Electronic Communications of the EASST
Volume 54 (2012)

Proceedings of the 7th International Workshop on Graph Based Tools (GraBaTs 2012)

Gray Box Coverage Criteria for Testing Graph Pattern Matching

Martin Wieber, Andy Schürr

12 pages

Guest Editors: Christian Krause, Bernhard Westfechtel
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Gray Box Coverage Criteria for Testing Graph Pattern Matching

Martin Wieber[1], **Andy Schürr**[2]

[1] martin.wieber@es.tu-darmstadt.de, [2] andy.schuerr@es.tu-darmstadt.de
Technische Universität Darmstadt,
Real-Time Systems Lab,
D-64283 Merckstraße 25, Darmstadt, Germany

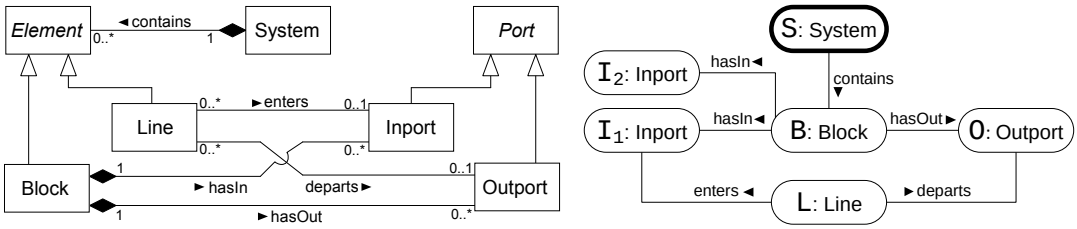
Abstract: Model transformations (MT) are a core building block of Model-Driven Engineering. The quality of MT specifications and implementations is vital to their success. The well-researched formal underpinning of graph transformation (GT) theory allows for proving quality-relevant properties and enables stringent implementations. Yet, in practice, MT implementations often depend on verification/validation techniques based on *dynamic testing*. This work presents a new *gray box coverage approach* for systematic testing of GT-based MT implementations and pattern specifications. The approach uses GT specifics and enforces systematic testing by examining variable binding and unbinding steps, thereby not making further assumptions about the underlying *pattern matching* algorithm. A family of coverage criteria is defined as temporal logic (LTL) formulae, and the effectiveness of concrete criteria in limiting the testing effort is examined by an example.

Keywords: gray box testing, adequacy criterion, coverage, graph pattern matching

1 Introduction

Model transformations (MT) are a core concept of Model-Driven Engineering (MDE), and like any other software, MT programs should be examined thoroughly w.r.t. quality and correctness. MTs are usually written in specialized languages, often rule-based, following a declarative approach (cf. [CH06]). Languages and tools (code generators, interpreters) are still frequently under construction, the number of users is limited, and esp. tools are commonly considered immature to some extent. Formal underpinnings like graph transformation theory [EEPT06] temper the negative effects of this situation, but application of formal methods to concrete realizations is often tedious and infeasible in practice due to the imposed effort and inherent complexity.

Testing seems to be a promising solution to both the problem of ensuring the quality of MT specifications as well as of assessing the quality of MT implementations. In contrast to general-purpose programming languages, for which implementation-based (*white* or *gray-box*) testing techniques exist, there is no such well-established equivalent for typical MT languages. A fact that is partly founded on the challenges inherent to model transformation testing [BGF+10]. In the rest of the paper, we restrict ourselves to the important class of graph pattern based MT languages. For this class of MT languages we introduce a novel concept of *coverage* which provides an *adequacy criterion* that is independent of the source code of a pattern matching (PM) implementation but still utilizes knowledge of the PM process. It limits the testing effort by defining min. level of coverage that a test suite should satisfy to be considered sufficient (construction of



(a) Simple metamodel (Simulink) (b) Graph pattern (abstract syntax)

Figure 1: Metamodel and graph pattern

tests – in form of concrete input models – is beyond the scope of this work). Several application scenarios are conceivable: (1) testing a concrete PM *implementation* based on the pattern (black-box) (2) testing an immediately executable MT *specification* (white-box) (3) rough figure of merit for existing test suite's completeness. For the sake of clarity, compare a graph transformation (GT) tool to the Java compiler: MT specification plus metamodel correspond to source code plus Java types. Testing a Java method requires test inputs that exercise the method's code sufficiently (w.r.t. statement coverage etc.). The same is true for good graph pattern tests exercising a PM implementation, except that other and new coverage notions are required.

The remainder of the paper is organized as follows: Section 2 introduces terminology and concepts based on an example. Section 3 describes the PM process (esp. its operational semantics) and necessary testing infrastructure. Section 4 presents our novel coverage notion and core contribution. Section 5 reports on first results drawn from an application of the concepts to the given example based on a prototype framework implementation and manually derived test suites. Section 6 provides related work, and in Section 7 we draw conclusions and outline future work.

2 Terminology and Running Example

Metamodels and Models A *metamodel* (MM) reflects the core concepts of the modeled domain. It features a graph-like structure and defines the (abstract) syntax of corresponding (instance) *models* (M). The MM comprises *classes* as its nodes and *associations* as its edges, the former define valid types of the models, the latter define the structuring/relations among them.

Figure 1a shows an exemplary metamodel. It represents a minimalist description of Simulink (a tool often used in engineering). A *System* contains *Blocks* and *Lines*, *Blocks* comprise *Outports* and *Inports*. *Blocks* can further be interconnected via *Lines* originating and ending at *Ports*.

Model- and Graph Transformations A MT takes models, conforming to one input MM, and transforms them to models, conforming to another MM. Here, we do not distinguish between the in-place modification of models and the translation of models from one language to another. The specifics of the considered MT are defined by *graph transformation* theory (c.f. [EEPT06]), where models are represented by typed graphs and transformations are formalized by graph rewriting rules. Several tools implementing a GT approach are available, although the formalizations/implementations can differ (cf. [Tae04, JBK10, LV02] for examples).

Graph Patterns The aforementioned tools all specify a MT via *declarative*, *rule-based* descriptions of *graph rewriting step*. They all use some sort of *graph pattern* (GP) notion, which is referred to as the *left-hand side* (LHS) of a GT *rule*. During the processing of one *rule*, the transformation engine tries to find a *match* for the LHS graph in the model (or host graph), where a match is a sub-graph of the host graph that is isomorphic to the LHS. This process is commonly implemented based on some elaborated search algorithm, instead of relying on a trivial "generate and test"-approach. The former usually combines depth-first-search and backtracking, one exception being RETE-like pattern matching algorithms which compute (sub)matches in parallel, sacrificing memory consumption for run-time performance [BÖR+08]. In the following we will restrict ourselves to depth-first-search algorithms (for more details see below).

Figure 1b depicts an exemplary GP. This GP comprises six *variables*: S, I_1, I_2, B, O, L of the respective types *State*, *Inport* and so forth. The specially outlined node S is initially *bound*, meaning that is initially set to some fixed node. During PM, the engine tries to assign *objects* (of compatible types; links constrain the number of valid variable bindings further) to the remaining variables. The process of assigning a concrete value to one of the variables is called *binding operation/step*. To reassign a new value to the variable, this binding step needs to be reverted first, resulting in an unbound (intermediate) state. We refer to this as *unbinding step*.

Pattern Matching The goal of PM is to find a match of the LHS of the GP in the host graph. As mentioned before, PM is usually done stepwise by a form of local search: a partial match (defined by a set of bound variables) is extended incrementally by binding additional variables, starting with a set of initially bound variables (in our example: $\{S\}$). If all variables can eventually be bound without violating constraints, a match will be found. If, during the process, the next to-be-assigned variable can not be bound, already bound variables get unbound and reassigned, when possible. This procedure is repeated until either a complete match can be determined or all possible permutations have been checked and discarded. In case a match has been found, it is replaced by a copy of the *right-hand side* (RHS) of the GT rule (rewrite step). This modifying step is out of scope and neglected here, since the focus lies solely on testing the PM.

In addition to structural and type constraints, and depending on the underlying graph model, the pattern might further be constrained by *attribute conditions*. Another common extension to basic GP is the concept of *negative application conditions* (NAC). NACs are specially marked elements of a pattern that, if their identification is possible, prevent a match from being valid. Other possible extensions to GP include *optional elements* (nodes or edges that can optionally be included in a match) and *set-nodes* (results in matches of variable and unknown size; comparable to the *-operator in regular expressions). In the remainder, only basic patterns are considered; NACs, optional elements, and set-nodes are omitted and left for future work.

3 Operational Semantics and Tracing

In this chapter we derive an abstract description of the run-time behavior of typical pattern matchers. This forms a (MT) language-neutral basis for the definition our coverage concepts in the following sections.

3.1 Operational Semantics of Pattern Matching

The process of determining a valid match for a given pattern in an input graph is crucial for the entire model transformation. Our experience in developing code-generator-based pattern matching engines shows that implementing/optimizing this task is complex and error-prone.

As already mentioned, a pattern is usually searched for iteratively by partial match extension and relying on backtracking. The overall process can be visualized in form of a decision tree as in Figure 2, where each decision/branching corresponds to an *atomic operation* like "extend partial match by navigating along link `l` binding previously free variable X", "check existence of link `l` between two bound nodes X, Y" or "ensure variable X and variable Y are not bound to the same node". The PM process is essentially characterized by such a sequence of operations (of types *extend* and *check*), and one (not necessarily unique) concrete ordering of such operations is called a *search plan* (SP). The derivation of all possible, viable SPs for a given pattern, in order to determine the optimal one, is often computationally infeasible, and optimality also depends partially on the model (cf. [VDWS12] for details on this issue).

Failed *check* operations eventually lead to backtracking steps and indirectly result either in variable reassignments or incomplete matches. The pattern matching process thus is characterized solely by a sequence of binding and unbinding steps, whereby, by definition, the unbinding steps can only occur in *inverse order* of the corresponding binding steps.

The motivation for this abstraction lies in the possibility to describe the PM process in as much detail as possible without depending on a concrete implementation (e.g. in the sense of source-code). When using a compiled, generator-based approach, where code is derived from the pattern specification, one might design test suites based on that code using classical techniques. Unfortunately, even small changes to the pattern or a non-deterministic code-generator would result in different code, deprecating existing tests. The situation even worsens when using an interpreter-based approaches. Testing the PM process based on the source-code of an interpreter seems futile, because the exercised fraction of code can not be expected to change much with different SPs or models as the authors of [HLG+12] indicate.

3.1.1 Generated Search Plans

We generated several code variants with our customized *CodeGen2* fork (from *Fujaba*), resulting in equivalent but different SPs. We considered two SPs, termed SP_1 and SP_2, for further investigation. Figure 2 depicts them. In the figure, each node represents a state, characterized by a mapping of the n ($n = 6$) variables of the pattern (S, I_1, ...) to the domain $\{0,1\}^n$, where a value of 0 indicates that the resp. variable is not bound, and a value of 1 indicates the opposite. Transitions are labeled with the operations that are either used to structurally extend the partial match or to check pattern constraints. We use grey highlighting to indicate no change in the binding vector. From our previous work [VDWS12], we borrow the notions of *adornments* and *masks* to express the application conditions of the operations, distinguish check from extend operations, and indicate navigation direction. Basically speaking, the adornments define which variables should be bound 'B' or unbound 'F'; masks define an ordering of all variables and indicate which variables are non-restricted '$*$' for applicability.

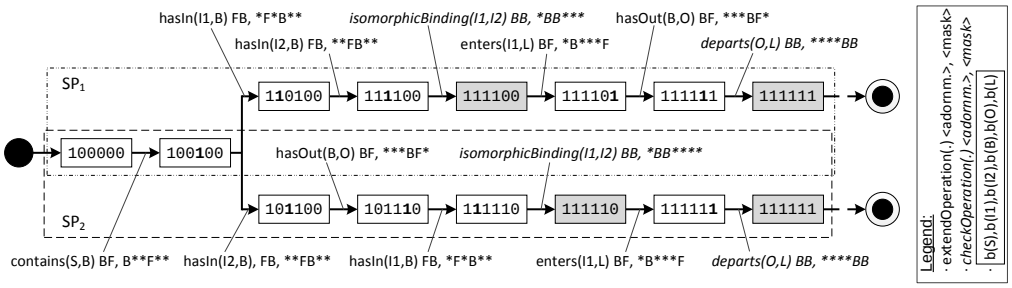


Figure 2: Search plans (see Equation 1 for the definition of function b)

3.2 Tracing the Pattern Matching Process

Observability is one prerequisite for *testability*, as Binder states in [Bin00]. So, when examining the dynamic behavior of the pattern matcher, one needs information on the engine during the algorithm's running. One can imagine that (almost) any pattern matching implementation can be easily extended or *instrumented* to output traces of states comprising the binding status of the pattern's variables. Consequently, we restrict ourselves to the information obtained through simple (offline) *tracing*, where status information is collected/stored during run-time. In our case, we instrumented the PM code manually, but adapting our generator templates is considered to be a straight forward task and left for future work.

4 Coverage Metrics

We continue by defining the gray box coverage metrics and motivate their usefulness to PM testing. As far as our experience goes, implementation bugs are likely to manifest in incorrect backtracking behavior during the search (e.g. premature, too late, etc.). On the other hand, bugs that only manifest in "unusual" situations are more likely to be overlooked in comparison to others. Consequently, for thorough testing, one needs to ensure that the test suite exercises the search plan in a *systematic* and *sufficient* manner. Our coverage metrics define minimal requirements so that when met, the test suite can be reasonably considered sufficient w.r.t. the examined patterns. Consequently, the metrics serve the same purpose as (code) coverage in traditional testing, namely "[...] to mitigate unavoidable blind spots [...]" [Bin00] during test suite design, and to help the tester in performing the task of *path sensitization*[1]. If one tries to test the *general functioning and fitness* of a PM engine, regardless of any concrete MT, one needs to provide metamodels plus patterns (together they form the test cases) that are "complex enough" to cover all supported language features – a task that is out of scope here. This suggests, that the presented method is MT specific.

Coverage Items The basic idea of our testing approach is to stimulate the PM engine in such a way that certain combinations of variable binding and unbinding steps actually do occur. This is achieved either by extending a partial match or by performing a backtracking step. One single

[1] "Process of determining [...] variable values that will cause a particular path to be taken". ([Bin00], p. 399)

sequence of such steps represents a *coverage item*. If a trace log indicates that a test run led to the occurrence of the demanded sequence, we say that the test covers the sequence and thus the coverage item. To be able to precisely control the testing effort, we developed different metrics and ensured them to form a hierarchy, where more elaborated metrics *subsume* more basic ones, so that coverage of the subsuming criterion implies coverage of the subsumed one.

In the following, *Linear-time Temporal Logic*[2] (LTL) is used to define the cov. items. Some basic definitions are required first (x refers to one variable, X to the set of variables defined by the pattern, $\mathscr{X} := (X, <)$ is a strict totally ordered set, and $\mathscr{X}' := \mathscr{X} \setminus \{\text{initially bound variables}\}$. Additionally, for this example $\mathscr{X} = (\{S, I_1, I_2, B, O, L\}, \text{order of appearance})$):

$$\mathrm{b} : \mathscr{X} \to \{0,1\}, \quad \mathrm{b}(x) := \begin{cases} 0, & \text{if variable } x \text{ is unbound} \\ 1, & \text{if variable } x \text{ is bound} \end{cases} \tag{1}$$

$$m \in \{-1, ?, +1\}, \quad m \triangleq \begin{cases} -1, & \text{matching failed} \\ ?, & \text{match still incomplete} \\ +1, & \text{match found} \end{cases} \tag{2}$$

Equation 1 defines a function that assigns binding information to variables, and in Equation 2 m is introduced, which represents knowledge about the outcome of the PM run. Figures 2 and 3 use these concepts. Both definitions are used to formulate temporal predicates, whereby the pattern, SP, and traces are kept implicitly fixed. In this regard, actual values depend on an implicit notion of time/state (corresponding indices/arguments are omitted here, as in [HR04]). For example, to state that a run should result in a complete match, one would define a predicate Φ^+_ω such as the one in Equation 3 using LTL-operators. It states that *eventually* ($\mathbf{F}(.)$) variable m should remain '+1' "forever" or *globally* ($\mathbf{G}(.)$). *Until* this happens ($(.)\,\mathbf{U}\,(.)$) variable m should have the value '?', indicating an incomplete match. Equation 4 considers the opposite outcome.

$$\Phi^+_\omega := (m = ?)\,\mathbf{U}\,(\mathbf{G}(m = +1)) \tag{3}$$

$$\Phi^-_\omega := (m = ?)\,\mathbf{U}\,(\mathbf{G}(m = -1)) \tag{4}$$

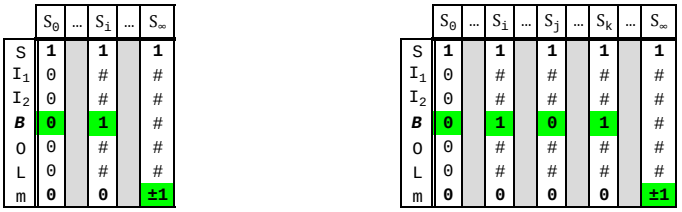
4.1 PMC_0

We now define the simplest of our coverage criteria, called PMC_0, whereby PMC stands for *pattern matching coverage*. It comprises the two conditions PMC^+_0 and PMC^-_0 which need to be fulfilled separately. Informally, the first states that for every variable x there has to be (at least) one test case that eventually binds x to some node in the model (of compatible type) and results in a complete match (*positive test*). The second one states the same but excludes a complete match in the end (*negative test*). More formally, the following proposition has to hold:

$$\mathrm{PMC}_0(\mathscr{T}) := (\mathrm{PMC}^+_0(\mathscr{T}) \wedge \mathrm{PMC}^-_0(\mathscr{T})) \tag{5}$$

whereby $\mathscr{T}$ denotes a set of captured traces, and one single trace (= sequence of trace entries, whereby an entry is a bit vector, cf. columns in Figure 3) is referred to as t. We also define a

[2] For a general introduction see, e.g., [HR04].



(a) PMC_0 for B (b) PMC_1 resp. $\mathrm{PMC}_{1,1}$ for B

Figure 3: Coverage metrics examples ("#" symbolizes "don't care")

theoretical *extended trace* t^∞ being an infinite sequence that repeats the last item of t in positive direction along the time axis.
The aforementioned proposition holds iff the following is true:

$$(\forall x \in \mathscr{X}' \; \exists t \in \mathscr{T} : \mathrm{PMC}_0^+(t,x)) \wedge (\forall x \in \mathscr{X}' \; \exists t \in \mathscr{T} : \mathrm{PMC}_0^-(t,x)) \tag{6}$$

$$\left(\mathrm{PMC}_0^\pm(t,x)\right) \Leftrightarrow \left(\Phi_0^\pm(x) \text{ holds for the extended trace } t^\infty\right) \tag{7}$$

$$\Phi_0^\pm(x) := \overbrace{\Phi_\alpha(x)}^{\text{initial configuration and global constraints}} \wedge \mathbf{F}(\mathrm{b}(x) = 1) \wedge \overbrace{\Phi_\omega^\pm}^{\text{final steady state}} \tag{8}$$

$$\Phi_\alpha(x) := (b(x) = 0) \wedge (m = ?) \tag{9}$$

Equation 8 and Equation 9 state that x has to be initially unbound and has to become eventually bound. At the end, a steady state has to be reached where either a match has been found or not.

Figure 3a visualizes the concept for the case where x is set to B. A minimal test suite meeting the requirements of PMC_0 is presented later on in Section 5. Note that the initial, one intermediate, and the final state are restricted w.r.t. variable states. Any number of arbitrary intermediate states (indicated by the dots in the resp. fig.) are allowed to occur. Also note, that no distinct matching order, as predetermined by a concrete SP, is required for the definitions. Note further that one can expect the variables to be bound in the final state, at least in the case where there is a complete match. Nevertheless, the presented formulation is open to adaption (potential extensions could supports optional nodes etc.). Summarized, PMC_0 only ensures a very rudimentary coverage, since it does not necessarily imply or require any form of complex branching during the PM process, but introduces the concepts and forms the basis for refinements.

4.2 PMC_1

The first extension is PMC_1. It *subsumes* PMC_0 and extends PMC_0 in that distinct variables x need to consecutively take on the bound, the unbound, and the bound state again, before reaching the steady state with either a match (PMC_1^+) or no match (PMC_1^-) as outcome (intermediate states are permitted as well). This ensures a more thorough testing due to additionally required binding and unbinding steps and is likely to require larger (element count) and more complex test models. When met, it guarantees that for each variable at least two (not necessarily distinct) options were evaluated within one captured test run. Figure 3b underlines the concept. For formalization, we can reuse Eqs. 5, 6, 7, and 9 directly (by changing index 0 to 1). Equation 10

is the adapted version of Equation 8. Compared to the latter case, there are additional nested statements requiring variable x to be bound first (outermost **F** in the disjunction), afterwards unbound (intermediate **F**), and ultimately bound thereafter (innermost **F**).

$$\Phi_1^{\pm} := \Phi_0^{\pm} \wedge \left(\mathbf{G}(\mathrm{b}(x) = 0) \vee \mathbf{F} \left((\mathrm{b}(x) = 1) \wedge \mathbf{F} \left((\mathrm{b}(x) = 0) \wedge \mathbf{F} \left(\mathbf{G}(\mathrm{b}(x) = 1) \right) \right) \right) \right) \quad (10)$$

Figure 3b and the previous remarks suggest that one could also demand more than one "cycle" of unbind/bind operations for a variable. This motivates the generalization of PMC_1, or $\mathrm{PMC}_{1,1}$ from now on (the second index stands for *one* cycle), to the case of $\mathrm{PMC}_{1,n}$ requiring n, $\{n \in \mathbb{N} \mid n \geq 1\}$, of such cycles. A higher value of n will likely lead to increased test model sizes, but this does not necessarily imply coverage of subsuming metrics as explained later.

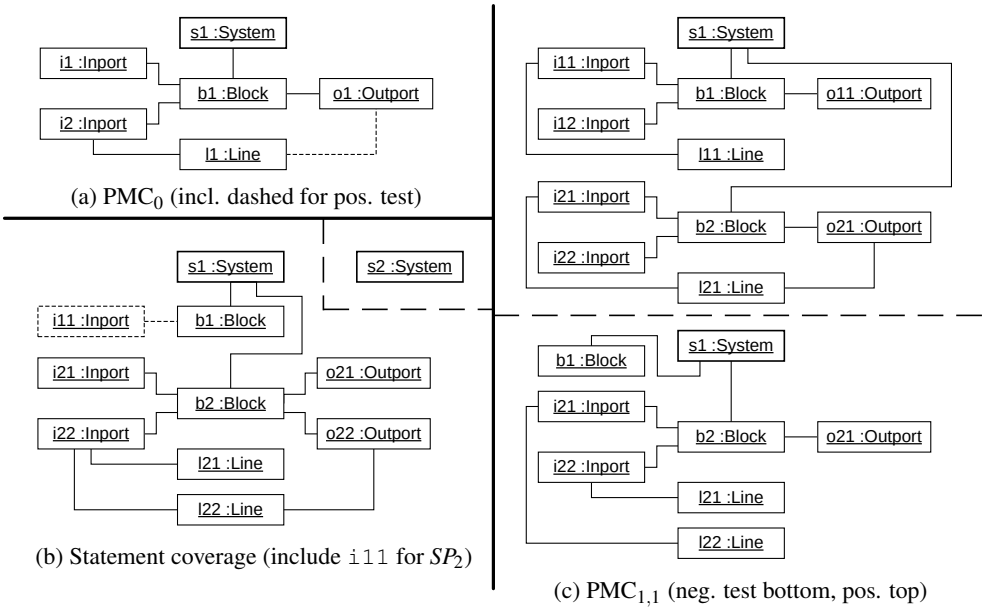
Right now, there is ongoing work spent on the extension of the concept to $\mathrm{PMC}_{2,n}$ (pairs of variables), $\mathrm{PMC}_{3,n}$ (triples of variables) and so forth by considering variable tuples instead of single variables. This would help in testing the interplay of binding/unbinding steps for different variables. Further details are omitted here, due to space limitations.

5 Application

Now we come to the discussion of preliminary results obtained during evaluation of our protoype coverage framework. We modeled the pattern of Figure 1b in our *eMoflon* [ALPS11] tool suite and generated two Java realizations featuring a distinct SP from it (cf. Figure 2). Tracing commands were added manually to the code, and the tracing information was collected and analyzed by a prototypic tool.

Basic Test Suites We used the setup to evaluate whether it is possible to construct input data that leads to complete coverage for certain metrics, given the previously fixed SPs. With prior knowledge of an actual SP, it turned out to require not much of an effort to find small test models meeting the requirements. Figure 4 depicts test suites for PMC_0 and $\mathrm{PMC}_{1,1}$ respectively. The results indicate that one can effectively limit oneself to a pair of input models to ensure coverage of one (or several, in case of subsumption) metric for all relevant pattern variables at the same time, although this seems not very favorable when thinking of maintainability and traceability. Nevertheless, metamodels with additional constraints might restrict input models (e.g. by upper multiplicity bounds) so that one can not achieve the required coverage level with a minimum of two models. Chained patterns (as in programmed GT) would complicate things considerably.

The results also support the working assumption that our stronger metric leads to more complex test data in comparison to the basic one. Additionally, it should be noted that there exists a distinction between *relevant* input complexity and input complexity *without influence* on the search process. For example, when relying only on the metamodel to construct input models, one could construct very diverse looking models with (superficial) complexity (e.g. high element count), which is all but relevant to thorough testing the pattern. Think, for example, of "early decisions" during the PM process, which might skip further examination of the alleged complex parts. If the search plan changes, previously "irrelevant" complexity could turn out beneficial, though, and the test data might lead to sufficient coverage. Our coverage notion enables us to construct tests with relevant complexity systematically, without relying on chance.



(a) PMC_0 (incl. dashed for pos. test)

(b) Statement coverage (include `i11` for SP_2)

(c) $PMC_{1,1}$ (neg. test bottom, pos. top)

Figure 4: Test models for different coverage criteria

Comparison with Code Coverage Deriving imperative code from a MT specification generally enables us to compare our coverage concept to well-accepted code coverage approaches. We used *Cobertura*[3], which supports *basic block coverage* and *branch coverage*, to measure code-based coverage figures for the example.

Results shows that even the test data for PMC_0 already lead to complete statement coverage of the code. We presume that this is always the case, but this claim needs to be supported by further investigation (ongoing work). Also, there is no such thing as a *canonical* implementation, and it would be interesting to compare code coverage measures for functionally equivalent (even down to the search plan) code representations which are sufficiently different.

We also investigated whether code statement coverage implies coverage in our sense. Think of the situation, where one tries to test the code sections that process a complete match. One needs an input model comprising a complete match as test case, which shall be the only test in an initial test suite. We could extend the model to cover as many additional statements as possible, but due to the code fraction that processes incomplete matches as ultimate result, it is not possible to cover all statements with only just one test. A second, rather small model without complete matches is required. In general, this model can be expected to be rather small so that it does not ensure PMC_0 coverage (cf. negative test). For the running example, Figure 4b depicts a test suite that achieves statement coverage but fails in achieving PMC_0 coverage. For our concrete example, statement coverage is implied by PMC_0 coverage, so here even our weakest coverage criterion subsumes statement coverage (unnecessary/dead code is neglected). This is obviously not a prove, though, and more experiments are required to investigate the interplay with code-based criteria.

[3] http://cobertura.sourceforge.net/

6 Related Work

Several groups published results related to MT testing. A wide range of articles examine test data generation, but this only partially relates to test adequacy evaluation. By far the most coverage related work involves specification-based black box testing. A good example is [FBMT09] by Fleurey et al., where they motivate their decision for a black box approach with its independence from the underlying transformation language. They introduce the notions of *class*, *attribute* and *association coverage*. Bauer et al. extend these concepts in [BKE11] by introducing *feature* and *transformation contract coverage*. Their overall goal is to assess and optimize the test suite quality for model transformation chains. Both works do not anticipate pattern definition and/or implementation related bugs directly.

In [GV11], Gogolla et al. present a testing approach based on OCL constraints on source and target metamodel called *Tracts* (MT contracts). They describe how their USE tool can check contract adherence on the outcome and how corresponding test models can be derived. Cabot et al. describe a similar approach in [CCGL10] where they use OCL in conjunction with their UMLtoCSP tool whereby focusing on the analysis of MT properties rather than on testing. Both works do not examine test data adequacy aspects and esp. no coverage notion.

Darabos et al. take a different view on testing in [DPV08]. They present a *fault model*, which condenses their knowledge of typical programmer faults during implementation of PM engines. They use a hardware verification technique called *Boolean difference method* to derive test data which is sensitive to such faults. In some sense, the faults captured by their model could be interpreted as coverage items. A drawback of this approach is that it depends on the quality of the fault model. Unconsidered faults are likely to be overlooked.

Hildebrandt et al. use TGG rules to derive pairs of input models and expected output models for testing a TGG implementation in [HLG+12]. Although this represents an elegant approach to test oracle construction, the approach can only be applied if a TGG specification exists. Unfortunately, TGGs are not (yet) as expressive as most MT languages. The authors also evaluate code coverage of their interpreter-based MT engine with different derived test suites, and state that it remains virtually unchanged.

In [KA07], Küster et al. report on what they call a white box approach to validation. In addition to the input metamodel they use the "design and implementation of the model transformation" by building on so-called *meta model templates* plus constraints. A basic fault model is provided and the *interplay of rules* is examined. Other white box approaches are presented by Ciancone et al. in [CFM10], where unit testing of MT specified in QVT Operational is applied, and in [MP09] by McQuillan et al., where standard code coverage metrics are used for ATL specifications. Such white box approaches have the draw-back of being language specific. They also require more insight in the machine-runnable implementation, which contravenes the declarative paradigm.

Steel et al. describe the test driven development of the Tefkat engine in [SL04]. Geiger et al. derive JUnit tests from pattern specifications in [GZ05] which are used to test transformation code conformance. A different approach is presented by Baldan et al. [BKS04] where GT specifications are used to model the behavior of code generators, and tests are derived from those specifications. All those works focus on testing code generators rather than on testing MTs, the former task being more comparable to compiler testing than to program testing.

There also exist quite some work on the combination of temporal logic and GT, mostly with

a focus on analyzing transformation properties. The work of Baresi et al. [BRRS08] is one example where LTL is used, but there exist several other texts on that topic.

7 Conclusion

Testing model transformations is vital for their future success in practice. Coverage concepts are an important aspect of a full-fledged testing process, but there is no silver bullet to this problem as the number of approaches show.

We introduced a new coverage concept for gray box testing of GT and esp. graph pattern matching, which is an error-prone sub-task. Our coverage criteria are based on the operational semantics defined by search plans (and traces), and we motivate its usefulness with first experimental results and a comparison of its properties and performance to that of statement coverage.

Future work comprises a comparison with more elaborated code-based coverage criteria (like branch or path coverage), and an extension of the approach to treat optional, set and negative nodes in patterns. On the implementation side remains the task of developing an integrated testing framework including the new coverage concept plus other functionality like test generation and oracle functions. On theory side, other/additional temporal constraints are conceivable.

Bibliography

[ALPS11] A. Anjorin, M. Lauder, S. Patzina, A. Schürr. eMoflon: Leveraging EMF and Professional CASE Tools. In *INFORMATIK 2011*. LNI 192, p. 281. GI, 2011.

[BGF+10] B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon, J.-M. Mottu. Barriers to Systematic Model Transformation Testing. *Commun. ACM* 53(6):139–143, 2010.

[Bin00] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. The Addison-Wesley Object Technology Series. Addison-Wesley, 2000.

[BKE11] E. Bauer, J. Küster, G. Engels. Test Suite Quality for Model Transformation Chains. In *Objects, Models, Components, Patterns*. LNCS 6705, pp. 3–19. Springer, 2011.

[BKS04] P. Baldan, B. König, I. Stürmer. Generating Test Cases for Code Generators by Unfolding Graph Transformation Systems. In *Proc ICGT '04*. LNCS 3256, pp. 194–209. Springer, 2004.

[BÖR+08] G. Bergmann, A. Ökrös, I. Ráth, D. Varró, G. Varró. Incremental Pattern Matching in the Viatra Model Transformation System. In *Proc GRaMoT '08*. Pp. 25–32. ACM, 2008.

[BRRS08] L. Baresi, V. Rafe, A. T. Rahmani, P. Spoletini. An Efficient Solution for Model Checking Graph Transformation Systems. *ENTCS* 213(1):3–21, 2008. Elsevier.

[CCGL10] J. Cabot, R. Claris, E. Guerra, J. de Lara. A UML/OCL framework for the analysis of graph transformation rules. *SoSyM* 9:335–357, 2010. Springer.

[CFM10] A. Ciancone, A. Filieri, R. Mirandola. MANTra: Towards Model Transformation Testing. In *Proc QUATIC '10*. Pp. 97–105. 2010. IEEE.

[CH06] K. Czarnecki, S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.* 45(3):621–645, 2006.

[DPV08] A. Darabos, A. Pataricza, D. Varró. Towards Testing the Implementation of Graph Transformations. *ENTCS* 211:75–85, 2008. Elsevier.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[FBMT09] F. Fleurey, B. Baudry, P.-A. Muller, Y. Traon. Qualifying input test data for model transformations. *SoSyM* 8:185–203, 2009.

[GV11] M. Gogolla, A. Vallecillo. Tractable Model Transformation Testing. In *Proc ECMFA '11*. LNCS 6698, pp. 221–235. Springer, 2011.

[GZ05] L. Geiger, A. Zündorf. Story Driven Testing - SDT. In *Proc SCESM '05*. Pp. 1–6. ACM, 2005.

[HLG+12] S. Hildebrandt, L. Lambers, H. Giese, D. Petrick, I. Richter. Automatic Conformance Testing of Optimized Triple Graph Grammar Implementations. In *Proc AGTIVE '11*. LNCS 7233. Springer, 2012. To appear.

[HR04] M. Huth, M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.

[JBK10] E. Jakumeit, S. Buchwald, M. Kroll. GrGen.NET. *Int. J. on STTT* 12:263–271, 2010. Springer.

[KA07] J. Küster, M. Abd-El-Razik. Validation of Model Transformations - First Experiences Using a White Box Approach. In *Models in Software Engineering*. LNCS 4364, pp. 193–204. Springer, 2007.

[LV02] J. de Lara, H. Vangheluwe. AToM^3: A Tool for Multi-formalism and Metamodelling. In *Proc FASE '02*. LNCS 2306, pp. 174–188. Springer, 2002.

[MP09] J. McQuillan, J. Power. White-Box Coverage Criteria for Model Transformations. In *(prel.) Proc MtATL '09*. Pp. 63–77. 2009. AtlanMod INRIA & EMN.

[SL04] J. Steel, M. Lawley. Model-Based Test Driven Development of the Tefkat Model-Transformation Engine. In *Proc ISSRE '04*. Pp. 151–160. 2004. IEEE.

[Tae04] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *Proc AGTIVE '04*. LNCS 3062, pp. 446–453. Springer, 2004.

[VDWS12] G. Varró, F. Deckwerth, M. Wieber, A. Schürr. An Algorithm for Generating Model-Sensitive Search Plans for EMF Models. In *Proc ICMT '12*. LNCS 7307, pp. 224–239. Springer, 2012.