Institute for Software Integrated Systems
Vanderbilt University
Nashville,
Tennessee 37235

TECHNICAL REPORT

TR #: ISIS-06-701

Title: Automated Test Case Generation for Code Generators based on Graph-Transformation Rules

Author: Sandeep Neema, Ingo Stürmer

3

Abstract

In automotive software development models are the central artefact within the whole development process (model-based development). The new technology of automatic code generation closes the gap between the software design on the base of a model (executable specification) and its implementation (controller code). This paper shows how to test formally specified code generators and gives guidance in test case determination for specific code generator transformation rules.

1. Introduction

The development of embedded software has become increasingly complex and abstraction appears the only viable means of dealing with this complexity. In the automotive sector, the way embedded software is developed has changed, in that executable models are used at all stages of development, from the first design phase up to implementation (model-based development). Such models are designed with popular and well-established graphical modelling languages such as Simulink and Stateflow from The MathWorks [1]. Some recent approaches allow the automatic generation of efficient code directly from the software model via so-called *code generators*, such as the Real-Time Workshop by The MathWorks [1] or TargetLink by dSPACE [2]. A code generator is essentially a compiler in that it translates a source language (a graphical modelling language) into a target language (a textual programming language). Code generators are examples of dependable software tools upon which software designers rely.

Code generators are complex tools and testing a code generator is a crucial task. There are many ways of proving the correct behaviour of a code generator. Formal proofs are used to show the (mathematical) correctness of the code generator itself or its correct functioning during runtime (e.g. [14], [15]). Much research has been done in the application field of compiler testing, i.e. test case generation techniques. In the field of compiler testing there are two ways of generating test cases, namely, *automatic test case generation* and *manual test case generation*. The first approach yields a great number of test cases in a short time and at a relatively low cost. Most of the work uses a grammar of the source language to derive programs by systematically exercising all productions of the grammar, as originally proposed by Purdom [22]. An overview of these approaches is given in [21]. A different method is to generate test cases manually with respect to given language standards, such as the Ada testsuite ACATS [20] or testsuites for C language conformance such as [17],.

When dealing with a code generator which translates a graphical source language (e.g. Simulink or Stateflow) into a textual target language (e.g. C or Ada), a natural approach consists in representing the generator via a set of graph transformation rules. Besides providing a clear and understandable description of the transformation processes, this formal specification technique can be used for test case derivation, allowing the specific and thorough testing of individual transformation rules as well as their interactions. This approach has been presented and discussed in [13] (see chapter 5 for an overview). An important advantage of this test approach for code generators is its high automation potential from test case design through to automatic test evaluation. The proposed testing method has been tried and tested for code generators implemented in a classical programming langauge such as C/C++ based on informal (e.g. textual) specification.

In this paper, we will focus on a code generator whose implementation is derived directly from its formal specification, i.e. graph transformation rules. The test case derivation method proposed allows effective test case generation for the exhaustive testing of specific transformation rules. The input to this code generator is a graphical modelling language known as the Embedded Control Systems Language for Distributed Processing (ECSL-DP). This language and the set of tools built around it provide a seamless tool-chain for design, simulation, and synthesis of distributed embedded automotive applications.

We initiate the rest of the discussions by introducing ECSL-DP, and the tool-chain built around it (section 2). Section 3 discusses principles of automatic code generation and shows how code generation could be specified with graph transformation rules. Section 4 is dedicated to code generator testing and introduces test case generation which starts from a formal specification and leads to executable models. Section 5 gives an overview on the code generator test approach and shows how to compare a code generator test case (i.e. input model) with its generated code. Finally, section 6 concludes the paper.

2. Embedded Control Systems Language for Distributed Processing (ECSL-DP)

ECSL-DP is a graphical design modelling language supported by Graphical Modelling Environment (GME), a meta-programmable graphical modelling tool developed at Vanderbilt University [3]. For GME, a modelling language is defined in terms of meta-models that capture the abstract syntax of the language. ECSL-DP has concepts and constructs suited for modelling distributed automotive embedded applications. The key categories of modelling elements in ECSL-DP include:

1. Dataflow Modelling – for hierarchical dataflow-diagram oriented modelling of signal flows, representing the functional design.

2. Stateflow Modelling – for hierarchical state machine diagrams, representing finite-state behaviour, in the functional design.
3. Component Modelling – for modelling of software components and partitioning of functional design over software components. Components are software artefacts that get instantiated on an embedded platform.
4. Hardware Topology Modelling – for modelling of the topology of the distributed platform including ECU-s, Buses, their physical ports, and their connectivity.
5. Deployment (Mapping) Modelling – for modelling of the deployment of components on ECU-s, including association with RTOS tasks, and mapping of component ports on to physical communication conduits (sensors, actuators, and bus messages)

For illustrative purposes and for later reference we briefly describe the Stateflow modelling category with its meta-model here (see [5] for details). GME uses UML-style class-diagrams to specify meta-models, which capture the abstract syntax for the models. How model elements are actually visualized in GME is determined by their stereotypes, which in effect provides a binding of the abstract syntax to concrete syntax (see the GME documentation [4] for details).

Figure 1 illustrates the Stateflow portion of the ECSL-DP meta-model, that supports the modelling of hierarchical finite state machines, semantics of which are described in [6]. The `Stateflow` folder contains `State<<Model>>`s, which are root models for hierarchical state machines. Each `State` can contain a number of `Data<<Atom>>s`, and `Event<<Atom>>s`, and subclasses of the (abstract) `TransConnector` (as in "transition" connector) class. The subclasses of `TransConnector` include Junctions, `TransInPorts`, `TransOutPorts`, `TransStart` (as in "transition" input and output ports and starting points), `History`, and `ConnectorRefs` (which are `<<Reference>>`s pointing to objects derived from the `TransConnector` base class). States contain Transition `<<Connection>>`s. These connections connect two objects (derived from the `TransConnector` class), and represent the state transition concepts of the hierarchical finite state machine. The operational semantics of transition is the same as those of transitions in SF. In GME there are no graphical means of depicting connections between objects that are not contained in the same parent, and hence cross-hierarchy transitions are represented using the `ConnectorRef.`

A State model also contains a `BlockRef <<Reference>>`, which points to a `Block` (a dataflow modelling object not described here). This mechanism provides the linkage between a Stateflow model and a Dataflow model. This meta-model when instantiated in GME provides the embedded system developer with a modelling environment in which he can design

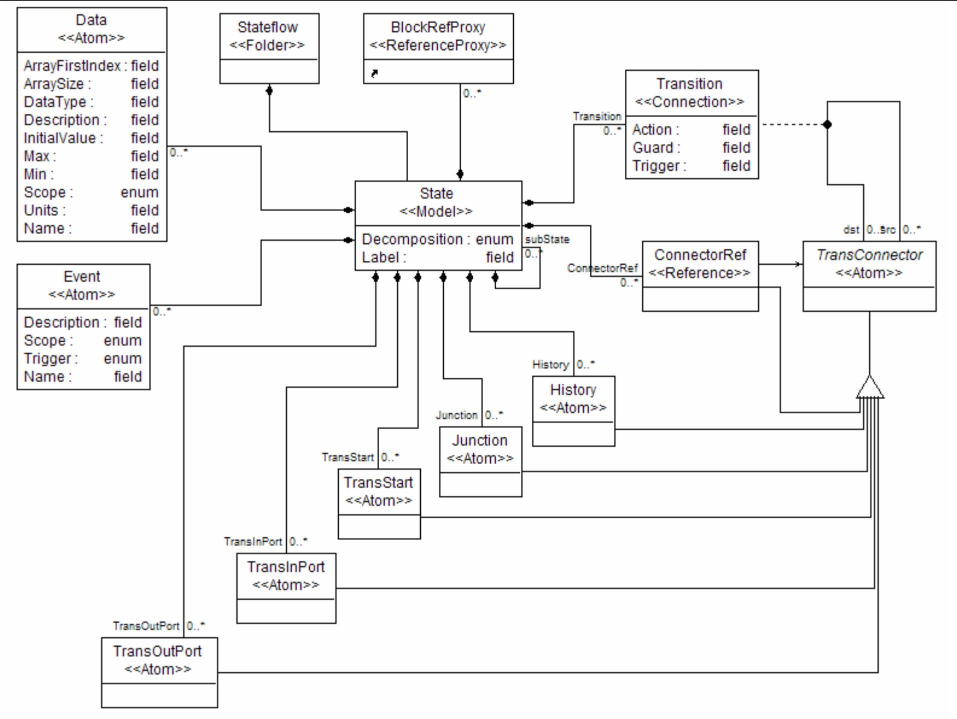


Figure 1: ECSL-DP meta-model (Stateflow modeling category)

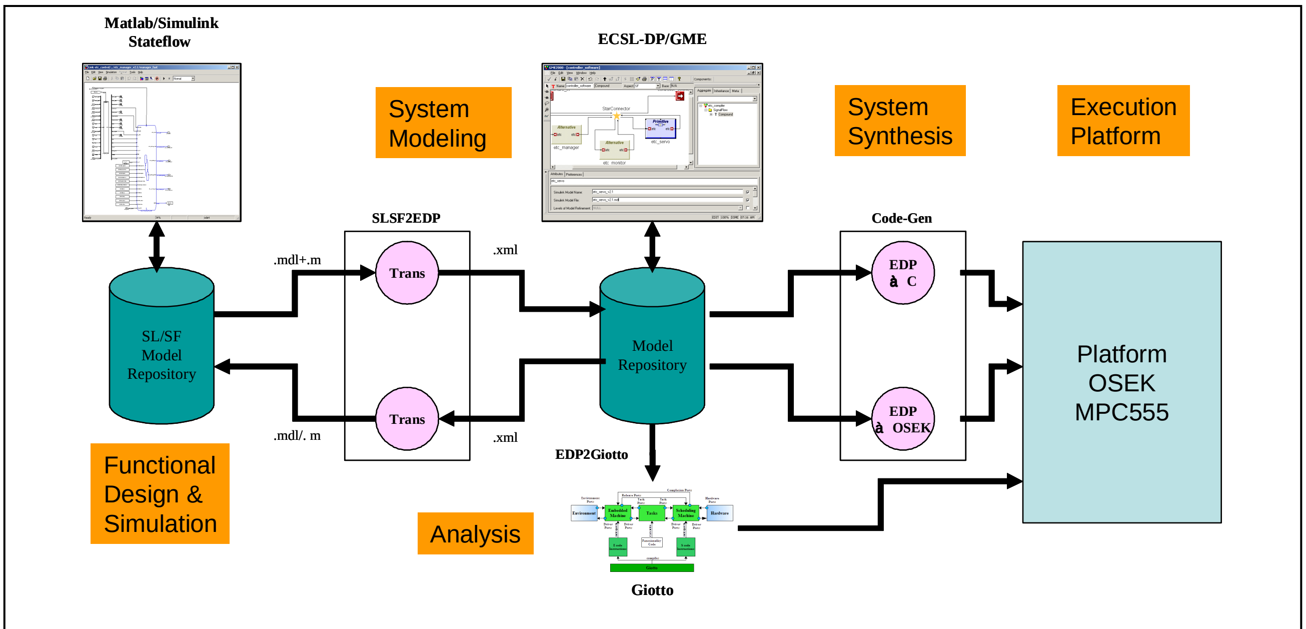


Figure 2: Embedded Automotive System Development Tool Chain with ECSL-DP

and specify the system.

2.1 ECSL-DP Tool Chain

The development of ECSL-DP was motivated by the lack of an integrated tool-chain that addresses the key aspects of a distributed embedded system development process such as functional design, platform design, deployment, analysis, and synthesis. The design intent was not to replace existing tools, but complement these tools as an integrator, by facilitating interchange, and providing convenient and open interfaces which makes it possible to integrate new tools with relatively modest effort. To that end we have developed (and are developing) a suite of translators to facilitate the syntactic and semantic interchange between tools. Figure 2 shows the prototype tool-chain developed around ECSL-DP with the following key components:

1. Simulink/Stateflow (SL/SF) – These tools are used for creating the functional design of the controller and for a hybrid simulation of the design
2. ECSL-DP/GME (EDP) – This is the ECSL-DP modelling language instantiated in GME used for component and platform modelling, partitioning functional design over components, performing the deployment modelling, and capturing the real-time properties and constraints of the system
3. Giotto – A time-triggered language used for schedulability analysis using time-triggered execution semantics.
4. SLSF2EDP – A translator that performs a mapping of SL/SF models into ECSL-DP models.
5. EDP2Giotto – A translator that synthesizes Giotto specifications from ECSL-DP models (under development)
6. EDP2C – A C code-generator from ECSL-DP models, specified as graph rewriting specifications, generates the code from functional design models.

In the next section we examine the code generator the testing of which is the focus of this paper.

3. Automatic Code Generation

The ECSL-DP to C (EDP2C) code generator deals with synthesizing implementation from Dataflow and Stateflow sublanguage of ECSL-DP. In the following discussion we will focus only on generating code from the Stateflow sublanguage of ECSL-DP.

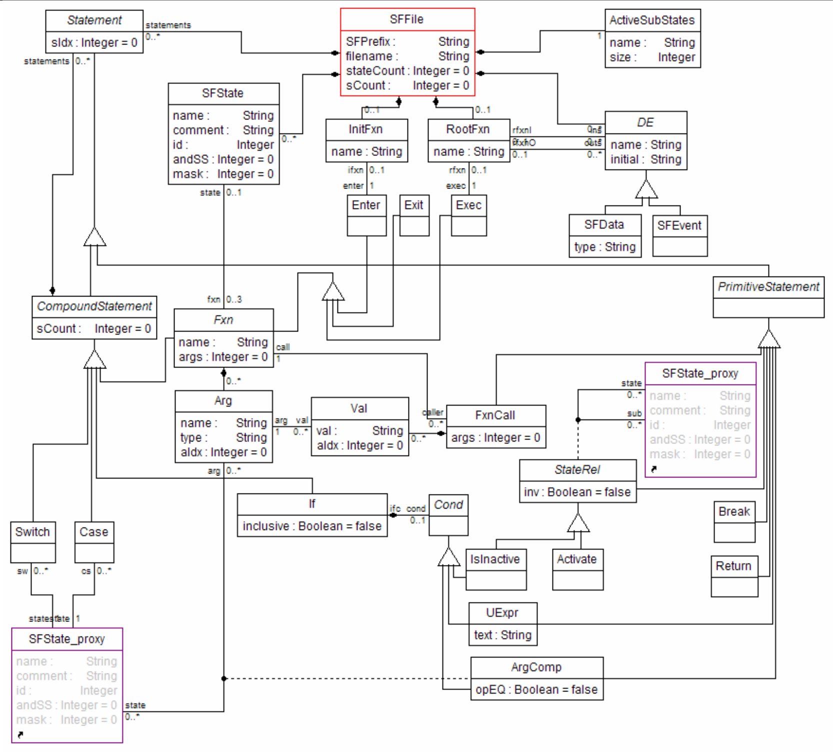


Figure 3: Meta-model of stylized C (SFC)

In general, a code generator uses a “traverse-transform-print” strategy in order to gather information from the design models, build intermediate data structures (e.g. tables) as necessary, and then output the resulting code. The Stateflow code generation follows a similar strategy; however, the transformation algorithm is developed and specified using a Graph Rewriting technique implemented in the ‘GReaT’ tool developed at Vanderbilt University [7].

The output of the code generator is a C program that implements the behaviour specified with the state-machine, according to the execution semantics described in the Matlab Stateflow documentation. The generated C code is a stylized subset of C, and we have created a UML meta-model of this stylized C, which we call SFC (see Figure 3). The output of the graph transformation is an object graph conformant to the SFC meta-model. Textual output from this object graph is generated by doing the equivalent of ‘anti-parsing’. The key entities in the SFC meta-model and what they represent are described below:

- `SFFile` – the top-level file object
- `InitFxn` – initialization function that must be invoked by the generated Simulink code once to initialize the state machine
- `RootFxn` – the main interface function that is invoked by the generated Simulink code
- `SFData`/`SFEvent` – the data, event variables within the state-machine that are the interface to the Simulink code. These variables form the input and output argument list of the root function, note the association between `RootFxn` and `DE`, the abstract base class of `SFData` and `SFEvent`
- `Enter`,`Exit`,`Exec` – these are the entry, exit, and step function corresponding to each compound state in the state-machine. `Fxn` is the abstract base class representing a function.
- `SFState` – these represent the states in the state machine, an enumeration list is printed in the generated code.
- `ActiveSubStates` – this singleton array variable represents the current list of active sub-state for each compound state in the state machine. The enumeration value of the compound state is used to index into this array to determine the current active sub-state in the generated code.
- `Statement` – this abstract base class represent code blocks in the generated code. Statements are sub-classed into `CompoundStatements`, and `PrimitiveStatements`. `CompoundStatements` are code blocks that include other

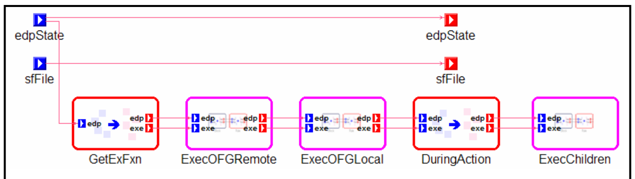


Figure 4: PopulateExecFxn Transformation Rule

`Statements`. These are sub-classed as `Switch`, `Case`, `If`, and `Fxn`. `PrimitiveStatements` are `FxnCall`, `Break`, `Return`, `ArgComp`, `Activate`, `IsInactive`, `UExpr`, etc.

For the purpose of illustration below we examine a couple key rules in the transformation specification, a complete description of the rules and transformation is beyond the scope of this paper. Please note that the transformation language implemented by GReaT, has a control flow structure in addition to the graph rewrite instructions (see [8] for details).

Figure 4 shows the `PopulateExecFxn` rule that generates code for the `Exec` fuction, which implements a step in a state-machine. Note that the purple colored boxes represent compound rules, and the blue and red colored port objects represent passing of objects to and from the rules. The ports `edpState`, and `sfFile` in the rule are bound to the state in the ECSL-DP network which is to be transformed, and the root object (a singleton instance of `SFFile`) in the SFC data-network, respectively. The generated code for the step function must check for enabled transitions leading out of this state, and if there is an enabled transition then the transition must be taken which requires a call to the exit function of the source state, performing the transition actions, and invoking the enter function of the destination state in the simplest case. If no transitions are enabled then the during action of the state must be performed, and then the step function must do a step on the sub-states. The `ExecOFGRemote`, and the `ExecOFGLocal` sub-rules of this rule emit the code for checking for enabled transitions and performing the transition step. The `ExecOFGRemote` rule handles remote transitions (source and destination state have different parents), while the `ExecOFGLocal` rule handles local transitions. The `ExecOFGRemote` rule is invoked prior to the `ExecOFGLocal` rule since cross-hierarchy transitions have a higher priority than local transitions. The `DuringAction` is a primitive rule, and we examine it next.

Figure 5 shows the `DuringAction` rule. This is a graph rewrite rules, which typically consist of a LHS which represents a pattern to be matched, and RHS which represents the modification in the graph. In this particular rule the pattern is simply an ECSL-DP `State`, and a `CompoundStatement`, which are objects passed as input to this rule. The light blue-border on the class `UExpr` represents creation of a new object instance of the `UExpr` class. Also, the blue-colored composition arrow represents creation of a composition relation between the `CompoundStatement` object and the created `UExpr` statement. In simple words this rule creates a `UExpr` object in the output data-network. The boxes labeled `am_idx`, and `am_ea` contain attribute mapping specifications. These are code snippets which are executed by the transformation engine when the pattern is matched. The red-circle labeled `hasDuringAction` is a guard which must be satisfied for the pattern to be matched. In this particular case the guard simply checks that the State has a during action.

There are ~75 rules in the complete transformation specification, which is a significant improvement over ~3000 lines of code in an equivalent transformation developed by hand. The transformation specifications can be executed by the GReaT tool in an interpretive fashion, as well can be compiled into C++ code that can be compiled and linked to build the code-generator.

4. Code Generator Testing

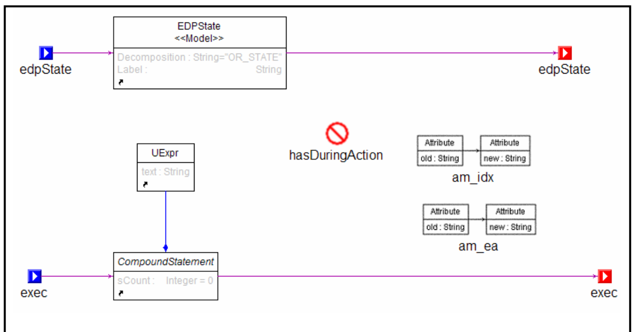


Figure 5: DuringAction Transformation Rule

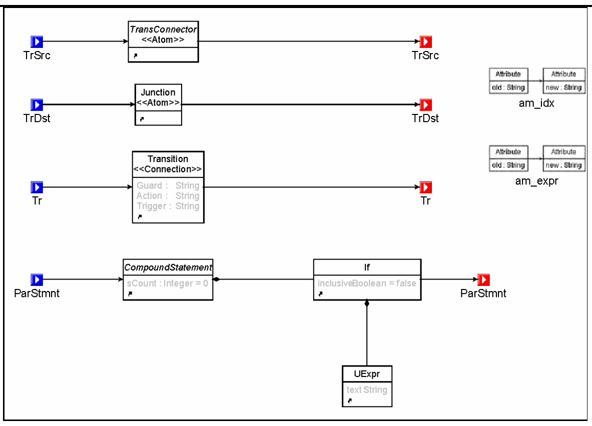


Figure 6: IfTrValid Transformation Rule

The most significant weakness of testing is that it can only ever prove that the test object functions as it should for those input situations chosen as test data (i.e. testing can show the presence of errors but not their absence). In practice, a complete test is impossible, with the exception of trivial cases, due to the large number of possible input situations. Testing is, then, a sampling procedure. Accordingly, the essential task during testing is the determination of suitable (i.e. error-sensitive) test cases, which ultimately determines the scope and quality of the test. Therefore – of all the testing activities – test case design is of decisive importance. It considerably affects test quality, since the selection of the test data to be used to test the test object determines the type, extent, and thus the quality of the test.

The inadequate testing of code generators is mainly due to the methodical inability to describe the mode of operation and interaction between complex transformations and optimisation rules clearly, thus making it possible to test them effectively. In order to do this, the program code and textual specification alone are unsuitable as a base for test case design. Therefore, an essential prerequisite for effectively testing a code generator is to choose a formal specification language which generally leads to higher quality, and also describes the functioning of the code generator as simply and clearly as possible.

4.1. Specification of Transformations

Practice shows that the use of formal techniques for test case design and generation result in high quality, reliable software products [9]. Taking account of this experience, a code generator testing approach based on graph rewriting rules (i.e. graph transformations) was proposed in [13].

With a formal specification of the translation process available, we can use the graph rewriting rules to derive test cases systematically as shown in the following section.

4.2. Test Case Design

Having a specification of the code generation process present as graph-rewriting rules, we can use this description to derive test cases. The application of graph rewriting rules for test case design has many advantages:

- Graph transformations provide the most faithful reproduction of transformation processes inside a code generator. For this reason, error-sensitive test cases can be derived from this specification.
- The correctness of the test cases can be checked (more on this later).
- High automation potential

In order to provide an example, we will use the *IfTrValid* graph transformation rule, from the ESF2C code-generator as a running example (see Figure 6). The rule generates code for a Stateflow transition as follows: with respect to the Stateflow semantics, a transitions represents an *if* branch of an *if-then-else* control structure. Each transition has a label of the form T [G]/{A}, where T is the *event*, triggering the transition, G is the *condition* (Boolean expression) guarding the transition and A is the *action* part executed. The transition will be translated into an *if-statement* of the language C. The left-hand side part of the rule consists of three node (classes), `TransConnector`, `Junction` and `Transition`. Each class is a member of the Stateflow meta-model as shown in Figure 1. If an instance of such pattern is found within the source graph (i.e. Stateflow model), a *CompoundStatement* of the destination graph (i.e. C code) is created with its *if*-branch. The destination graph is an instance of the C code meta-model as depicted in Figure 3.

If someone wants to test this code generation rule for correct implementation, the main objective for test case design is initially the possible input domain for the left-hand side of the graph rewriting rule. To be precise: what are the possible graph instances the rule could match?

Domain-oriented test derivation techniques such as the Classification-tree Method (CTM) developed by Grochtmann and Grimm [10] seem to be appropriate for this purpose. The CTM (a further development of partition testing) is an approach for deriving abstract test case specifications from a given specification. The basic idea behind this method is to split up the input domain of the test object (here: the GG left-hand side) into partitions (called classifications) according to different aspects and to subdivide these partitions into equivalence classes (i.e. the leaves in the tree). These classes are then recombined and instantiated to form the test data.

In order to be able to apply the classification-tree method to the testing of a transformation rule, it must describe the input domain of the transformation rule.

To give the reader an idea of the method, a classification tree for the *IfTrValid* graph transformation (ref Figure 6) is shown in Figure 7 (upper part). The name of the test object itself forms the root node of the tree (here: *IfTrValid*). Regarding the LHS of the rule, the input domain can be partitioned into the transition source (*TrSrc*), the transition itself (*Tr*) and its destination (*TrDst*). Regarding the meta-model of Statflow (ref. Figure 1), *TrSrc* is a *TransConnector* class (atom) with possible instances such as *TransInPort*, *TransOutPort* or *Junction*. Since we expect a transition from the source to the destination node we only have to check, if the attributes *guard*, *action* and *trigger* are existent (*yes*) or not (*no*). Finally, we always expect a destination root which is of the type *Junction*.

The test cases derived are shown in the lower part of Figure 7 and are arranged in a kind of table in which each numbered row indicates a test case. A dot in a column indicates a special class (or value) for a test case. A possible instance of test case 1 is the test model (flow chart) depicted in Figure 8. It represents an executable model as an input for the code generator.

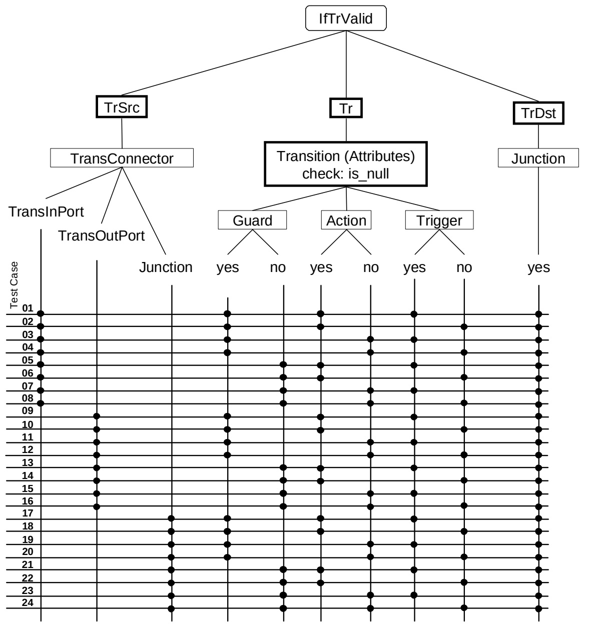


Figure 7: Classification Tree of *IfTrValid* Transformation

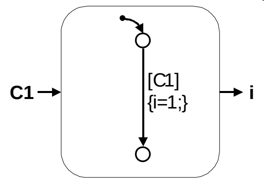


Figure 8: Instance of Test Case 1

The combination of all the classes leads to a total of 24 test cases. These include both correct (i.e. translatable) and incorrect models (i.e. those refused by the code generator). Regarding the transformation rule specified, the correct test cases could be valid (the code generator should apply the transformation) or invalid (the code generator should apply a different or no transformation). With each of the 24 test cases instantiated as an input model for the code generator, we have a suitable set for testing the *IfTrValid* transformation. However, what remains is to stimulate all these models with appropriate input data and to compare the code generated with the model, respectively. How this could be done is shown in section 5.

The classification tree depicted in Figure 7 has been created manually from the *IfTrValid* graph rewriting rule. Obviously, it would be more advantageous to generate the tree directly from the graph rewriting rule automatically. A possible way of doing this is presented in the next section.

4.3. Automatic Classification-Tree Generation

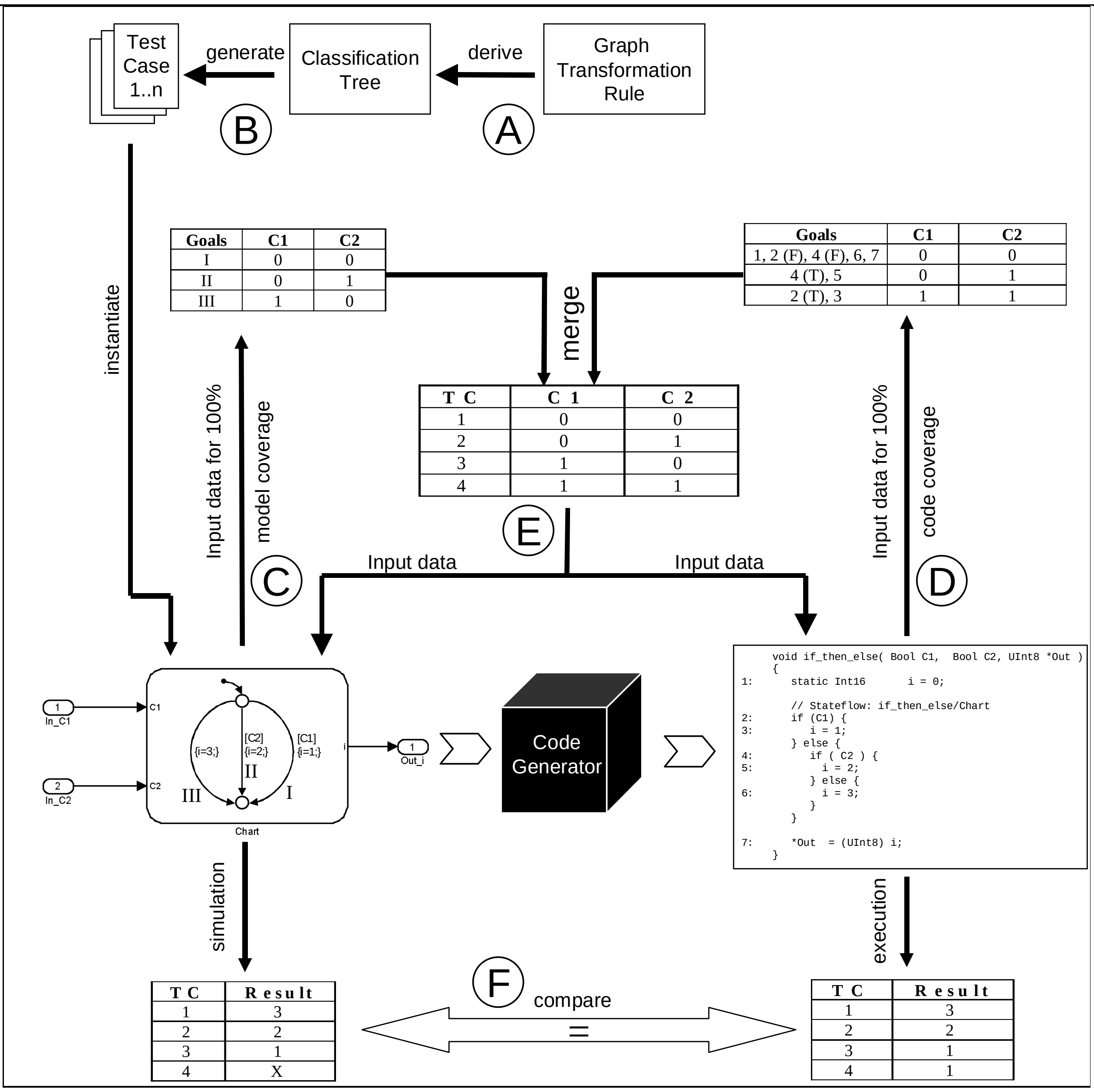
The key intuition in deriving a classification tree from a graph-rewrite based transformation specification is to formulate the domains of possible matches of the LHS of a rewrite pattern. This involves examining a pattern specification. A pattern specification in GReaT may include classes from the input meta-model, classes from the output meta-model, associations from the input and/or output meta-model, possibly cross-associations, and Boolean guards that are evaluated over the attributes of potentially matched class objects. The algorithm (1) below demonstrates the systematic derivation of a classification tree from a pattern specification:

```
1. for each Primitive Rule P in a GReaT
   specification GS
2. create a new classification tree CT,
3. for each class $C_i$ from the input
   meta-model in the rule insert a node
   $N_{ci}$ in the tree CT,
4. if class $C_i$ is an abstract class,
   then for each derived concrete class
   of $C_i$, insert a node in the sub-tree
   rooted at $N_{ci}$,
5. for each association $A_i$ between two
   classes from the input meta-model in
   the rule insert a node $N_{ai}$ in the
   tree CT, and for each node $N_{ai}$ insert
   two branches in the sub-tree rooted
   at $N_{ai}$ corresponding to the presence
   or absence of the association,
6. for each guard $G_i$ over classes from
   the input meta-model in the rule
   insert a node $N_{gi}$ in the tree CT, and
   for each node $N_{gi}$ insert two branches
   in the sub-tree rooted at $N_{gi}$
   corresponding to the guard evaluating
   to true or false.
```

This basic algorithm explores the input domain of a Primitive Rule in GReaT specifications. However, GReaT in addition to the primitive rules also has control structures with conditional constructs, which also are responsible for partitioning the input domain and hence must be used in deriving classification tree. Conditional constructs are specified in GReaT with a test-case construct. A Test-Case construct has multiple Cases each of which is specified as a LHS pattern (with no side-effects or rewrites). The classification tree for each Case construct can be derived using the basic algorithm shown earlier. The following algorithm (2) shows the systematic derivation of classification tree from a test-case construct in a GReaT specification:

```
1. for each Test in a GReaT
   specification
2. create a new classification tree CT,
3. for each Case in the Test create a
   new node N in the tree CT,
4. casting the Case as a Primitive Rule
   use the steps 3-6 of algorithm 1 to
   populate the sub-tree rooted at N.
```

We are currently in the process of implementing the algorithms shown above.



Goals	C1	C2
I	0	0
II	0	1
III	1	0

Goals	C1	C2
1, 2 (F), 4 (F), 6, 7	0	0
4 (T), 5	0	1
2 (T), 3	1	1

T C	C 1	C 2
1	0	0
2	0	1
3	1	0
4	1	1

```
      void if_then_else( Bool C1,  Bool C2, UInt8 *Out )
      {
1:       static Int16       i = 0;

         // Stateflow: if_then_else/Chart
2:       if (C1) {
3:          i = 1;
         } else {
4:          if ( C2 ) {
5:            i = 2;
            } else {
6:            i = 3;
            }
         }

7:       *Out  = (UInt8) i;
      }
```

T C	R e su lt
1	3
2	2
3	1
4	X

T C	R e su lt
1	3
2	2
3	1
4	1

Figure 9: The Code Generator Testing Approach

5. Code Generator Testing Approach

This chapter surveys the overall principles of the proposed code generator testing approach (for a more detailed explanation, the reader should refer to [13]). The main tasks are shown in Figure 9 and are described in the following:

(A) A formal specification of a transformation is created as a graph rewriting rule such as that presented in Figure 6.

(B) The formal specification is then used as a blueprint to describe the possible input domain of a transfor-mation rule with the classification-tree method (ref. section 4.2). With the test models derived from the classification tree we now have representative input models to verify the code generator's correct functionality with regard to a specific transformation.

However, before we can observe the code generator's correct behaviour, we need the right input data to stimulate these models.

(C) In order to stimulate all possible 'simulation pathways' through a given test model, we apply structural testing. Here, structural testing has the goal of automating the input data generation for suitable white box (or structural) testing criteria on model level, i.e. to find a selection of input data which achieves full structural model coverage. This can, to a large extent, be automated using tools such as Reactis [23]. After code generation has been carried out, a similar approach is followed on code level: this time structural testing is used to create a second set of input data, which guarantees complete structural coverage of the C code generated (here, we used *branch coverage* [12] as an appropriate measure). In this case, automation can be achieved with the aid of the evolutionary structure test [11]. Evolutionary structural testing uses evolutionary algorithms, an iterative search procedure based on biological evolution, to generate test data for each branch of C code generated.

(D) After input data for model and code coverage have been generated, both input data sets created are merged together.

(E) Finally, the model and the code outputs are compared. If these are identical for one and the same test datum, this is an indication that the code generator and the other tools used (e.g. compiler, linker) are working correctly. If, however, they are (substantially) different, one can conclude that this is due to an incorrect implementation of the code generator, a problem with one of the other tools involved, a faulty test model or an incomplete specification of the optimisation (incorrect graph transformation).

6. Conclusions

In this paper, a general test approach for the verification of a code generator's correct functioning based on graph transformation rules has been researched. In this approach, test case derivation is based on the formal specification of the code generators functional behavior. The description of the possible input domain of dedicated transformation rules with the classification-tree method is a suitable means of creating input models which test transformations effectively. Furthermore, a method was suggested to create such classification trees from a graph transformation rule systematically.

The main benefits of this testing approach could be summed up as follows: First of all, it shows a way how to design test cases for a code generator systematically, starting from its formal specification through to test case derivation. Second, it is possible to compare the model and the code generated from it by means of structural testing. Finally, the test process could be automated to such an extent, that the test suite could be applied to new tool releases with respect to changing requirements.

6.1. Future Work

The amount of test cases derived is (in the example presented) quite low: 24 test cases for one specific rule. With more classes in the classification tree the number of test cases could, in most cases, amount to approximately a few hundred. It would obviously be impractical to instantiate all these test cases manually as a model. The automatic generation of these models from the classification tree would be advantageous. A so-called *model generator*, which allows the automatic generation of the input models and the execution of the models within a test environment for code generators, is currently being developed. First experiences with the tool promises high automation potentials of the code generator testing process.

References

[1] The MathWorks, Simulink, Stateflow and Real-Time Workshop at http://www.martworks.com/products, Website, 2003.

[2] dSPACE, TargetLink at http://www.dspace.com, Website, 2003.

[3] Ledeczi A., Bakay A., Maroti M., Volgyesi P., Nordstrom G., Sprinkle J., Karsai G. "Composing Domain-Specific Design Environments," *Computer*, pp. 44-51, November, 2001.

[4] GME Software Manual at http://www.isis.vanderbilt.edu/Projects/gme/GMEUMan.pdf

[5] S. Neema and G. Karsai, “Embedded Control Systems Language for Distributed Processing,” ISIS Technical Report, 2004.

[6] Harel D., “Statecharts: A Visual Formalism for Complex Systems,” Science of Computer Programming, vol. 8, pp. 231-274, June 1987.

[7] Karsai G., Agrawal A., Shi F., Sprinkle J. “On the use of Graph Transformations in the Formal Specification of Computer-Based Systems,” IEEE TC-ECBS and IFIP10.1 Joint Workshop on Formal Specifications of Computer-Based Systems, p. 19-27, Huntsville, Alabama, April 9, 2003.

[8] Agrawal A., Karsai G., Shi F.: Graph Transformations on Domain-Specific Models, ISIS-03-403, November, 2003

[9] A. Boujarwah and H. Saleh, “Compiler test suite: evaluation and use in an automated environment”, *Information and Software Technology*, 1994, pp. 607-614.

[10] M. Grochtmann and K. Grimm, "Classification-trees for partition testing", *Software Testing, Verification and Reliability*, 3 (2), 1993, pp. 63-82.

[11] J. Wegener, H. Stahmer and A. Baresel, “Evolutionary Test Environment for Automatic Structural Testing”, Special Issue of Information and Software Technology, vol. 43, 2001, pp. 851-854.

[12] B. Beizer, “Software Testing Techniques”, New York: Van Nostrand Reinhold, 1983.

[13] I. Stürmer and M. Conrad, “Test Suite Design for Code Generation Tools”, *Proc. of 18th Int. Automated Software Engineering Conference*, 2003, pp. 286-290.

[14] G. C. Necula, "Translation Validation for an Optimizing Compiler", *Proceedings. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000, pp. 83-95.

[15] A. Pnueli, O. Shtrichman and M. Siegel, "Translation Validation: from SIGNAL to C", in K.G. Larsen, S. Skyum, and G. Winskel, (editors), *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming* (ICALP 1998), volume 1443 of Lecture Notes in Computer Science, Springer-Verlag, 1998, page 235-246.

[16] E. Lehmann and J. Wegener, “Test Case Design by Means of the CTE XL”, *Proc. of the 8th European International Conference on Software Testing, Analysis & Review* (EuroSTAR 2000), Kopenhagen, Denmark, Dec. 2000.

[17] ANSI/ISO FIPS-160 C Validation Suite (ACVS™), www.peren.com, WebSite, 2003.

[18] Plum Hall Validation Suite for ANSI C™, www.plumhall.com, WebSite, 2003.

[19] Associated Compiler Experts (ACE), SuperTest C&C++ Test and Validation Suite, www.ace.nl, WebSite, 2003.

[20] Ada Conformity Assessment Authority, The Ada Conformity Assessment Test Suite (ACATS), Available from: www.adaic.org, WebSite, 2003.

[21] A.S. Boujarwah and K. Saleh, “Compiler test case generation methods: a survey and assessment”, *Information and Software Technology*, 39(9), 1997, pp. 617-625.

[22] P. Purdom, “A Sentence Generation for Testing Parsers”, BIT, 1972, pp. 366-375.

[23] Reactive Systems Inc, Reactis Simulator / Tester at www.reactive-systems.com, Website 2003

[24] G. Goos and W. Zimmermann, “Verifying Compilers and ASMs”, In Abstract State Machines, LNSC Vol. 1912, 2000, pp. 177-202.