

Graph Transformations for Model-based Testing

Mirko Conrad, Heiko Dörr, Ingo Stürmer
DaimlerChrysler AG
Research and Technology
Alt-Moabit 96 A, D-10559 Berlin
{Mirko.Conrad, Heiko.Doerr,
Ingo.Stuermer}@DaimlerChrysler.com

Andy Schürr
University of the Federal Armed
Forces, Munich
Werner-Heisenberg-Weg 39.
D-85579 Neubiberg
Andy.Schuerr@UniBw-Muenchen.de

Abstract: Model-based development uses modeling and simulation as essential means for specification, rapid prototyping, design, and realization of embedded systems. The classification-tree method complements model-based development with a formal approach for test case description and automation. This paper shows how “raw” classification trees are transformed into complete classification trees using an extensible tool, the classification-tree transformer (CTT). This tool and its domain specific extensions are generated using the graph rewriting system PROGRES.

1. Introduction

The use of software is of increasing importance for the competitiveness of mechanical systems. Many systems would not be able to serve their purpose without embedded software components. Examples are large systems like aircrafts or space systems, but also small ones like mobile phones or airbag systems. Domain-specific development methods and tool environments have emerged with the evolution of embedded (sub-)systems. In particular, elaborated methods for control engineering like modelling by function block diagrams and simulation are nowadays applied during embedded software development.

This paper shows how quality assurance by means of testing can be further automated within a model-based development process. In particular, the well-established classification-tree method (CTM, [GG93a]) for systematic testing of software and systems in general has been reassigned to models developed with the de-facto standard modeling and simulation tool Matlab/Simulink/Stateflow [IAC98], [IAC00], [ML00], [Je00]. Within this approach, several test development steps have already been automated exploiting the information incorporated in the model. Still the execution of some steps are left to the human developer since they are engineering tasks requiring ingenuity. Nevertheless, when rules-of-thumb have been developed for a certain domain they can also be fixed and exploited for further automation.

This paper shows how fixed heuristics applied in a specific step in the testing method are captured by transformation rules. For this purpose, the paper introduces in section 2 model-based testing and the application of the classification-tree method within a model-based development scenario. Automatically executed classification-tree extensions are a major step missing in this scenario. Hence, the requirements for extension rules are set up in section 3. These extension rules are specified by graph transformation and introduced in section 4. Section 5 shows the particular graph transformation rules which implement a specific classification-tree extension. A conclusion is given at the end of the paper.

2. Model-based Testing

The source of the information which is transformed into test scenarios will be used for a proper categorization of testing methods. For the sequel the term *model-based testing* is used for those software testing techniques in which test scenarios are derived from an executable behavioral model of the software. According to our approach, model-based tests consequently take up an intermediate position between functional tests on the one hand, in which the test scenarios are derived from the functional specification (only the interfaces of the object to be tested are to be considered here), and the structural tests on the other hand, in which the structure of the object to be tested is considered.

Model-based testing is studied by scientific and industrial research for some years. A model-based development process enables a tight integration of development and testing activities. A lot of the information contained in the behavioral model can be utilized for the automation of this testing process. Furthermore, the early accessibility of an executable behavioral model enables most testing activities to be based on it and, therefore, to be started at early development stages.

Our approach to model-based testing utilizes the classification-tree method. This method is a specific instance of black-box partition testing, partly using and improving ideas from the category partition method [OB88]. The classification-tree method has been used successfully in various fields of application at DaimlerChrysler. Commercial tool support is available with the classification-tree editor (CTE, [GG93b], [CTE01]). The classification-tree method is an instance of partition testing where the input domain of the test object is split up under different aspects usually corresponding to different input data sources. The different partitions, called *classifications*, are subdivided into (input data) equivalence *classes*. Finally different combinations of input data classes are selected and arranged into test sequences.

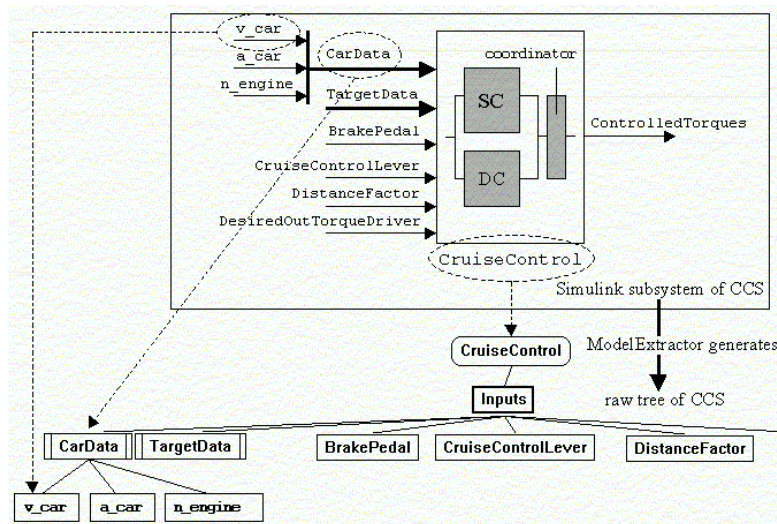


Figure 1: CCS Model (Simulink/Stateflow) and its raw classification tree

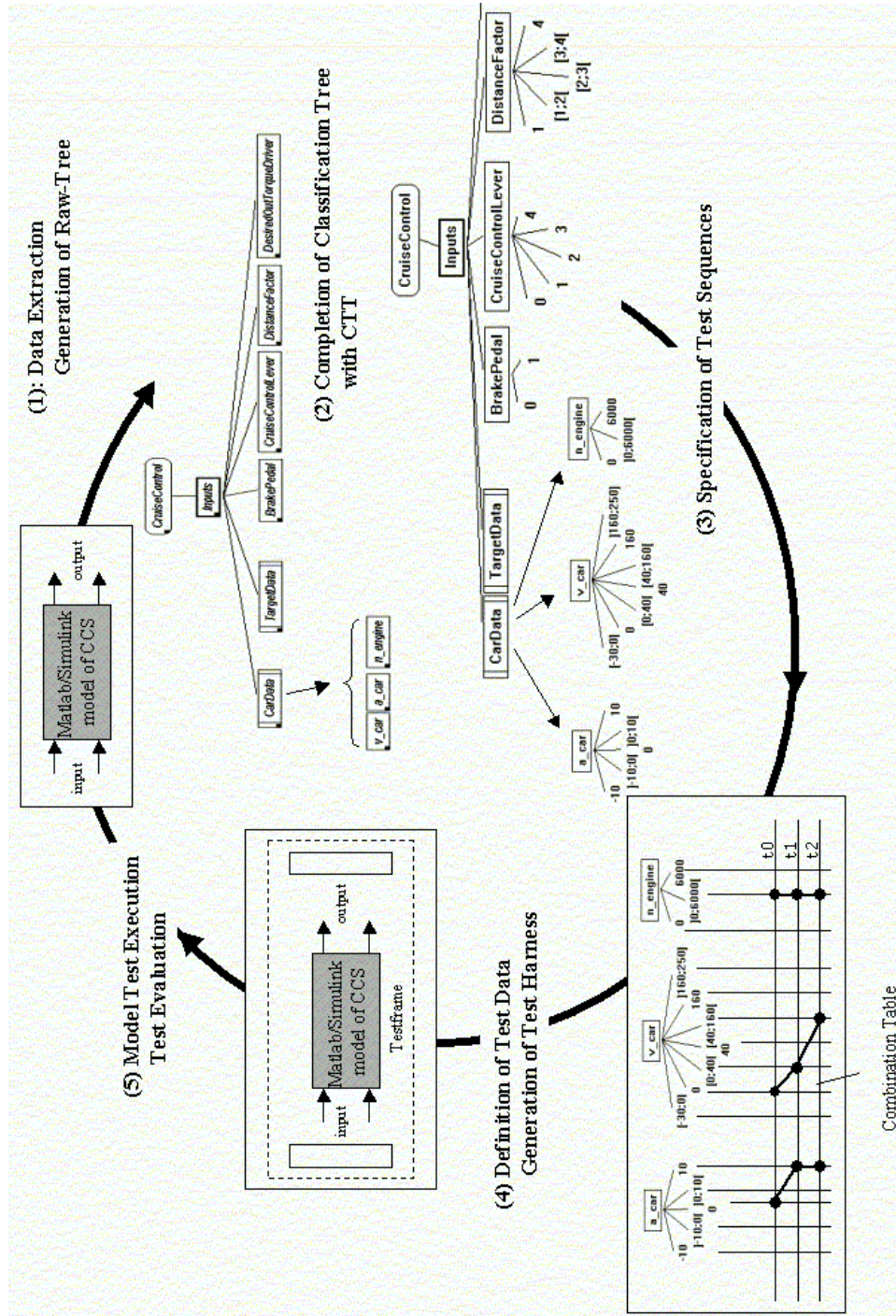


Figure 2: Model-based testing cycle

In this paper we will use the following example of a cruise control system (CCS [CH98], top of Figure 1). Upon the driver's request (e.g. BrakePedal), the CCS controls the speed of the car ensuring a safe distance to the vehicle driving ahead. If there is no vehicle driving ahead, the CCS controls the speed. Once a vehicle driving ahead has been recognized, a distance control is activated. The system which is controlled by the cruise control lever contains separate speed (SC) and distance control (DC) components. Both controllers define desired output torques (e.g. n_engine) that are coordinated by a downstream component according to the current driving situation.

The *model-based test* of the CCS consists of five steps (Figure 2). The subsystem of the behavioral model (here CCS), developed with Matlab/Simulink/Stateflow, serves as a basis for the derivation of the test scenarios.

1. A so-called *model extractor* (see bottom of Figure 1) analyses the model systematically with respect to the test scenario specification. Relevant data will be automatically extracted as tree elements. The name of the test object itself forms the root node (here: CruiseControl). Input signals (e.g. CruiseControlLever), elements of input busses, i.e. collections of input signals, and internal model parameters are extracted from the model and denoted as classifications below the root node. The input busses, e.g. CarData, are represented by refinement nodes. The refinement for CarData aggregates the classifications of the individual input signals v_car , a_car , n_engine , and $i_transmission$. The generated tree elements, describing the inputs and parameters of the model, and informations for the hierarchical structure of the inputs (e.g. busses, vectors) are processed by the model extractor which fully automatically outputs a first, incomplete instance of a classification-tree called *raw tree*.
2. The generated raw tree is transformed into a *complete classification tree*. For this purpose all generated classifications of needed input signals and parameters must be disjointedly and completely partitioned into (equivalence) classes which are suitable abstractions of individual input values for testing purposes. The range of the binary signal BrakePedal is now divided into class 0 (not activated) and class 1 (activated). Vehicle speed v_car , which can take on continuous values between -30 and 250 km/h, is divided into one interval for all negative values, three intervals for all positive values, and separate classes for the special cases 0, 40, and 160. Step 2 completes the systematic analysis of the input space.
3. Based on the input partition *test scenarios* can be determined. These scenarios specify how the behavior of the regarded unit under test should be evaluated. The domain for the description of test scenarios is provided by the completed classification-tree. The tree is used as head of the combination table. Each scenario captures a data abstraction of the behavior of the unit under test and, hence, describes – largely independent from concrete data – what is to be tested. In order to represent test scenarios in an abstract way, they are decomposed into individual test steps. Each test step defines the input situation at a certain time. A sequence of such test steps is called a test sequence (cf. [Co00]).
4. In a fourth step, the test sequences will be instantiated by discrete waveforms. Test harnesses, which stimulate the test object with the defined test-data series and record test results, are generated automatically and fed with the input vectors.

The tests are executed within Matlab/Simulink/Stateflow using the model-based test tool MTest [Co99].

5. Finally test evaluation is performed by systematic investigation of the execution results. The test sequences as defined in the context of model testing as well as the captured results may be reused for software testing and embedded system-testing. In doing so, there will be regression test cycles reduced to steps 4 and 5 above.

To summarize, raw classification trees are automatically created by the model extractor. However, there was no efficient tool support for step 2, the raw tree completion. Closing the gap we developed a tool, the CTT [St00] (cf. section 3 and 4), automating a manual step in the model-based testing cycle.

3. Tool Support Requirements

The cycle of model-based test shown in Figure 2 is a mixture of automated and manually performed steps, where the automated steps mostly perform data transformations. E.g. the extraction of the interface information into the raw tree is an isomorphic function which translates a subsystem's interface into a raw tree. Experience in the use of the classification-tree method has shown that the next step in the test cycle, namely the completion of the raw tree by definition of classes, can be partially automated, too. In this step, classes must be defined to partition the value space of input signals. For this test design step a number of heuristics have been developed which led to further automation steps. They could be divided into three testing heuristics.

1. *Data type related heuristics*: Some of the heuristics are often related to the data type of the signal. E.g. the classification of a Boolean signal is set up by two classes *true* and *false* or enumeration types are classified by setting up a class for each enumeration value. A signal of the type double with an initial value *init* and a permissible range restricted by *min* and *max* would be divided into the following classes: *min*, *]min;init[*, *init*, *]init;max[* and *max*. Another example is the division of a predetermined value domain into *n* equal-sized sub-intervals. The partition of all generated classifications of input signals into disjoint sets of “useful” equivalence classes is left open to the engineer although this task is usually straightforward and guided by project independent rules.
2. *Problem-specific partitioning heuristics*: An example are the speed signals in the CCS. Typically, velocity values range between -30 and 250 km/h. Furthermore, in the CCS context there is a distinguished speed range (40 to 160 km/h) where the cruise control can be activated by the driver. Above and below the threshold values it is not possible to use the cruise control functionality. The initial value coincides with the special case stationary vehicle (0 km/h). This leads to the problem-specific classification of the velocities into the intervals $[-30;0]$, 0 , $]0;40[$, 40 , $]40;160[$, 160 , $]160;250]$. Such a problem-specific partitioning can be determined by pattern-matching on the signal names or the signal value domains. The CCS example uses the prefix ‘v_’ for velocities by convention.
3. *General testing heuristics*: Finally, a third type of heuristics will be applied during the classification step related to general testing heuristics. Testing practice has for example shown that many faults occur due to improper handling of boundary val-

ues. Hence, a boundary value testing heuristics states that in particular the areas around specified boundary values should be investigated. Applied to the current setting it follows that a given classification should be refined. Instead of just checking intervals including the boundary value further tests should be executed around the boundary. For instance, an integer signal with negative minimum values and positive initial and maximum values could be divided into the classes: $] -inf; min-1]$, min , $[min+1; -1]$, 0 , $[1; init-1]$, $init$, $[init+1; max-1]$, max and $[max+1; inf[$.

Besides the test heuristics which are applied for raw tree completion further tree transformations may be applied for structure refinement. For example, a classification with high tree-width might be restructured by introduction of further structuring levels. Or, as another application example, a large classification tree may be separated into several parts. To summarize, it is possible to identify the following categories of classification-tree transformation rules:

- *Data type specific rules* define standard classifications for several data types.
- *Domain specific rules* enable a problem-specific partitioning.
- *Error type specific rules* generate additional classes for error-prone special cases.
- *Structuring rules* are used for redesigning the shape of a generated classification tree.

The heuristics for classification are becoming stable so that it makes sense to automate the step of classification-tree transformation. On first sight, all heuristics could be implemented in a straightforward manner and built into the classification-tree editor. So, the standard functionality of the tool would be enhanced by a fixed set of heuristics. This approach has been taken for some examples. Quite soon it turned out that the rapid development of especially domain specific heuristics could not be supported since hard-coding of the rules has been too cumbersome and not accessible to test engineers. Consequently an approach had to be found which, first, enables the *domain aware test engineer* to specify his heuristics such that, second, a transformation engine can be generated which performs the tree completion according to the specification. The specification language should be as close as possible to the tree view used in the classification-tree editor. So, at least tree transformations should be used for specification purposes. Moreover, the transformation rules must be collected in sets to build up a library of test heuristics which can provide tree extension rules for specific application domains or different projects.

4. Tree Transformation Rules

A generic framework must be identified to implement the classification-tree transformation. Implementing all these tree transformations by hand in a programming language like Java is not appropriate for a test engineer who wants to document his experience. As a consequence a very high-level language or tree processing system is

and debugging of attributed graph transformations, which are a superset of context-sensitive tree transformation rules. A rather sophisticated pattern matching algorithm guarantees for restricted classes of graphs (e.g. for graphs with a tree skeleton) a rather efficient execution of specified rules. Furthermore, PROGRES combines an incremental attribute evaluator with a procedural sublanguage for the development of parameterized rule application strategies, and it supports interactive debugging as well as the generation of standalone graph processors (implemented in C with an optional tcl/tk-based graph display interface).

The transition from trees to graphs enforced us to select a standard “encoding” of classification trees as classification-tree graphs, henceforth called *CT graphs*. Figure 3 shows one example of the CT graph representation of a raw classification tree (a cut-out of the CruiseControl tree of Figure 1). Next-edges are used to order the subnodes of a parent node, whereas first- and last-edges point from a parent node to its first and last child.

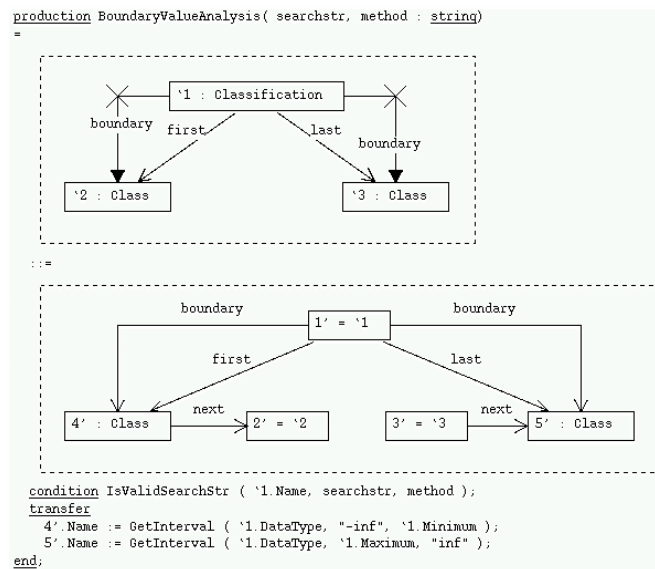


Figure 4: Boundary Value Analysis CT Graph Extension

A first example of a graph transformation rule programmed in PROGRES, which adds boundary values to any node whose name has the prefix ‘v_’, is presented in Figure 4. This rule (production) is parameterized in order to be reusable if the conventions for the definition of velocity value names change. Its first parameter *searchstr* provides a substring such as ‘v_’, whereas the second parameter *method* determines whether the given substring has to be a prefix, infix, or postfix of a regarded classification-tree (graph) node name. The left-hand side of the production (the graph above ‘:=’) matches a classification node with at least two different classes as subnodes. Furthermore, the rule’s left-hand side requires that these subnodes are not yet the result of a “boundary value analysis” extension process, i.e. it requires the nonexistence of (crossed-out) boundary edges between a regarded classification node and its first and

last child. Finally, the expression below the keyword ‘condition’ uses an elsewhere defined function (written in PROGRES itself or in C) to check that the given parameter *searchstr* is a substring of the name of the regarded classification node. The rule’s right-hand side (the graph below ‘:=’) merely adds two new class nodes as the first and last child of the selected classification and connects them to the rest of the processed CT graph via first-, last-, next-, and boundary-edges appropriately. These new nodes receive appropriate values for their *Name*-attributes by the assignments below the keyword ‘transfer’. Furthermore, the rule preserves all three nodes matched by its left-hand side, indicated by nodes with inscriptions of the form $n' = n$ in its right-hand side.

5. Rule Design and Application

In section 3 categories of classification-tree transformation rules (e.g. data type specific rules) were mentioned. One example of a simple graph transformation rule was presented in Figure 4. The complete PROGRES specification of the CTT kernel consists of the following sections:

- A graph schema (class diagram) defines all classification tree elements produced by the model extractor for Matlab/Simulink/Stateflow models. This graph schema must be extended in the general case if new properties of processed models are regarded or if new modeling tools are integrated.
- A first group of graph transformation rules offers all services needed by a *parser* that translates a textual classification tree description into a raw CT graph.
- A second group of rules implements a recursively defined *unparsing* process from CT graphs into text files which relies on derived attribute evaluation mechanisms.
- All other sections contain different categories of CT graph *extension rules* ranging from low-level data type specific rules to domain specific rules as introduced in section 3.

New (groups of) rules may be defined by either constructing new graph transformation rules such as the one for boundary value analysis from scratch or by combining already existing rules into more complex ones. For this purpose PROGRES offers various kinds of control structures, the functional abstractions of which are called *transactions*.

Let us assume that the following basic graph transformation rules are available:

- `CreateClassification(parent, type, name)`: adds a new classification node of the given type as the last child to a given classification tree node parent.
- `FindAllNodesOfType(type, out nodeSet)`: returns the set of all nodes in the regarded CT graph which have the type `type`.
- `CreateNumberIntervals(searchstr, method)`: adds to all classifications with a given `searchstr` as prefix, infix, or suffix (determined by `method`) of their *Name*-attribute a sensible set of intervals as equivalence classes.
- `BoundaryValueAnalysis(searchstr, method)`: adds to a classification with a set of intervals as classes two additional classes as described in section 4.

Based on these four rules a new CT graph transformation may be constructed which

- (1) identifies all classifications which represent input vectors (arrays of a given data type with a fixed dimension) and returns the result as a set of CT graph nodes of type Vector,
- (2) creates a new subclassification for each dimension as a child of the regarded Vector node which has the same name as its parent node except of an attached dimension number suffix, and
- (3) applies then the standard rules for subdividing these new classifications into sensible subclasses (the actual parameter values of these rules guarantee that just created CT graph nodes are processed).

Figure 5 shows the definition of the needed complex CT graph transformation (transaction) `HandleVectors`. In this case the PROGRES language itself has been used to specify a new CT graph transformation strategy by combining a number of already existing graph transformation rules appropriately.

```
transaction HandleVectors =
  FindAllNodesOfType ( Vector, out nodeSet )
  & for_each n:= elem ( nodeSet ) do
    for_each d:= 1 .. node.Dimension do
      CreateClassification ( n, n.Type, n.Name & string ( d ) )
    end
    & CreateNumberIntervals ( n.Name, "prefix" )
    & BoundaryValueAnalysis ( n.Name, "prefix" )
  end
end;
```

Figure 5: Complex CT graph transformation

6. Conclusion

The classification-tree extension heuristics and their generic implementation within the tool CTT provides another step in the automation of model-based testing. It reduces the amount of human activities within model-based testing to the real engineering tasks. Repetitive efforts which are guided by the application of well-defined heuristics are made accessible for automation by the specification of transformation rules. These rules take a raw classification-tree as input and produce an extended classification-tree, where (almost) all input data classifications are partitioned into appropriate equivalence classes.

Not only a single test project is speed up, but testing experience can be easily re-used in following projects and spread in an organisation. By accumulation of a body of test knowledge, the maturity of the organisation increases.

The classification-tree extension has been realised based on the graph transformation programming environment PROGRES. By means of the powerful generation mechanism provided by PROGRES, a tree transformer is generated out of a set of extension rule specifications. Furthermore, the generation mechanism is customised such that any generated transformer closely integrates into the environment set up for the model-based test of Matlab/Simulink/Stateflow models. As an integral part of the testing environment MTest the classification-tree transformer functionality will be available soon. In the future the rule-based tree transformation will be exploited for other tasks within the classification-tree method. For instance, consistency checks on

the tree will be specified and executed by means of graph transformations, too. The CTT is currently being pilot-operated with respect to its contribution to shortening the testing times and the acceptance of the user interface.

7. References

- [CH98] Conrad, M., and Hötzer, D. Selective Integration of Formal Methods in the development of Electronic Control Units. In Proc. of ICFEM'98 (Brisbane, Australia, Dec. 1998), IEEE Press, 144-155 (1998)
- [Co00] Conrad, M., Dziobek, C., Fey, I., Keller, H., and Rau, A. CSD and MTest: Model-based Software Development and Testing, in [IAC00] (2000)
- [CHP91] Cordy, J.R., Halpern-Hamu, Ch.D., Promislow, E. TXL: A Rapid Prototyping System for Programming Language Dialects. *Computer Languages*, 16(1), 97-107 (1991)
- [CTE01] Classification-Tree-Editor for Embedded Systems (CTE/ES), User Guide, Razorcat Development GmbH, Berlin (2001)
- [GG93a] Grochtmann, M., and Grimm, K. Classification Trees for Partition Testing. *Software Testing, Verification and Reliability*, 3, 63-82 (1993)
- [GG93b] Grochtmann, M., Grimm, K., and Wegener, J. Tool-Supported Test Case Design for Black-Box Testing by Means of the Classification-Tree Editor. Proc. of EuroSTAR '93 (London, UK, Oct. 1993), 169-176 (1993)
- [IAC98] Proc. of International Automotive Conference, (Stuttgart, Germany, July 1998), The MathWorks Inc (1998)
- [IAC00] Proc. of International Automotive Conference, (Stuttgart, Germany, May 2000), The MathWorks Inc (2000)
- [Je00] Jersak, M., Cai, Y., Ziegenbein, D., Ernst, R.: A Transformational Approach to Constraint Relaxation of a Time-driven Simulation Model. In Proceedings 13th International Symposium on System Synthesis, Madrid, Spain, September 2000.
- [LW00] Lehmann, E., and Wegener, J. Test Case Design by Means of the CTE XL. Proc. of the EuroStar 2000 (2000)
- [Li88] Lipps, P., Möncke, U., Olk, M., Wilhelm, R. Attribute (Re-)Evaluation in OPTRAN. *Acta Informatica*, 26(3), 213-239 (1988)
- [ML00] Merz, R., Litz, L.: Objektorientierte Mathematische Modellierung – generische Methoden bei komplexen dynamischen Systemen. *Informatik Spektrum* 2-2000, 23.4.2000, S.90-99
- [OB88] Ostrand, T.J., and Balcer, M.J. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6), 676-686, (June 1988)
- [RT88] Reps, T. and Teitelbaum, T. *The Synthesizer Generator: A System for Constructing Language-Based Environments*, Springer-Verlag (1988)
- [St00] Stürmer, I., *Graphtransformationen zur Manipulation von Klassifikationsbäumen beim modell-basierten Testen*, diploma thesis, Institute for Software Technology, Department of Computer Science, University of the German Federal Armed Forces, Munich, 2000
- [SWZ99] Schürr, A., Winter, A.J., and Zündorf, A. *PROGRES: Language and Environment*. Ehrig, H., Engels, G., Kreowski, H.J., and Rozenberg, G. (eds.) *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, 2 487-550, World Scientific (1999)
- [WTP00] Wang, W., Tao, K., Palsberg, J. JTB 1.2.2 – Java Tree Builder,

[XSL99] <http://www.cs.purdue.edu/jtb/>, Purdue University (2000)
W3C Recommendation. XSL Transformations (XSLT) – Version 1.0,
<http://www.w3.org/TR/xslt> (16 Nov. 1999)