The Unified Modeling Language (UML) allows the description of complex discrete devices in a clear understandable manner. After the mapping of such a high level model to a basically tested programmable hardware, the test of the special behavior arises. This paper introduces the object oriented modeling of discrete devices by UML, shows the model transfer between the CASE tool and the test tool by XML emphasizes the generation and parameterization of the test cases, and presents our practical results.

1. Introduction

Generally, discrete devices need a hardware basis, which may be first, a set of logic gates and flip-flops, second, a set of configurable logic blocks (CLB) or third, a complete microprocessor. The concrete behavior of the discrete device is specified by an appropriate program. In the first case the program defines the dedicated connection between the gates, so that the program becomes part of the hardware. In the second case the program is a bit stream for the programmable logic device (PLD), which specifies both the behavior of each CLB and the programmable connections between them. Finally, in the third case the program is a sequence of instructions. The execution of these instructions transforms the input pattern of the discrete device into their output pattern. The advantage of the second and the third architecture is their programmability, which allows a very fast time to market, and a change of the behavior on run time. A further advantage of these architectures is that the test of the basic hardware can be separated from the test of the concrete behavior of the discrete device.

The specification of the behavior needs a formal definition of a basic language, which may be in simple case Boolean functions or finite state machines. For very complex discrete devices higher languages like the Unified Modeling Language (UML) are more advantageous. In [1] was shown that the object oriented high level model can be automatically transformed into the previous mentioned programmable discrete devices. In this paper we focus on the test of the specific behavior of discrete devices.
2. Object Oriented Modeling

UML is a universal description language for all kinds of object-oriented software [2]. The static behavior of a device will be described by a class diagram (Fig. 1) and, for the dynamic behavior the sequence diagram (Fig. 2) will be used. A class diagram models the class structure and displays relationships such as containment, inheritance, associations and others. A rectangle represents a class in a class diagram. There are three parts in a rectangle of a class. In the top is located the name of the class, the middle part represents the attributes and the lower part the operations of the class.

The association relationship is the most common relationship in a class diagram. The association expresses the collaboration between classes. For example, the class Order is associated with the class Customer. In the case of a “part of whole” relation between two classes it is an aggregation or a composition relationship. Another common relationship in class diagrams is a generalization. A generalization relationship between two classes means that one class inherits the attributes and operations of another one. For example, the class Customer is the base class for the classes Corporate Customer and Personal Customer.

![Class Diagram](Fig. 1. Class Diagram)

![Sequence Diagram](Fig. 2. Sequence Diagram)
Interaction diagrams are used in order to model the behavior of several objects solving a task. In detail they demonstrate how the objects collaborate for the behavior. Sequence diagrams express the behavior of objects by describing the objects and messages between them. The time is scheduled top down. The example above (Fig. 2) shows an actor object that starts the behavior by sending a message to an object of class Corporate Customer. Messages pass several objects until the actor receives the final message.

3. CASE - Tools and XML – Export

A CASE (computer aided software engineering) tool is the Software to creating and process the UML-Models. In this paper we have used the CASE tool Rational Rose. To use the data from the models in another program, we export the models using the XML Language. XML stands for EXtensible Markup Language and was designed to describe data and their types [3]. XML tags are self-descriptive and not predefined. Own tags can be defined using a Document Type Definition (DTD) or an XML scheme. Therefore it is possible to define markups to describe UML model elements.

4. Generation of Test Templates

The test of the behavior can be done by test templates, which have to be parameterized with correct or a wrong values for attributes or a function parameters [4]. This paper presents an algorithm that generates test templates automatically. Each model element that is really used by one business process will be tested only once. The generated test templates cover the necessary modeled behavior.

At first we build an UML model that described the behavior to be tested. Then we export this model from the CASE tool to a file in XML format. Using a Reader-program an equivalent internal model is created.

The main concept of the algorithm is to test the behavior by testing of its business processes. Using this idea, only operations embedded in a business process will be tested. They shouldn’t be tested independently, but only in connection with a sequence diagram.

Classes and operations are the main model elements to be tested. A Boolean number is assigned to each model element. The values of attributes of classes and the parameters of the operations will be coded by Boolean values, too. Correct (wrong) values of attributes or parameters are coded with a one (zero) in the test template. If the value of a model element is not important for a test, then we code it with a dash.
Classes and operations are coded by a different number of bits. The first \( n \) bits enumerate the model elements. The value of \( n \) depends on the number of elements of the model and will be determined during the reading process. Next we reserve \( a \) bits for class attributes. A class will be coded therefore by \( K \) bits, \( K = n + a \) (Fig. 3). For example, the class `Order` carries the number 6 and covers four attributes, thus their dependency vector is 0110----.

Additionally to the operation number we need the information whether the function is a constructor and whether it returns a value. Further one bit for the function result and \( p \) bits for values of function parameters must be reserved. The function code needs therefore \( O \) bits, \( O = n + 3 + p \), (Fig. 4).

For example, the operation `remind(order : Order, date : Date) : Boolean` is coded by the number 7, it is not a constructor, return a value and has two parameters. The associated dependency vector is 011101----.

If a function is not a constructor and does not return a value, its result bit will be always one that means the function is returning always a correct result. In order to generate test templates for a business process, described with a sequence diagram, we will create at first separate test template lists for all classes and functions contained in this diagram, using the corresponding Boolean code.

All attributes of a class and each function parameter, which influence the test result, are coded with dashes. Each such dash should take once a correct
and once a wrong value. If a Boolean code consists of \( n \) dash elements then we can get \( 2^n \) different test templates after replacing by one or zero. This number of test templates may be reduced, when the sequence of test template values will be considered. That means if already the first dash of a test template is substituted by zero, the remaining dashes do not have to be fixed to zero or one, because the test result should be false anyway.

In general, a test template is a ternary vector where at least one dash of an independent value in the dependency vector is replaced by a one (correct value) or a zero (wrong value).

To create the list of test templates for a class or an operation we replace at first the first dash element in the corresponding Boolean code with a zero. It is already the first test template in the list. Then we replace in this first test template zero-element with a one and the next dash element with a zero. This test template will be added to the test template list. We repeat this procedure for each dash. At the end we get a test template in which all dashes are substituted by ones that means all elements get the correct value.

<table>
<thead>
<tr>
<th>Corporate Customer class #</th>
<th>Corporate Customer class attributes</th>
<th>billForMonth operation #</th>
<th>No constructor return no value Result is always true Function parameter</th>
<th>Order class #</th>
<th>Order class attributes</th>
<th>dispatch operation #</th>
<th>No constructor return no value Result is always true</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 1</td>
<td>0 - -</td>
<td>0 0 1 1 1</td>
<td>0 1 1 0</td>
<td>0 0 0 0 0</td>
<td>0 - - -</td>
<td>0 0 0 0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0 0 1 0 1</td>
<td>0 1 0 -</td>
<td>0 0 1 1 1</td>
<td>0 0 1 0</td>
<td>0 0 0 0 0</td>
<td>0 1 0 -</td>
<td>0 0 0 0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0 0 1 0 1</td>
<td>1 1 1 0</td>
<td>0 0 1 1 1</td>
<td>0 0 1 0</td>
<td>0 0 0 0 0</td>
<td>0 1 0 -</td>
<td>0 0 0 0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0 0 1 0 1</td>
<td>1 1 1 0</td>
<td>0 0 1 1 1</td>
<td>0 0 1 0</td>
<td>0 0 0 0 0</td>
<td>0 1 0 -</td>
<td>0 0 0 0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0 0 1 0 1</td>
<td>1 1 1 0</td>
<td>0 0 1 1 1</td>
<td>0 0 1 0</td>
<td>0 0 0 0 0</td>
<td>0 1 0 -</td>
<td>0 0 0 0 1</td>
<td>0 0 1</td>
</tr>
</tbody>
</table>

To create the test template list for a sequence diagram the separate test template lists should be combined. The sequence of this connection is very important.

A sequence diagram describes a business process with many messages (see Fig. 2). Each message has a sender object and a receiver object. To the
initially empty test template list of a sequence diagram will be added therefore at first the test template list of a sender object, then the test template list of a receiver object and at last the test template list of the operation, corresponding to the message. We chain up two test template lists such that the last test template with all correct values from the first list will be supplemented by the test templates from the second test template list. The last row has the same property after the chaining of two test template lists. Thus this row can be used for the next chaining. As example the test template list of the sequence diagram above is shown in the table 1. The test template lists for single model elements are separated by colors.

5. Parameterization of Test Templates

The generated test template is described as a row of coded numbers of test elements and information about their correctness. In order to execute tests, we need the parameterized test templates, called test cases. A test case carries concrete values associated to the true – false information of the test templates. To get test cases we have to carry out a sequence of three tasks (Fig. 5). The generated test templates must be analyzed at first. The second task is to find suitable values for test elements. At last the values must be substituted in the test templates. We assigned these tasks to three models, called analysis model, design model and interpreter model.

Fig. 5. Three tasks for parameterization of test templates into test cases.

The analysis of test templates comprises the check of each row of the test template list. This search contains the encoding of numbers of elements, access to the internal UML model and saving of relevant information in a special list of test template elements. Supplementary, the list contains for each test template element the information about correctness.

Using this list we define, for which of elements values are needed. The testing person has to decide about these values. We have developed a graphical user interface to query these values. The user can input one or more values
for each element. The input of more than one value for one element causes several test cases associated to one test template. The values of all test cases are saved as connected groups in a data base.

The substitution of the captured values into test templates is the last task of parameterization of test templates. The list of test elements and the list of values are combined into the list of test cases. Figure 6 shows this list inside of the interpreter model as a class diagram. The list contains for every test template an entry. The entries consist of pairs of a test template element and a corresponding value.

![Fig. 6. Class diagram of interpreter model.](image)

6. Experimental Results

The represented test method was carried out with some examples of models, created with the CASE tools Together and Rational Rose. Table 2 shows the results of this examination.

<table>
<thead>
<tr>
<th>#</th>
<th># model elements</th>
<th># test templates</th>
<th>maximal # TT elements</th>
<th>maximal # bits</th>
<th>generation time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>12</td>
<td>10</td>
<td>34</td>
<td>31</td>
</tr>
<tr>
<td>2</td>
<td>19</td>
<td>34</td>
<td>28</td>
<td>106</td>
<td>47</td>
</tr>
<tr>
<td>3</td>
<td>26</td>
<td>12</td>
<td>11</td>
<td>85</td>
<td>47</td>
</tr>
<tr>
<td>4</td>
<td>27</td>
<td>14</td>
<td>13</td>
<td>99</td>
<td>91</td>
</tr>
<tr>
<td>5</td>
<td>37</td>
<td>15</td>
<td>12</td>
<td>85</td>
<td>78</td>
</tr>
<tr>
<td>6</td>
<td>43</td>
<td>14</td>
<td>12</td>
<td>78</td>
<td>109</td>
</tr>
<tr>
<td>7</td>
<td>44</td>
<td>19</td>
<td>18</td>
<td>267</td>
<td>110</td>
</tr>
</tbody>
</table>

The second column of Table 2 contains the number of model elements and the third column the numbers of generated test templates in a Boolean model. It is obvious that the number of test templates in the third column and the maximal number of test elements in the fourth column are not determined by a number of model elements. Only the number of classes, their attributes and methods together with their parameters in sequence diagrams is decisive.
for the number of test templates. The last column includes the time in milliseconds of the transformation of the XMI file into the Boolean model and the generation of all test templates.

There is a connection between the number of test elements in the fourth column and the number of code bits in the fifth column of Table 2. The maximum number of bits corresponds to the last row of the Boolean test template list.

7. Conclusions

Our results show that a high level test of discrete devices is possible. Comprehensible object oriented models can be quickly entered into a CASE tool like Together or Rational Rose using the unified modeling language UML. The transfer of this model into our test tool uses a XMI file.

The suggested simple algorithm for generating of test templates based on a Boolean model created inside of our test tool from the UML model of the discrete device. The ternary representation of a test template reduces the test expense significantly.

A test template describes by Boolean values whether attributes and parameters have to be correct (true) or wrong (false). Thus each test template covers a very large set of concrete test cases.

For practical reasons only a small subset of test case can be included in a finite test. At least one test case must be created by parameterization from each test template. In order support the test engineer in this important task, we implement an adaptive user interface.

One of the main advantages of our approach consists in the general applicability of the test generation for several types of discrete devices.

References


