows to determine whether the system is working properly in its current state or whether it failed, i.e., it is not working properly. In JRELIABILITY, a Boolean function is used to represent this so called *structure function* of the system, encoded in *Binary Decision Diagrams* (BDDs).

To be able to derive the *ReliabilityFunction* of the overall system, the *ReliabilityFunction* of each element in the system has to be given. For this purpose, a data-structure called *FunctionTransformer* is used in JRELIABILITY that is a simple mapping of each element to its corresponding *ReliabilityFunction*. JRELIABILITY comes with a large set of predefined common *ReliabilityFunctions* based on *exponential distributions*, *Weibull distributions*, and many more.

With this knowledge, the JRELIABILITY structure can be refined as follows:



Note that the *BDD* encoding the Boolean function in connection with the *FunctionTransformer* is just one way to derive the overall *ReliabilityFunction*. Other ways can be implemented by the user and are also future work for the JRELIABILITY library.

For more information on how to use *BDDs* see the next section, for examples on how to setup a *BDD* for a given system see the last two sections of this tutorial.

2.2 Evaluation

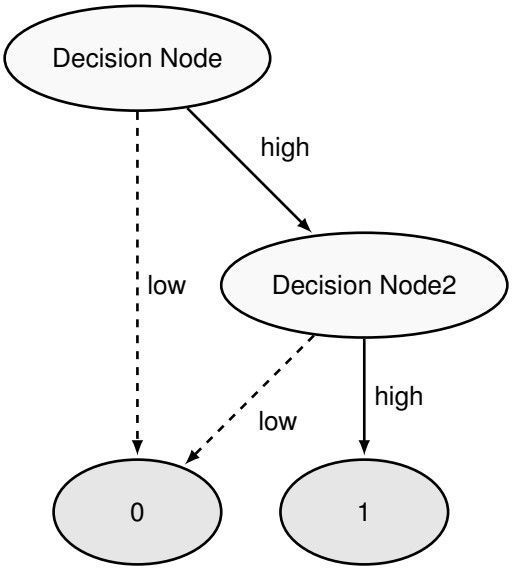
Once the *ReliabilityFunction* of the overall system is determined, the reliability-related measures like *Mean-Time-To-Failure* (MTTF), *Mission-Time* (MT), *failure-rates* etc. are derived using so called *Evaluators*. An *Evaluator* commonly takes a *ReliabilityFunction* as an input and determines the desired measure either analytically or via sampling or simulation. Some *Evaluators* also need access to the *ReliabilityFunction* of each element of the system and to the Boolean structure function of the overall system that is encoded in the *BDD*. For this purpose, a special *ReliabilityFunction* called *BDDReliabilityFunction* is provided that also gives access to both, the *ReliabilityFunction* of each element and the *BDD*.

For more information on how to use the evaluators see the examples in the last two sections of this tutorial.

3 BDDs - A very brief Introduction

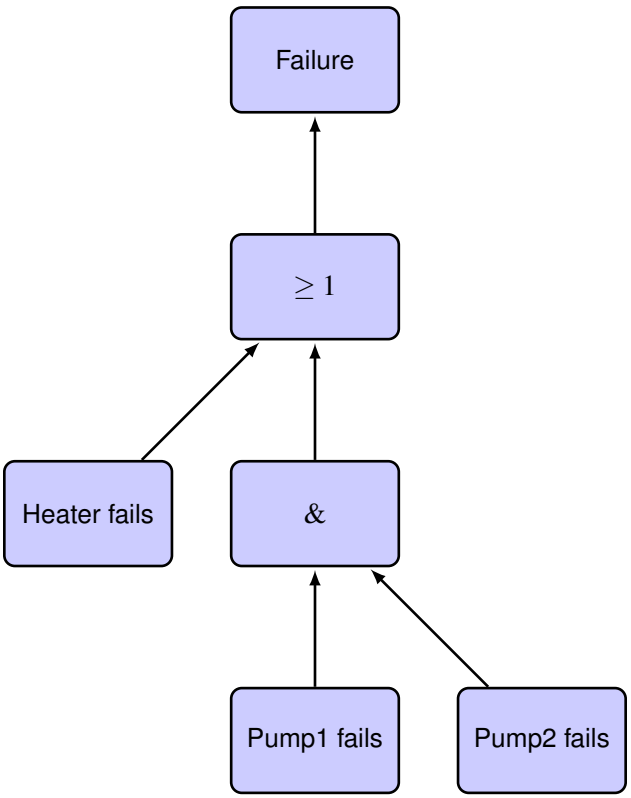
The basic model for the systems behavior that is currently feature in JRELIABILITY are *Boolean function*. The basic data-structure to encode these function efficiently in JRELIABILITY are *Binary Decision Diagrams* or short *BDDs*. An important feature of JRELIABILITY is that it offers a special generic interface to several *BDD* libraries that allows to directly use the *Java Objects* that model the real system *components* as variables of the *BDD*.

BDDs allow a canonical representation of *Boolean functions*. A *BDD* itself is a rooted, directed, acyclic graph that consists of *decision nodes* that correspond to variables and two *terminal nodes* 0 and 1 that correspond to the return value of the *Boolean function*. Since each variable is a *binary* variable that can only take the values 0 or 1, each decision node has two outgoing edges *low* and *high* that correspond to the variable being 0 or 1. Each variable assignment that reaches the 1 terminal node corresponds to a proper working system while each assignment that reaches the 0 terminal node corresponds to a system that failed, respectively.



In this reliability library, each variable commonly corresponds to a *component*, i.e., its representation as a Java *Object*, of the system with a 0 value corresponding to a fault of the component while a 1 value corresponds to a proper working component, respectively. The *Objects* of each of the components are directly connected using logical operators like, e.g., *AND* and *OR* to encode the overall *structure function* of the system in a single *BDD*.

A simple example of a heater and two pumps is given as a *fault-tree*:



The statement that can be derived from this *fault-tree* is

$$0 = \text{Heater fails} \vee (\text{Pump1 fails} \wedge \text{Pump2 fails}) \tag{1}$$

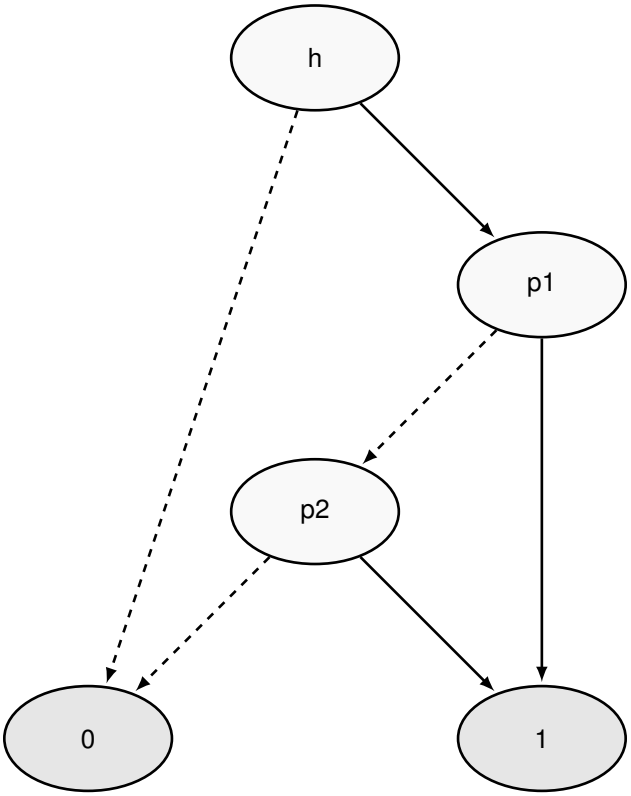
Since it is more intuitive to model the *BDD* in a way that 1 is the desired result of the *Boolean function*, the *fault-tree* can be simply inverted, leading to

$$1 = \text{Heater works} \wedge (\text{Pump1 works} \vee \text{Pump2 works}) \tag{2}$$

Using the variables h for the Heater, p_1 for pump1 and p_2 for pump2, the desired *Boolean function*, i.e., the *structure function* φ is:

$$\varphi(x, p_1.p_2) = h \wedge (p_1 \vee p_2) \tag{3}$$

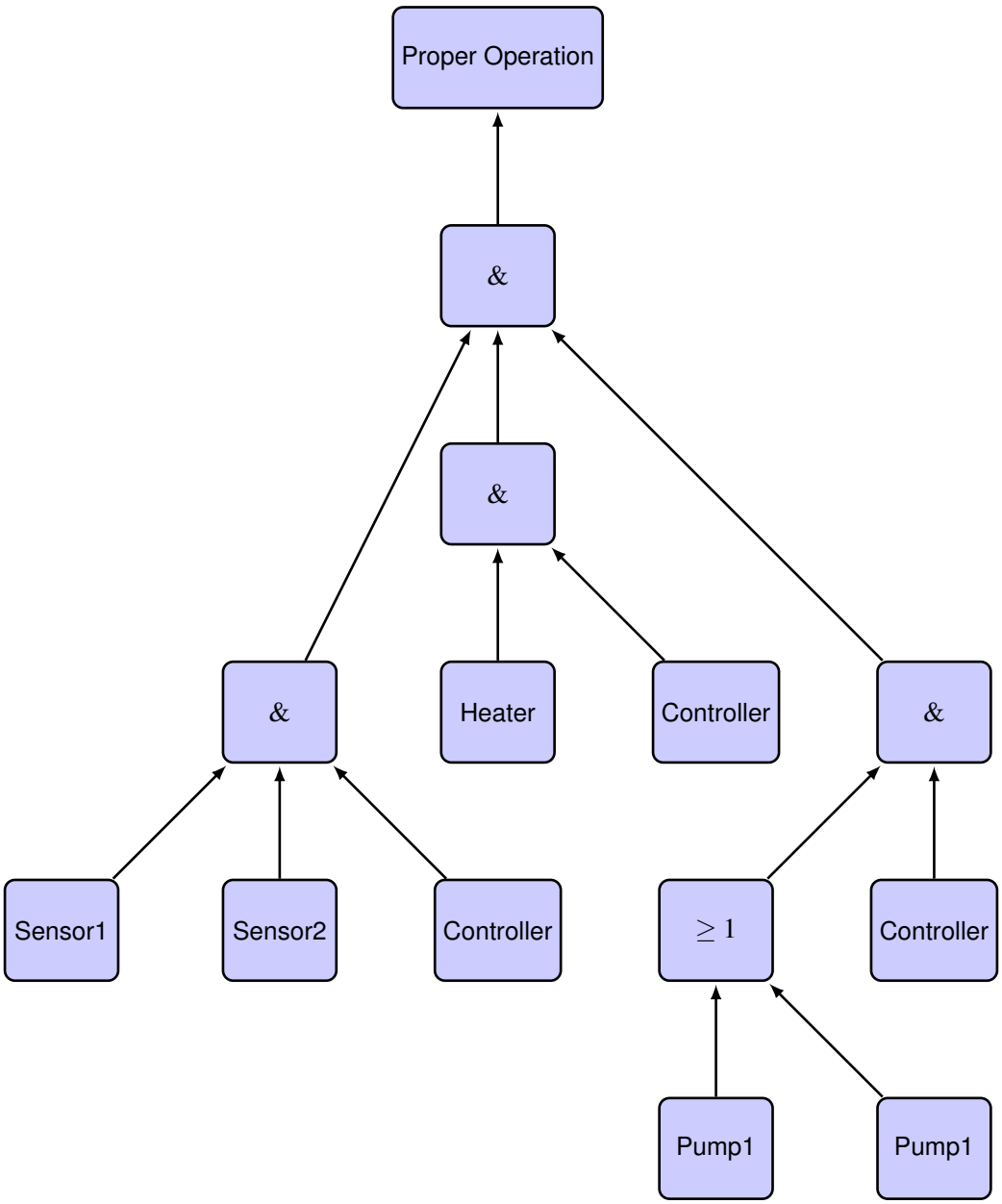
The corresponding *BDD* looks as follows:


h
p1
p2
0
1

4 Example 1: A hot water Boiler

This example introduces a simple system of a hot water *Boiler*. It shows how the system can be modeled, represented as a *Boolean function* and encoded in a *BDD*, converted into a *ReliabilityFunction*, and evaluated afterwards.

The *Boiler* models a boiler that is responsible for keeping the water in a tank at the desired temperature and pumping it to a destination if needed. The *Boiler* consists of two *Sensors* that both measure and compare the measured water temperature to be sure the correct temperature is known, a *Controller* that activates and deactivates a *Heater* to control the water temperature as well as it activates and deactivates one of two available *Pumps* to pump the water to its destination if needed. The following *success-tree* visualizes the system:



The non-minimized Boolean function that describes whether this system works correctly (evaluates to 1) or fails (evaluates to 0) is as follows:

$$((\text{Sensor1} \wedge \text{Sensor2}) \wedge \text{Controller}) \wedge (\text{Controller} \wedge \text{Heater}) \wedge (\text{Controller} \wedge (\text{Pump1} \vee \text{Pump2})) \quad (4)$$

Given this description of the system, the following tutorial will show how this *Boiler* can be modeled and evaluated using JRELIABILITY. First, an abstract *BoilerComponent* is defined that will serve as the base class for all components of the *Boiler*.

BoilerComponent.java

```
public abstract class BoilerComponent {
```

```
  protected final String name;

  public BoilerComponent(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }

  public String toString() {
    return name;
  }

}
```

The *BoilerComponent* includes all the things that the system components have in common, in this case, a name is assigned to all of them. Next, the classes *Sensor*, *Controller*, *Heater*, and *Pump* are defined.

The system components

```
public class Sensor extends BoilerComponent {

  public Sensor(String name) {
    super(name);
  }

}

public class Controller extends BoilerComponent {

  public Controller(String name) {
    super(name);
  }

}

public class Heater extends BoilerComponent {

  public Heater(String name) {
    super(name);
  }

}

public class Pump extends BoilerComponent {

  public Pump(String name) {
    super(name);
  }

}
```

For the sake of simplicity, no further functionality is added to these classes. When modeling the real

system, one could implement further functionality or add more data to each type of component. In this tutorial, the different classes are used to show that the encoding of the *system structure* in the *BDD* can be performed using different and user defined classes that model the system components.

The next step is to model the overall system, i.e., the *Boiler*.

Boiler.java

```
public class Boiler {

  protected Sensor sensor1 = new Sensor("Sensor1");
  protected Sensor sensor2 = new Sensor("Sensor2");
  protected Controller controller = new Controller("Controller");
  protected Heater heater = new Heater("Heater");
  protected Pump pump1 = new Pump("Pump1");
  protected Pump pump2 = new Pump("Pump2");

  protected List<BoilerComponent> components = new ArrayList<BoilerComponent>();
  protected BoilerTransformer transformer;

  public Boiler() {
    super();
    initialize();
    transformer = new BoilerTransformer(this);
  }

  //
  // Initializes the list of components of the Boiler.
  //
  private void initialize() {
    components.add(sensor1);
    components.add(sensor2);
    components.add(controller);
    components.add(heater);
    components.add(pump1);
    components.add(pump2);
  }

  //
  // Returns a model of the Boiler as a BDD.
  //
  public BDD<BoilerComponent> get() {
    BDDProviderFactory bddProviderFactory = new JBDDProviderFactory();
    BDDProvider<BoilerComponent> bddProvider = bddProviderFactory
        .getProvider();

    BDD<BoilerComponent> sensor1BDD = bddProvider.get(sensor1);
    BDD<BoilerComponent> sensor2BDD = bddProvider.get(sensor2);
    BDD<BoilerComponent> controllerBDD = bddProvider.get(controller);
    BDD<BoilerComponent> heaterBDD = bddProvider.get(heater);
    BDD<BoilerComponent> pump1BDD = bddProvider.get(pump1);
    BDD<BoilerComponent> pump2BDD = bddProvider.get(pump2);

    BDD<BoilerComponent> sensorSubSystem = sensor1BDD.and(sensor2BDD);
    BDD<BoilerComponent> senControlSubSystem = sensorSubSystem
        .and(controllerBDD);
```

```
    BDD<BoilerComponent> heatingSubSystem = heaterBDD.and(controllerBDD);

    BDD<BoilerComponent> pumpSubSystem = pump1BDD.or(pump2BDD);
    BDD<BoilerComponent> pumpControlSubSystem = pumpSubSystem
        .and(controllerBDD);

    BDD<BoilerComponent> boilerBDD = senControlSubSystem
        .and(heatingSubSystem);
    // Important: With-operators consume (destroy!) the BDD that is the
    // argument i.e. the pumpControlSubSystem is destroyed after this
    // operation, while
    // BDD<BoilerComponent> boilerBDD = boilerBDD.and(pumpControlSubSystem);
    // would not destroy anything
    boilerBDD.andWith(pumpControlSubSystem);

    return boilerBDD;
  }

  //
  // Returns the components of the Boiler.
  //
  public List<BoilerComponent> getComponents() {
    return components;
  }

  //
  // Returns the BoilerTransformer.
  //
  public BoilerTransformer getTransformer() {
    return transformer;
  }

}
```

The *Boiler* contains all the components of the system and calculates the *BDD* that represents the *system structure* φ. Note that the function *get()* creates the *BDD* using simple *AND* and *OR*-operators.

As the last modeling step, the *Transformer* that assigns each system component a *ReliabilityFunction* has to be provided.

BoilerTransformer.java

```
public class BoilerTransformer implements FunctionTransformer<BoilerComponent> {

  Map<BoilerComponent, ReliabilityFunction> reliabilityFunctions = new HashMap<
      BoilerComponent, ReliabilityFunction>();

  public BoilerTransformer(Boiler boiler) {
    initialize(boiler);
  }

  private void initialize(Boiler boiler) {
    for (BoilerComponent component : boiler.getComponents()) {
      ReliabilityFunction reliabilityFunction = new WeibullReliabilityFunction(
          0.5, 2);
      reliabilityFunctions.put(component, reliabilityFunction);
```

```
    }

  }

  public ReliabilityFunction transform(BoilerComponent element) {
    ReliabilityFunction reliabilityFunction = reliabilityFunctions
        .get(element);
    return reliabilityFunction;
  }

}
```

In this case, the transformer simply assigns each component a *WeibullReliabilityFunction* with scale set to 0.5 and a scale of 2.

Since the *Boiler* is modeled using a *BDD* with a *FunctionTransformer*, the *BoilerTransformer*, being available as well, the standard constructor of the class *BDDReliabilityFunction* can be used that creates the desired *ReliabilityFunction* given a *BDD* and a *FunctionTransformer*.

As a last step, a *BoilerTester* is used to show how to use *Evaluators* and how to invoke the JRELIABILITY extended GUI.

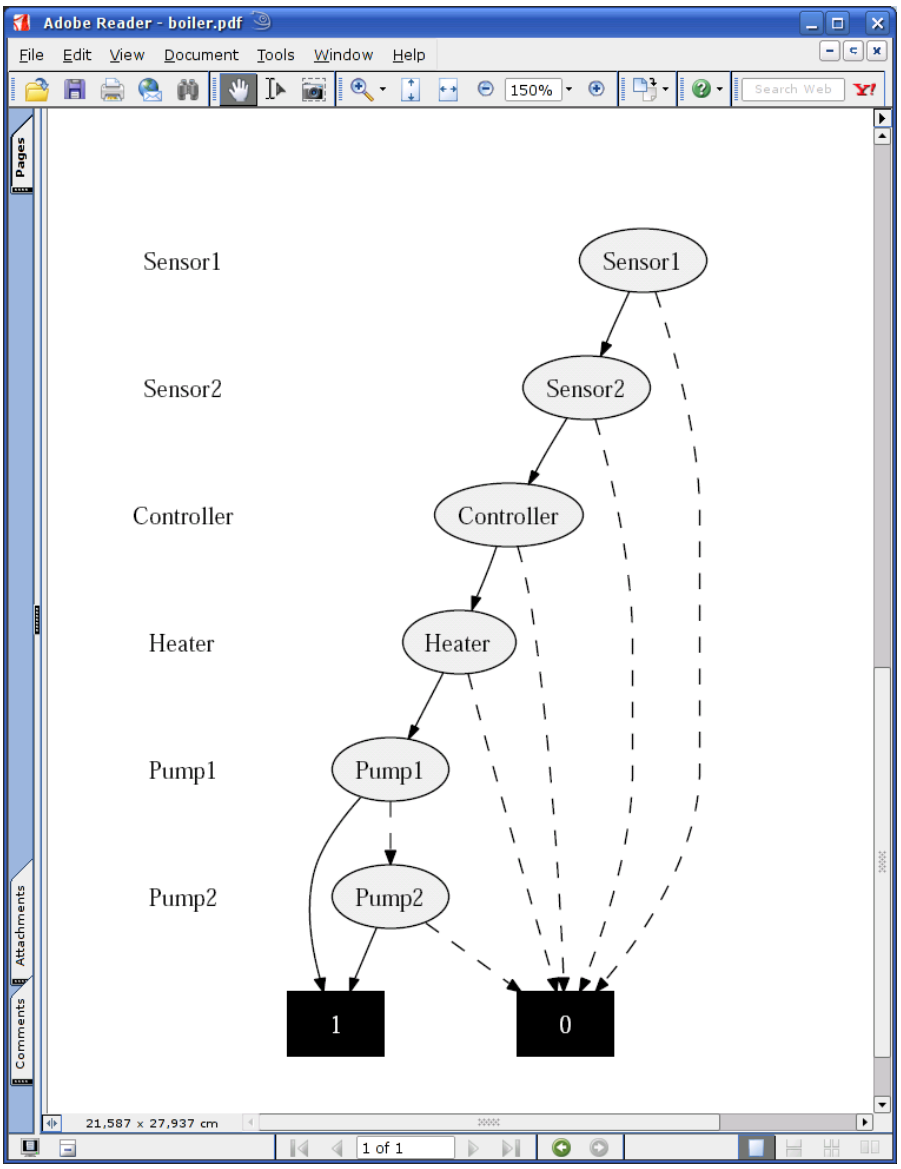
BoilerTester.java

```
public class BoilerTester {

  //
  // Main.
  //
  public static void main(String[] args) {

    Boiler boiler = new Boiler();
    BDD<BoilerComponent> boilerBDD = boiler.get();

    // Visualizing the BDD
    String dot = BDDs.toDot(boilerBDD);
    System.out.println(dot);
    System.out.println("***");

    BoilerTransformer transformer = boiler.getTransformer();

    BDDReliabilityFunction<BoilerComponent> reliabilityFunction = new
        BDDReliabilityFunction<BoilerComponent>(
        boilerBDD, transformer);

    // Using Evaluators
    // Calculate Mean-Time-To-Failure (the first moment of the density
    // function)
    MomentEvaluator moment = new MomentEvaluator(1);
    Double mttf = moment.evaluate(reliabilityFunction);
    System.out.println("Mean-Time-To-Failure: " + mttf);
    System.out.println("***");

    // Using the GUI
    Map<String, ReliabilityFunction> reliabilityFunctions = new HashMap<String,
```

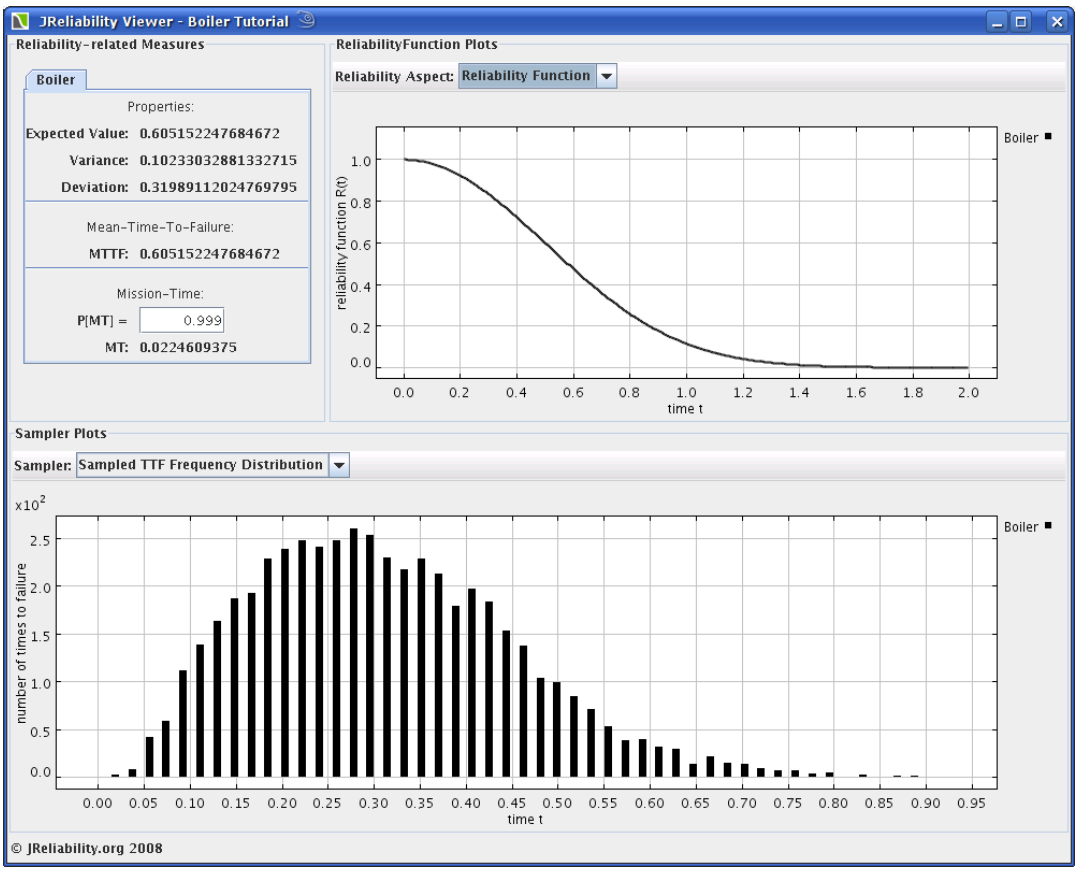
```
        ReliabilityFunction>();
    reliabilityFunctions.put("Boiler", reliabilityFunction);

    ReliabilityViewer.view("JReliability Viewer - Boiler Tutorial",
        reliabilityFunctions, true);

  }

}
```

The *BoilerTester* first prints the generated *BDD* to standard out using the DOT input format. Afterwards, it creates the desired *ReliabilityFunction*. Given the *ReliabilityFunction*, using an *Evaluator* is straightforward: In this case, the *MomentEvaluator* calculates the first *moment* of the *density function* that can be automatically derived from the *ReliabilityFunction*. This first moment equals the well-known *Mean-Time-To-Failure* (MTTF) of the system. Afterwards, the GUI can be started with the *ReliabilityFunction* of the hot water system that has been assigned a name before. This way, many different systems or the same system with different parameters can be added to the same GUI and shown together in the plot.

The output of the *BoilerTester* should be the DOT input string and a MTTF of 0.605152247684672. Plotted using DOT and converted to *.pdf*, the *BDD* looks as follows:



The JRELIABILITY viewer should show the following *reliability function*:



This extended GUI shows common reliability-related measures, different aspects of the system that can be derived analytically as well as the histogram of sampled datas that are derived by a *Evaluator* that determines times-to-failures by a sampling approach. The extended GUI is launched by setting the *showSampleHistograms* when using *ReliabilityViewer.view()* to *true*.

5 Example 2: Triple-Modular-Redundancy

This example covers a well-known construct that provides fault-detection as well as fault-tolerance, the *Triple-Modular-Redundancy* or short TMR. The *TMR* is a 2-out-of-3 voter that compares the results provided by three (often equal) components performing the same functionality and propagates the result of at least two of the three components that have identical results. Otherwise, a fault in more than one component is assumed and the correct result cannot be determined anymore.

The 2-out-of-3 voting is an interesting case where the modeling of the system is much easier using a *linear constraint*. Without the linear constraint, the *structure function* would looks like this:

$$(\text{component1} \wedge \text{component2}) \vee (\text{component1} \wedge \text{component3}) \vee (\text{component2} \wedge \text{component3}) \quad (5)$$

On the other hand, formulated as a linear constraint, the function becomes quite compact:

$$\text{component1} + \text{component2} + \text{component3} \geq 2 \quad (6)$$

JRELIABILITY offers a special function to represent a linear constraint as a *BDD* in its *BDDs* class that serves as a toolbox. The linear constraint consists of *variables* and corresponding *coefficients*, e.g., $3x$, on the left-hand-side, a *comparator*, i.e., $\geq$, >, < $\leq$, and =, as well as the right-hand-side that is an integer value.

In the following, the implementation of the *TMR* is shown. Note that there is no special class defined for the components of the *TMR* but the class *String* can also be easily used for the modeling.

TMR.java

```
public class TMR {

  protected String component1 = new String("component1");
  protected String component2 = new String("component2");
  protected String component3 = new String("component3");
  protected FunctionTransformer<String> transformer;

  public TMR() {
    super();
    initialize();
  }

  //
  // Initializes the FunctionTransformer of the TMR.
  //
  private void initialize() {
    Map<String, ReliabilityFunction> reliabilityFunctions = new HashMap<String,
        ReliabilityFunction>();
    ReliabilityFunction function = new ExponentialReliabilityFunction(0.1);
    reliabilityFunctions.put(component1, function);
    reliabilityFunctions.put(component2, function);
    reliabilityFunctions.put(component3, function);
    transformer = new SimpleFunctionTransformer<String>(
        reliabilityFunctions);
  }
```

```
  //
  // Returns a model of the TMR as a BDD.
  //
  public BDD<String> get() {

    BDDProviderFactory bddProviderFactory = new JBDDProviderFactory();
    BDDProvider<String> bddProvider = bddProviderFactory.getProvider();

    BDD<String> component1BDD = bddProvider.get(component1);
    BDD<String> component2BDD = bddProvider.get(component2);
    BDD<String> component3BDD = bddProvider.get(component3);

    // To use the inbuilt constraint functionality, setup the left-hand-side
    // first

    List<Integer> coefficients = new ArrayList<Integer>();
    List<BDD<String>> variables = new ArrayList<BDD<String>>();

    coefficients.add(1);
    variables.add(component1BDD);

    coefficients.add(1);
    variables.add(component2BDD);

    coefficients.add(1);
    variables.add(component3BDD);

    BDD<String> tmr = BDDs.getBDD(coefficients, variables, ">=", 2);

    return tmr;
  }

  //
  // Returns the FunctionTransformer.
  //
  public FunctionTransformer<String> getTransformer() {
    return transformer;
  }

}
```

In this example, all components are assigned an *ExponentialReliabilityFunction* with a failure rate of 0.1.

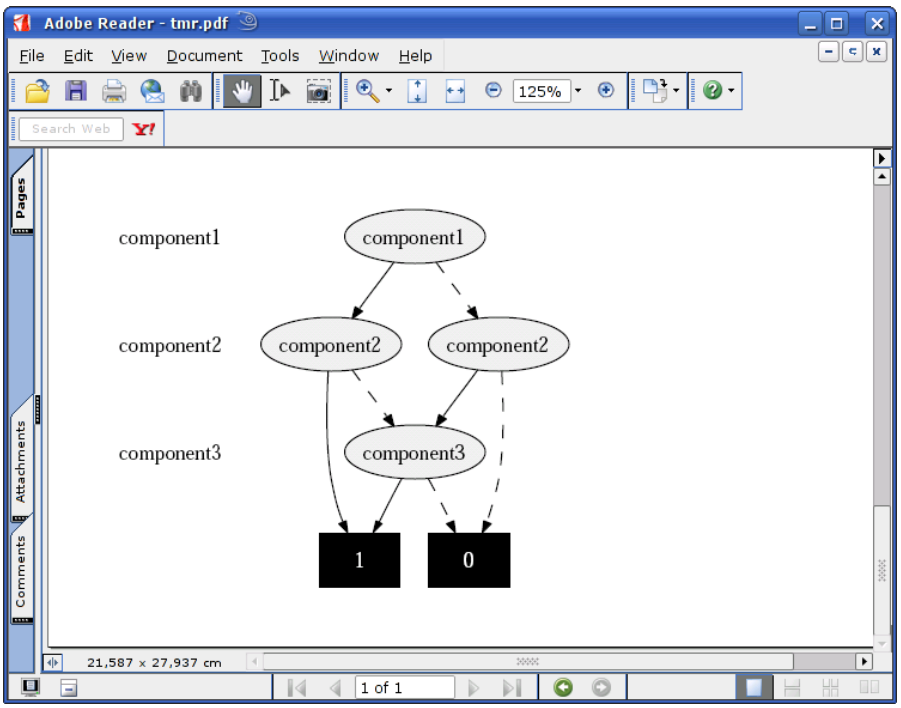
Given the definition of the *TMR*, a *TMRester* similar to the one used in the *Boiler* example is implemented. A difference is that a second *ExponentialReliabilityFunction* with a failure rate of 0.1 is created that corresponds to a system with only one instead of three components in the *TMR*.

TMRTester.java

```
public class TMRTester {

  //
  // Main.
  //
  public static void main(String[] args) {
```

```
    TMR tmr = new TMR();
    BDD<String> tmrBDD = tmr.get();

    // Visualizing the BDD
    String dot = BDDs.toDot(tmrBDD);
    System.out.println(dot);
    System.out.println("***");

    FunctionTransformer<String> transformer = tmr.getTransformer();

    BDDReliabilityFunction<String> reliabilityFunctionTMR = new
        BDDReliabilityFunction<String>(
        tmrBDD, transformer);

    // The single element solution equals a simple
    // ExponentiaRreliabilityFunction
    ReliabilityFunction reliabilityFunctionSingle = new
        ExponentialReliabilityFunction(
        0.1);

    // Calculate Mean-Time-To-Failures (the first moment of the density
    // function)
    MomentEvaluator moment = new MomentEvaluator(1);
    Double mttfTMR = moment.evaluate(reliabilityFunctionTMR);
    System.out.println("Mean-Time-To-Failure of TMR: " + mttfTMR);
    Double mttfSingle = moment.evaluate(reliabilityFunctionSingle);
    System.out.println("Mean-Time-To-Failure of single element: "
        + mttfSingle);
    System.out.println("***");

    // Using the GUI
    Map<String, ReliabilityFunction> reliabilityFunctions = new HashMap<String,
        ReliabilityFunction>();
    reliabilityFunctions.put("TMR", reliabilityFunctionTMR);
    reliabilityFunctions.put("Single Component", reliabilityFunctionSingle);

    ReliabilityViewer.view("JReliability Viewer - TMR Tutorial",
        reliabilityFunctions);

  }

}
```

The output of the *TMRTester* should be the DOT input string and a MTTF of 8.318560681776674 for the *TMR* and a MTTF of 9.929773583403188 for the single component solution. Plotted using DOT and converted to *.pdf*, the *BDD* of the *TMR* looks as follows:



The standard JRELIABILITY viewer (not launching the histogram for sampled data) should show the well-known comparison between *TMR* and the single component system: