Model-driven Engineering for Dynamic Data Structures*

Jan H. Boockmann, Kerstin Jacob, Gerald Lüttgen
Software Technologies Research Group, University of Bamberg, Germany
{firstname.lastname}@swt-bamberg.de

Abstract

Model-driven engineering (MDE) has become a key technology in such diverse fields as signal processing, control engineering and software engineering. Our research has adopted the MDE paradigm for the analysis of complex software involving dynamic data structures, e.g., of device driver managers that employ custom list structures. Here, the central model studied by us is logic predicates that describe data structure shapes.

This paper highlights aspects of our research on how shape predicates can support a range of activities: automated code generation for defensive programming, visualization for program comprehension and test case generation and formal verification for quality assurance. We discuss the commonalities and differences to the MDE of control-intensive systems and outline how our test case generation approach may be adapted to complex object-oriented software.

Keywords: model-driven engineering, shape predicate, program comprehension, test case generation

1 Introduction

Model-driven engineering (MDE) [10] has vastly broadened the role of software models from documentation and analysis artifacts to drivers for automation. It has found its way into many application areas and is nowadays supported by a variety of academic and commercial tools; consider, e.g., signal processing with its block diagrams and NI's LabView tool[1], control engineering with its added state machines and Mathwork's Simulink/Stateflow[2], and software engineering with its class diagrams and Yatta's UML Lab[3]. Notable here is the use of visual languages and the support of a variety of development and quality assurance activities, especially code and test case generation, simulation, and verification by model checking (see Figure 1, items labelled (1) and (3)).

Our research highlighted in Section 2 adopts the MDE paradigm for the analysis of complex software systems with *dynamic data structures*. Such pointer-based structures are typically encapsulated in programming libraries or run-time environments. One particular focus of us is system-level software that uses highly customized, nested dynamic data structures, e.g., for device driver[4] and file system management, in order to achieve highly performant solutions. Understanding and verifying such software is crucial for developing dependable, reliable, and secure systems. In our domain, the models of interest are not visual but logical, namely *shape predicates* phrased in, e.g., *separation logic* [15], which characterize complex pointer structures such as doubly-linked lists that contain lists or topological sort trees with lists that run through tree nodes.

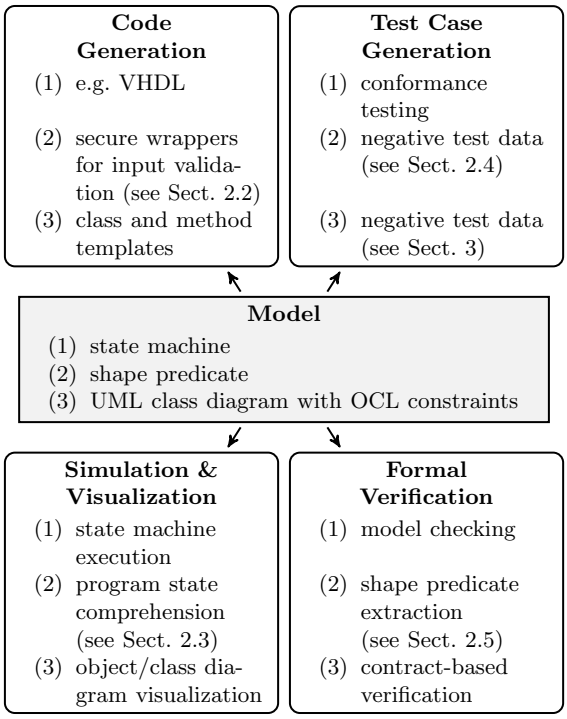


Figure 1: Application context of model-driven engineering in the context of (1) state machines, (2) shape predicates, and (3) UML class diagrams with OCL constraints.

Similar to classical MDE models, shape predicates can be used for various activities (see Figure 1, items labelled (2)): automatic generation of wrapper code for pointer input validation at trust boundaries [17], test case generation for structural integrity checks of the dynamic data structures behind pointers [1], visualization of evolving structures within memory graphs [4, 5], and

*This research is partially supported by the German Research Foundation (DFG) under project DSI2 (grant no. LU 1748/4-2).

[1] http://ni.com/de-de/shop/labview/

[2] http://de.mathworks.com/products/stateflow.html

[3] http://uml-lab.com

[4] http://libusb.info/

formal program verification with tools such as KeY[5]. While shape predicates may be provided by engineers during software design, one focus of our research is *program comprehension*, e.g., in the context of software reengineering, where we extract shape predicates from program executions [3] and use our visualization tools for debugging complex program states [5].

We believe that our MDE-inspired techniques developed for dynamic data structures have also something to offer the MDE of *object-oriented software* which also require structural integrity checks. Typically, object-oriented structures are modeled, e.g., by UML class diagrams with OCL constraints [7] or, alternatively, in Alloy [16]. While most MDE tools focus on code generation (e.g., UML Lab [3]), visualization (e.g., USE [6]), and verification (e.g., KeY [5]), the activity of test generation is, in our opinion, not yet supported to full extent. We outline in Section 3 how our test case generation technique based on shape predicates may be lifted to object-oriented software and benefit the validation of object structures passed to methods.

2 MDE for Programs with Dynamic Data Structures

Dynamic data structures are essential for storing, retrieving, and operating on information, which makes their correct functioning highly relevant for system correctness. Our work uses shape predicates [8] as models of dynamic data structures for various automation purposes: wrapper generation for input validation, visualization for program comprehension, test case generation, and formal verification for quality assurance. In the following, we discuss our different approaches for automation based on shape predicate models, as well as the analogies and differences to the traditional MDE of control-intensive systems.

2.1 Modeling Dynamic Data Structures

Dynamic data structures can be modeled, e.g., by shape predicates [8], to reason about their structural properties exhibited at runtime. For example, a binary tree, where each node has a left and a right reference, has to fulfil three structural constraints: child nodes must be distinct, cycles must not exist, and each node must have at most one parent node.

Separation logic [15] has been used successfully in research and industry projects as a modeling language to concisely define dynamic data structures. The logic simplifies the modeling of program heaps via its separating conjunction operator "$*$", which divides a heap into disjoint segments. This allows one to elegantly express inductive data types that characterize dynamic data structures, including structures with backlinks, e.g., doubly-linked lists.

[5] http://key-project.org

[6] www.db.informatik.uni-bremen.de/projects/USE-2.3.1/

The following separation logic predicate bt encodes the linkage pattern exhibited by a binary tree:

$$\begin{aligned} \mathrm{bt}(n) \quad &\stackrel{\text{def}}{=} \quad (\mathrm{emp} \wedge n = \mathrm{nil}) \\ &\quad \vee (\exists l, r.\ n \mapsto \langle l, r \rangle * \mathrm{bt}(l) * \mathrm{bt}(r)) \end{aligned}$$

Here, the base case of an empty tree, i.e., an empty heap segment, is specified by the predefined predicate "emp", and value "nil" denotes a null pointer. In the inductive case, operator "$\mapsto$" in $n \mapsto \langle l, r \rangle$ indicates that argument n points to a valid memory region containing two pointers "$\langle l, r \rangle$". The predicate is subsequently invoked twice with l and r as argument, respectively. The separating conjunction requires the memory regions of the left sub-tree, the right sub-tree, and the current node to be disjoint, and thus implicitly handles all three mentioned constraints.

In contrast, specification languages built upon classical logic, such as the Java Modeling Language (JML) [13], require explicit statements about memory separation, typically resulting in less concise specifications. Nevertheless, JML is often considered more engineering friendly, because specifications are written similar to boolean expressions in Java. However, less expressive pattern-based languages than JML and separation logic may be preferred in an MDE approach for particular activities such as visualization, due to their intuitive appeal.

```
1  void print_bt_wrapped(struct bt* root) {
2   struct mem_pool* mp = mem_pool_init();
3   bt(mp, root); // evaluate predicate bt
4   print_bt(root); // invoke function
5   mem_pool_free(mp);
6  }
7  void bt(struct mem_pool* mp, struct bt* this) {
8   if (!(this == 0)) {
9    struct bt* left = bt_left(mp, this);
10   struct bt* right = bt_right(mp, this);
11   // label memory region as visited
12   mem_chunk(mp, ((void*)this), sizeof(struct bt));
13   bt(mp, left); bt(mp, right);
14  }
15 }
```

Figure 2: Secure wrapper excerpt generated for predicate bt by the approach of [17].

2.2 Generating Wrappers For Input Validation

To ensure that data provided to software components at trust boundaries adheres to the expected constraints, the defensive programming paradigm demands a thorough input validation. However, writing input validation code manually is an error-prone task, especially for complex objects such as dynamic data structures. This is not unlike the situation when implementing sequential control software from concurrent state machines, which motivated the automation of coding. To automatically generate secure wrappers

which check whether a referenced memory structure meets its specification, the work in [17] employs separation logic shape predicates for specifying the format of data exchanged at trust boundaries.

We studied in [6] how high-level data structure information can be translated to separation logic shape predicates, to make the approach of [17] more accessible to engineers. Figure 2 depicts an excerpt of a secure wrapper generated for our binary tree structure. The excerpt evaluates the bt predicate for the memory location specified by pointer `root` before invoking method `print_bt`; separation logic semantics is obeyed by storing visited memory regions in `struct mem_pool`.

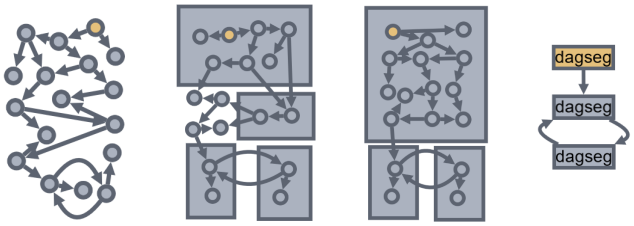


Figure 3: The cyclic RTL tree structure observed in the GNU compiler, visualized as a flat memory graph (left) and shape predicate summarization using the binary tree segment (mid left) and directed acyclic graph segment predicate in unfolded (mid right) and folded state (right).

2.3 Visualizing Memory Structures

Another focus of our research is the debugging of executable code. We are especially concerned with program comprehension by enhancing the visualization of program heaps with information about dynamic data structures contained in the code. While the majority of MDE tools usually simulate models, our visualization approach utilizes our models, i.e., shape predicates, to analyze a program state observed during debugging.

Our work in [5] introduces MGE (*Memory Graph Explorer*), a memory analyzer and visualizer that combines a shape predicate-based memory graph abstraction with an interactive visualization. Separation logic shape predicates are matched against the observed memory graph to identify and name data structure instances. Thereby, the objects associated with an identified structure can be summarized in an interactive visualization. In contrast to existing tools for memory graph visualization, our predicate-based abstraction can also derive meaningful summarizations for corrupt data structure instances, where one or multiple references have been set incorrectly. These deviations often cause bugs and make debugging necessary in the first place. In addition, our approach offers adjustable abstractions for large heaps.

Figure 3 displays four different visualizations obtained from MGE for a tree-like structure used by the GNU compiler, which incorrectly contains a cycle in the program state, causing the compiler to crash. While the curved visualization style for cyclic edges already makes the cycle location obvious, this mechanism does not scale to large graphs. Our abstraction employed by MGE is able to identify groups of objects that adhere to a certain shape predicate and group them in the visualization. Folding these groups of objects together and representing them as a single group node retains a comprehensible visualization. Regarding our binary tree example in Figure 3, a binary tree segment predicate identifies four groups of objects and the generalized directed acyclic graph segment predicate identifies three groups, thus drastically simplifying the manual search effort for the faulty cyclic edge.

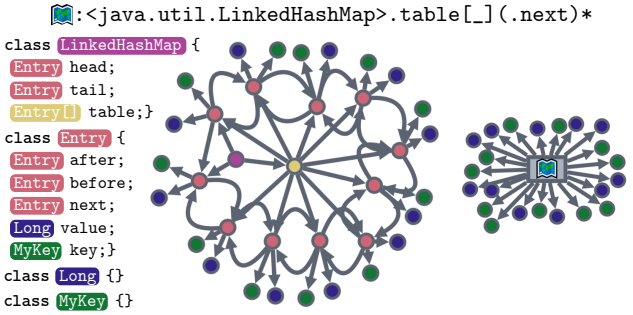


Figure 4: Definition of a heap pattern (top) for the `java.util.LinkedHashMap` class (left), for transforming a flat (mid) to a summarized memory graph (right).

While the ability of reusing shape predicates to inform program state visualization is beneficial, it is cumbersome to specify each object structure to be summarized during debugging as a shape predicate. To tackle this problem, our work in [4] extends MGE with the concept of *heap patterns*, a lightweight language paired with an evaluation semantics to characterize and identify object structures. Albeit less expressive than separation logic, heap patterns enable an intuitive style of specifying object structures. They still achieve a coarse-grained abstraction that is sufficient for detecting unexpected object sharing and informing visualization with object group hierarchies.

Heap patterns use a regular expression-like language to describe paths in a memory graph. A collection of paths that belong to the same heap pattern and originate from the same root object, describes a group of objects. Figure 4 shows a description of the type information for class `java.util.LinkedHashMap` and a heap pattern characterizing the objects associated with this class at runtime. This heap pattern matches paths starting at objects of class `LinkedHashMap`, followed by an arbitrary object in the `table` array, followed by dereferencing the `next` reference an arbitrary number of times. Considering the `table` array alone does not suffice to capture all objects related to a `LinkedHashMap`, because objects sharing the same hash are chained via attribute `next`. Enriching the flat memory graph visualization from Figure 4 with this heap pattern allows us to summarize all data structure related objects and display them as a single group node.

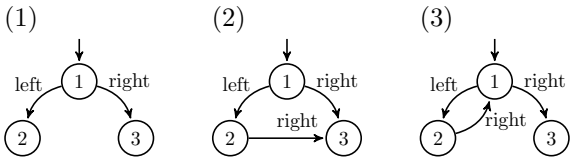


Figure 5: A valid exemplary binary tree (1) and two invalid trees (2–3) generated by a single mutation on instance level.

2.4 Generating Test Cases

Model-based testing traditionally centers around behavioral models, e.g., state machines or UML activity diagrams, in contrast to structural models. While this is beneficial for generating test steps, it makes producing concrete test data difficult, especially when the structure of objects is complex, as is the case for dynamic data structures. Similar to the wrapper generation topic introduced in Section 2.2, we are particularly interested in validating the correctness of an input validation method. Although wrappers can in principle be generated automatically, manually written code might yield better performance, and testing can further enhance trust.

To this extent, our work in [1] introduces a novel mutation-based approach for the automatic generation of test inputs for input validation methods for dynamic data structures. *Negative test inputs*, i.e., invalid data structure instances, are of particular interest. Existing approaches in the field of bounded-exhaustive testing [14] have difficulties in generating negative test inputs for complex data structures, simply because the number of invalid data structure instances is often significantly larger than the number of valid ones. In the light of this, our approach stands out in that it generates small test suites of representative inputs.

Our approach receives a shape predicate of a data structure as input. In a first step, we derive positive test inputs up to a certain size (*bound*), i.e., valid data structure instances that pass input validation. These instances are then mutated wrt. their pointer fields to derive mutated and possibly invalid test inputs, which should be detected by the input validation (see Figure 5). Further work [2] has extended our approach with value mutations to corrupt data structures that reason about payload values, such as binary search trees (BST), AVL trees (AVL), and red-black trees (RBT). A preliminary evaluation (see Table 1) shows that our approach produces fewer test inputs than the prior state-of-the-art tool *Korat* [14] while achieving an equally high instruction coverage.

2.5 Extracting Shape Predicates for Verification

Separation logic not only enables a concise characterization of complex object structures, but it also lends itself to the verification of programs operating on shared mutable data structures. The well-known verifier VeriFast [18] supports separation logic for C and Java programs and checks whether a given program complies with its specification encoded as a contract in the form of pre- and postconditions to methods. When verifying code containing dynamic data structures, a verifier typically requires an inductive shape predicate, like predicate bt, describing the run-time structure of the data structure. This is problematic because many software engineers have little training in formal methods.

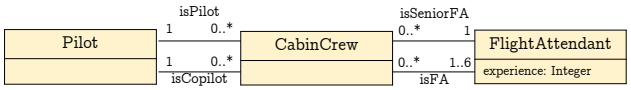
Table 1: Preliminary evaluation benchmark results that report, for the approaches Korat [14] and our single mutation, the number of generated negative test inputs and their achieved instruction coverage (prefix '*' denotes optimal coverage) for a data structure's input validation method

Data structure	Bound (#nodes)	Korat		Our Approach	
		#neg. inputs	instr. cov.	#neg. inputs	instr. cov.
BST	1	3	84%	2	84%
	2	184	*94%	19	*94%
AVL	1	7	80%	3	80%
	2	1640	*89%	30	*89%
RBT	1	15	29%	4	29%
	2	8236	*92%	35	*92%

Our work in [3] addresses this problem by automatically deriving candidate shape predicates from a set of data structure instances obtained from memory snapshots taken at runtime. This serves a similar purpose to extracting class diagrams from code, namely supporting engineers with complex modeling tasks. To handle memory graphs containing possibly nested data structures, we first decompose the input graphs into sub-graphs, each of which exhibits a single data structure. Then, we generate candidate shape predicates of increasing complexity and retain only those that lend themselves to characterize the memory graphs provided as input. Our search algorithm also tackles complex shape predicates that require additional arguments to encode backlinks. The following separation logic predicate btp encodes a binary tree with an additional argument p to encode a parent pointer:

$$\begin{aligned}\mathrm{btp}(n,p,\mathbf{s}) \quad \stackrel{\text{def}}{=} \quad & (\mathrm{emp} \wedge n = \mathrm{nil} \wedge \mathbf{s} = \mathbf{0}) \\ & \vee (\exists\, l, r, \mathbf{s_1}, \mathbf{s_2}.\ n \mapsto \langle l, r, p\rangle \\ & \quad * \mathrm{bt}(l, n, \mathbf{s_1}) * \mathrm{bt}(r, n, \mathbf{s_2}) \\ & \quad \wedge \mathbf{s} = \mathbf{1} + \mathbf{s_1} + \mathbf{s_2})\end{aligned}$$

The shape constraints (in roman font) of predicate btp are automatically extracted by our approach and may further be refined by verification engineers to also consider additional properties, such as the size of the data structure (in bold font), or payload values and their relationships, such as sortedness.




```
1  CabinCrew inv pilotAndCopilotAreDistinct:
2    (self.isPilot <> self.isCopilot)
3  CabinCrew inv seniorFAIsAmongFAs:
4    self.isFA->includes(self.isSeniorFA)
5  FlightAttendant inv experienceIsGreaterThanZero:
6    (self.experience > 0)
7  CabinCrew inv seniorFAHasMostExperience:
8    self.isFA->forAll(f:FlightAttendant |
9      (self.isSeniorFA.experience >= f.experience))
```

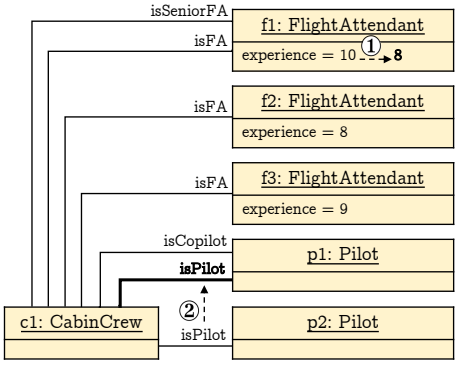


Figure 6: UML class diagram (top) with OCL constraints (mid), and object diagram (bottom) of a valid instance with a candidate value mutation ① and association mutation ②.

3 Test Case Generation for Object-oriented Software

While dynamic data structures employ a regular, recursive pointer structure, the object structure of object-oriented software is more diverse in terms of classes and associations. In the course of MDE, code generation (e.g., by UML Lab[3]), verification (e.g., by KeY[5]), and visualization (e.g., by USE[6]) approaches have been developed that handle such diverse structures. However, the area of test case generation is, in our experience, not yet fully supported.

Just as with dynamic data structures, it is necessary to check the object structure in order to identify potentially corrupt data passed through interfaces. Due to the diversity of references and constraints, such input validation may not be trivial to implement and should be tested thoroughly. We believe that our MDE-inspired test case generation technique developed in the context of dynamic data structures [1] can also be applied to object-oriented software and help us to produce small sets of concrete, characteristic test inputs.

As a running example consider, a simple object-oriented system for managing the cabin crew on an airplane, i.e., pilots and flight attendants (FA), and the rules for their team formation. The structural properties of such object systems are typically modeled with UML class diagrams enriched with OCL constraints [7] or, e.g., in the Alloy specification language [16]. Our system's structure is shown as a class diagram in Figure 6 (top) with four OCL constraints (bottom): The cabin crew consists of a pilot and a copilot, which must be distinct (see `pilotAndCopilotAreDistinct`). In addition, the cabin crew has at least one senior flight attendant and up to five additional flight attendants with at least one year of experience (see `seniorFAIsAmongFAs` and `experienceIsGreaterThanZero`). The senior flight attendant has the most experience of all attendants (see `seniorFAHasMostExperience`).

Modeling tools for object-oriented software commonly offer the generation of instances of the modelled structure, e.g., USE [11] and Alloy [12] generate exemplary object diagrams from a class diagram using an OCL evaluator and an SMT solver, respectively. Such tool support can be employed to generate *positive* test data, i.e., instances that satisfy the given constraints [16]. However, characteristic *negative* test data not satisfying the constraints is crucial for assessing whether invalid object structures are passed to methods. The two mutation-based approaches described next are tailored to deriving negative test data while reusing the above tools' capabilities of generating valid model instances.

Mutating at instance level In analogy with our single pointer mutations applied to instances of dynamic data structures, one can construct negative test data by injecting small faults into valid model instances, e.g., by mutating associations between objects or values within an object. Figure 6 (bottom) shows a candidate object diagram for our CabinCrew example with two exemplary mutations: ① setting the experience of the senior flight attendant to 0, thereby violating `experienceIsGreaterThanZero` and `seniorFAHasMostExperience`, and ② changing the target of the `isPilot` association to the current copilot, thereby violating `pilotAndCopilotAreDistinct`.

We consider instance mutations worthwhile to explore for object-oriented software, because they can be applied independently of the specification language. However, further research is required to apply our instance mutation approach developed in the context of dynamic data structures to object-oriented systems. Suitable mutation operators for values and associations in object-diagrams need to be proposed, and evaluated wrt. to their achieved code coverage and fault-detection capabilities for input validation methods.

Mutating at specification level Our instance mutation approach is feasible for dynamic data structure, because test inputs of small sizes typically suffice to achieve a good coverage. In contrast, object-oriented systems often operate on large program states that may yield unmanageably large test suites when exhaustively mutating instances.

To tackle the problem of generating a small but characteristic set of test cases, we propose mutating

directly at specification level. Our envisaged mutation operator is inspired by the concept of *classifying terms* (CT) [11, 9] which derives more diverse test data by manually providing additional constraints to specify equivalence classes among test inputs. Departing from this, our mutation operator negates each existing constraint from the specification, thus resulting in specifications that each describe a particular class of negative test data.

The OCL specification of our running example consists of four individual constraints each describing a distinct property of the cabin crew system. We construct mutated specifications by negating each of the four OCL constraints, and generate instances from these mutated specifications in the desired size. Compared to instance-based mutations, the obtained negative test data can directly be linked to the incorrectly implemented part of the specification, thereby fostering traceability. For example, mutation ② in Figure 6 (bottom) is generated using our specification mutation approach when negating constraint `pilotAndCopilotAreDistinct`. In contrast, mutation ① invalidates the two constraints `experienceIsGreaterThanZero` and `seniorFAHasMostExperience`, and can thus either be obtained by an instance mutation or by negating both constraints.

Future work should assess whether negating single constraints at specification level suffices to achieve characteristic test data, or whether multiple negations need to be considered, too. Similar to instance mutations, the necessary size of generated negative test data remains a question that requires further investigation. This future research will strengthen the MDE of object-oriented systems by automatically generating test data wrt. structure and value constraint validation.

4 Conclusions

This paper presented our research on the analysis of complex software systems with dynamic data structures, and compared and contrasted it to the traditional model-driven engineering of control-intensive systems. Using *shape predicates* as structural models of dynamic data structures, we contributed to tool-supported methods to aid secure wrapper generation, memory graph visualization, test case generation, and formal verification.

While the classical MDE of control-intensive systems works with *behavioral* models and uses simulation for their validation, our shape predicates are *structural* models that drive visualizations for validating memory snapshots. In addition, while model-based testing generates test cases from *behavioral* models, we employ shape predicates to generate test data for *structural* integrity checks. Such checks are also needed in object-oriented software when passing object structures around, so our work on mutation-based test case generation is also applicable there.

References

[1] J. H. Boockmann, K. Jacob, and G. Lüttgen. Towards robustness testing of functions operating on dynamic data structures. In *Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS)*, volume 2021/7, 2021.

[2] J. H. Boockmann, K. Jacob, and G. Lüttgen. Automatic mutation-based test generation for data structure input validation. 2022. Submitted for publication.

[3] J. H. Boockmann and G. Lüttgen. Learning data structure shapes from memory graphs. In *LPAR*, volume 73 of *EPiC Series in Computing*, pages 151–168. EasyChair, 2020.

[4] J. H. Boockmann and G. Lüttgen. Heap patterns for memory graph visualization. IEEE, 2022. To appear in *VISSOFT*.

[5] J. H. Boockmann and G. Lüttgen. Shape-analysis driven memory graph visualization. In *ICPC*, pages 298–308. ACM, 2022.

[6] J. H. Boockmann, G. Lüttgen, and J. T. Mühlberg. Generating inductive shape predicates for runtime checking and formal verification. In *ISoLA*, volume 11245 of *LNCS*, pages 64–74. Springer, 2018.

[7] J. Cabot and M. Gogolla. Object Constraint Language (OCL): A definitive guide. In *SFM*, volume 7320 of *LNCS*, pages 58–90. Springer, 2012.

[8] W. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, 2012.

[9] R. Clarisó and M. Gogolla. A feasibility study on using classifying terms in Alloy. In *OCL*, volume 2513 of *CEUR Workshop Proceedings*, pages 45–58. CEUR-WS.org, 2019.

[10] A. Rodrigues da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Comput. Lang. Syst. Struct.*, 43:139–155, 2015.

[11] F. Hilken, M. Gogolla, L. Burgueño, and A. Vallecillo. Testing models and model transformations using classifying terms. *Softw. Syst. Model.*, 17(3):885–912, 2018.

[12] D. Jackson. *Software Abstractions — Logic, Language, and Analysis*. MIT Press, 2006.

[13] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, volume 523 of *Kluwer Intl. Series in Engin. and Comput. Sci.*, pages 175–188. Springer, 1999.

[14] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid. Korat: A tool for generating structurally complex test inputs. In *ICSE*, pages 771–774. IEEE, 2007.

[15] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE, 2002.

[16] A. Sullivan, K. Wang, R. N. Zaeem, and S. Khurshid. Automated test generation and mutation testing for Alloy. In *ICST*, pages 264–275. IEEE, 2017.

[17] N. van Ginkel, R. Strackx, and F. Piessens. Automatically generating secure wrappers for SGX enclaves from separation logic specifications. In *APLAS*, volume 10695 of *LNCS*, pages 105–123. Springer, 2017.

[18] F. Vogels, B. Jacobs, and F. Piessens. Featherweight VeriFast. *Log. Methods Comput. Sci.*, 11(3), 2015.