CoverageCity: Test Coverage for Clinical Guidelines

Reinhard Hatko[1] and **Joachim Baumeister**[2] and **Frank Puppe**[3]

Abstract. In this paper, we introduce various metrics for test coverage of clinical guidelines, modeled in the graphical language DiaFlux. Additionally, an intuitive visualization method supports the process of test creation and communicating the reached coverage levels to medical experts involved in the authoring of the guideline. The goal is to reach a sufficiently high test coverage to assure patient safety under all circumstances.

1 Introduction

Testing is an important step in the development of any software artifact, be it a program, a knowledge-based system, or a clinical guideline. The two most prevalent testing strategies in Software Engineering are *black-box* and *white-box* testing. The former approach is unconcerned with the actual implementation and derives the test cases solely from the underlying specification. The latter one, in contrast, allows to create tests based on the implementation and to examine it during execution of the tests. This introspection enables to capture which basic elements of the tested artifact were executed - and thus *covered* - by a test suite. Different metrics of *Test Coverage* were developed to objectively measure and assess the thoroughness of such testing efforts. In classic Software Engineering, metrics have been defined to assess the coverage of, e.g., methods of a program, statements of a method, taken decisions of control-flow, and so on [16].

Hence, the benefit of coverage metrics - and their proper visualization - is two-fold, increasing the effectiveness and efficiency of the test creation process: First, they help to avoid the creation of redundant tests. Second, they can be used to identify untested elements.

Both are also important aspects in the area of knowledge-based system in general and computerized clinical guidelines in particular. The creation of test cases for a clinical guideline will most likely involve both parties, the medical expert and the knowledge engineer. It is thus an expensive task, which should be completed efficiently. Though, the overall goal is to create a guideline, that assures patient safety under all circumstances, which can best be guaranteed by a thorough test suite. This is especially important in the area of automated guidelines, which are applied by closed-loop devices. They act autonomously on a patient to improve her state, without requiring constant supervision by a clinical user.

For the interpretation of coverage metrics a visualization is helpful, especially to find untested elements. Test coverage of software programs most usually is visualized by some kind of syntax highlighting, by coloring, e.g., the executed statements. Though also graphical representations were developed, e.g., [11]. We adopted a visualization method from a related area in (object-oriented) Software Engineering, namely *Software Metrics*. They are used to assess code quality with respect to structural properties of classes, e.g., number of methods, number of members, lines of code, and so forth. Those metrics are purely static, not involving the execution of the program itself. They can graphically be visualized as a *CodeCity* [22] to determine design flaws.

In this paper, we introduce various metrics to determine the test coverage of clinical guidelines, modeled in the graphical language DiaFlux. Furthermore, we adapted the city metaphor by creating a *CoverageCity* for communicating the reached coverage levels to the involved medical experts in an accessible manner.

The rest of this paper is structured as follows: In the next section we give a short introduction into the graphical language DiaFlux for clinical guidelines. Section 3 presents coverage metrics for DiaFlux models. Following, in Section 4, we present the results of a case study. Finally, we conclude the paper with a summary and an outlook.

2 The DiaFlux Guideline Language

Clinical guidelines are an accepted means to improve patient outcome. Therefore, they offer a standardized treatment, based on evidence-based medicine. They are developed for several decades. In their beginnings, they were solely text-based documents that relied on the proper application by clinicians. The ongoing computerization and data availability, also in domains with high-frequency data as, e.g., Intensive Care Units (ICUs), allows for an automation of guideline application by medical devices.

Several formalisms for Computer-Interpretable Guidelines were developed, every one with its own focus, like shareability between institutions [4] or decision support. Most of them are graphical approaches, that employ a kind of *Task-Network-Model* to express the guideline steps [17]. However, in the area of (semi-)closed-loop devices, rule-based approaches are predominant, e.g., [12, 15].

A downside of rule-based representations is their lower comprehensibility compared to graphical ones. This especially holds true, as medical experts are most usually involved in the creation of guidelines. Therefore, we have developed a graphical guideline formalism called DiaFlux [7]. Its main focus lies on the direct applicability and understandability by domain specialists.

2.1 Application Scenario

The main application area of DiaFlux are mixed-initiative devices that continuously monitor, diagnose and treat a patient in the setting of an ICU. Such closed-loop systems interact with the clinical user during the process of care. Both, the clinician and the device, are able to initiate actions on the patient. Data is continuously available as a

[1] University of Wuerzburg, Germany, email: hatko@informatik.uni-wuerzburg.de
[2] denkbares GmbH, Germany, email: joachim.baumeister@denkbares.com
[3] University of Wuerzburg, Germany, email: puppe@informatik.uni-wuerzburg.de

result of the monitoring task. It allows for repeated reasoning about the patient state, and to carry out appropriate actions to improve her condition, if necessary.

2.2 Language Description

To specify a clinical guideline, two different types of knowledge have to be effectively combined, namely declarative and procedural knowledge [6]. The declarative part contains the facts of a given domain, i.e., findings, diagnoses, treatments and their interrelation. The knowledge of how to perform a task, i.e., the correct sequence of actions, is expressed in the procedural knowledge. It is responsible for the decision which action to perform next, e.g., asking a question or carrying out a test, in a given situation. Each of these actions has a cost (monetary or associated risk) and a benefit (like information gain or therapeutic effect) associated with it. Therefore, the choice of an appropriate sequence of actions is mandatory for efficient diagnosis and treatment.

In DiaFlux models, the declarative knowledge is represented by a domain-specific ontology, which contains the definition of findings and solutions. This application ontology is an extension of the task ontology of diagnostic problem solving [3]. The ontology is strongly formalized and provides the necessary semantics for executing the guideline. Like most graphical guideline languages, DiaFlux employs flowcharts as the Task-Network-Model. They describe decisions, actions and constraints about their ordering in a guideline plan. These flowcharts consist of nodes and connecting edges. Nodes represent different kinds of actions. Edges connect nodes to create paths of possible actions. Edges can be guarded by conditions, that evaluate the state of the current patient session, and thus guide the course of the care process. Figure 1 shows a module of an exemplary guideline for diagnosing weight problems.

In the following, we informally describe the most important language elements:

- **Test node:** Test nodes represent an action for the acquisition of data during runtime. This may trigger a question, the user has to answer, or data to be automatically obtained by sensors or from a database.
- **Solution node:** Solution nodes are used to set the rating of a solution based on the given inputs. Established solutions generate messages for the clinical user and can, e.g., advice him to conduct some action.
- **Wait node:** Upon reaching a wait node, the execution of the protocol is suspended until the given period of time has elapsed.
- **Composed node:** DiaFlux models can be hierarchically structured. Defined models can be reused as modules, represented by a composed node in the flowchart using it.
- **Abstraction node:** Abstraction nodes offer the possibility to create abstractions from available data. These values can then be used for therapeutic actions by influencing the settings of the host device.

2.3 Guideline Execution

The execution engine for DiaFlux models is intended for, but not limited to, closed-loop devices, that provide data from sensors or manually entered by the clinical user and carry out therapeutic actions on the patient, i.e., changing device settings in certain ranges. The architecture of the DiaFlux guideline execution engine consists of three components. First, a knowledge base, that contains the application ontology and the flowcharts. Second, a blackboard, that stores all findings about the current patient session. Third, a reasoning engine, that executes the guideline and carries out its steps, depending on the current state as given by the contents of the blackboard. Therefore, the reasoning engine is notified about all findings, that enter the blackboard. A more thorough introduction to the DiaFlux language and its execution engine is given in [7].

The execution of the guideline is time-driven. The reasoning starts by acquiring data and by interpreting this data. The results are written to the blackboard. Then, the guideline is executed. This involves making decisions and possibly the generation of hints to the user and therapeutic actions by the device. Finally, the time of the next execution is scheduled, pausing the execution until that instance in time, waiting for the effects of the therapeutic interventions to take place.

3 Test Coverage of Clinical Guidelines

Verification and validation of a clinical guideline are important steps in its development. That way, patient safety shall be assured under all circumstances. Verification usually consists of formal methods, proofing, that a given guideline is free of internal inconsistencies and incompleteness. Normally, these kinds of checks can be performed without executing the guideline. An overview of verification methods applied to clinical guidelines is given in [10]. Anomaly detection for DiaFlux models is described in [8]. In contrast, the validation of a guideline usually involves its execution by a set of test cases (i.e. a test suite) and comparing the actual results against the expected ones [2]. The thoroughness of such empirical testings can be determined by different metrics of test coverage.

In conventional software engineering (SE), test coverage (also known as *code coverage*) is a well-established technique to measure how well a piece of software is exercised by a test suite. Often, the reached level of coverage is also used as an indicator for the quality of the tested program, as tested elements are less likely to contain errors than untested ones. Coverage criteria have been defined on different levels of granularity, from the method-level down to single statements, or even parts of them (so called *condition coverage*) [16]. In the field of AI research, coverage measures have been proposed for rule-based systems. In this case, the results of such a coverage analysis can, e.g., be used to prune the rule base [1]. For graphical model representations, coverage measures have been proposed, e.g., for business processes [14], taking their specifics into account, for example, the coverage of *fault handlers*.

In general, employing coverage metrics during the creation of a test suite may help improving it in terms of minimality and completeness. While a high coverage of the object under test is worthwhile, this should be accomplished with the possibly minimal set of test cases, as test creation is a difficult and costly task. This especially holds true for the test creation of clinical guidelines, as it may involve knowledge engineers as well as domain specialists like medical experts. Therefore, adopting the mentioned techniques for clinical guidelines and their graphical representations, offers the possibility to improve this process.

In the following, we introduce coverage metrics on different levels of granularity to assess the test coverage of a DiaFlux guideline. Such a guideline usually is modularized in self-contained modules, i.e. flowcharts. To represent this modularization also by the coverage metrics, we define most of them over the elements of individual flowcharts, hence the restriction of nodes and edges to those of a single one. This focusing alleviates the increase of coverage by additional tests, as deficiencies are easier to spot.

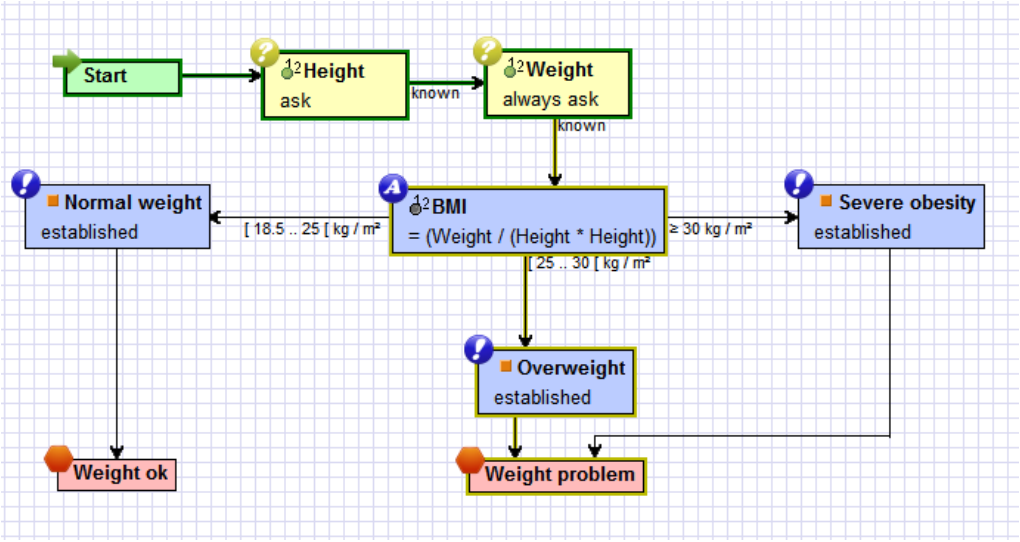

Figure 1. A module of a DiaFlux guideline for diagnosing weight problems.

3.1 Flowchart Coverage

Flowchart Coverage is defined as the ratio of the number of flowcharts that are executed by a test suite and the overall number of flowcharts in the guideline.

Let F be the set of DiaFlux models in guideline G, and F_T^{ex} be the set of flowcharts executed by test suite T. Then, the Flowchart Coverage FC_T of G is given by:

$$FC_T(G) = \frac{|F_T^{ex}|}{|F|}$$

This metric only gives a bird's eye view of the testing situation. It can be used to guarantee at least some testing in all areas of the guideline during the starting phase of test creation. As it is a very coarse-grained measurement, a FC_T value of 100% should be aimed at. Otherwise, major parts of the guideline remain untested. In SE, the equivalent metric is *function coverage*, which reports, if every function of a program has been tested.

3.2 Node Coverage

Nodes represent the elementary steps of a guideline. A node being covered by a test suite, means, that its associated guideline step has been carried out at least once during the execution of the test suite.

Let $N_f, f \in F$ be the set of nodes of the flowchart f, and $N_{f,T}^{ex}, f \in F$ be the set of nodes of the flowchart f, that are executed by test suite T. The according *Node Coverage* metric NC_T of a flowchart f for a given test suite T can then be calculated as:

$$NC_T(f) = \frac{|N_{f,T}^{ex}|}{|N_f|}, f \in F$$

Similar to *Flow Coverage*, a NC_T level of 100% should be reached for every flowchart in the guideline, as untested nodes can enact actions with unforeseen effects. This metric is equivalent to *Statement Coverage* in classic SE, which reports, if every statement of a tested function has been executed.

3.3 Edge Coverage

Edges are used to create the control flow of a flowchart, by defining paths of possible sequences of guideline steps. Every node can have several outgoing edges. Each of these edges can be guarded by a condition, to select the appropriate successor node, depending on the outcome of each guideline step. Therefore, the *Edge Coverage* metric reports, if all possible outcomes of the guideline steps - in terms of the equivalence classes that are defined by the edge guards - have been considered within the test suite.

Let $E_f, f \in F$ be the set of edges of flowchart f, and $E_{f,T}^{ex}, f \in F$ be the set of edges of the flowchart f, that are executed by test suite T. Then, EC_T is defined as the number of activated edges of a flowchart f to their overall number, executing a given test suite T:

$$EC_T(f) = \frac{|E_{f,T}^{ex}|}{|E_f|}, f \in F$$

As edges connect nodes, an *Edge Coverage* subsumes the *Node Coverage* of the according flowchart. This metric can be compared to Decision Coverage in SE, that keeps track, if each decision in a program under test (e.g., `if`- and `switch`-statements) has at least once been taken and once not.

3.4 Condition Coverage

An edge guard may not be an atomic condition, but consist of several sub-conditions, connected by boolean operators. For such non-atomic guards, *Edge Coverage* gives no detailed information about which of its sub-conditions were satisfied and which were not. This is of special interest, if the sub-conditions are joined by an *OR*-operator. In this case, every possible combination of atomic conditions, that can fulfill the overall condition, have to be tested.

A more detailed view about this issue is given by *Condition Coverage*. It checks, if every atomic condition has once been satisfied and once not. In classical SE, several different metrics for this issue have been developed, e.g. *Modified Condition / Decision Coverage*

[5]. As those can directly be applied to the guarding conditions of edges, we will not further elaborate on this issue.

3.5 Path Coverage

A path through the guideline consists of consecutive nodes and edges. Such a path can be seen as the execution of decisions and actions for a given clinical scenario. In Software Engineering, it usually is not possible to reach a full path coverage, as soon as loops are involved, as each number of iterations results in an additional path. In clinical guidelines, there are no loops of an unlimited number of iterations, as, for instance, some time has to pass, until an action can be repeated. Nevertheless, the number of paths through the complete guideline throughout multiple nested DiaFlux models most likely exceeds the possibilities of test creation. Given a proper modularization, each flowchart is responsible for a specific aspect of a guideline. Each path through such a single flowchart can be seen as one specific scenario concerning this aspect. Therefore, we assess each flowchart independently, and define an according *Path Coverage* metric over the paths of each individual flowchart.

Let $P_f, f \in F$ be the set of paths through flowchart f, and $P^{ex}_{f,T}, f \in F$ be the set of paths through flowchart f, that are executed by test suite T. Then, PC_T is calculated as the number of paths taken through flowchart f, by the execution of test suite T, divided by the total number of paths:

$$PC_T(f) = \frac{|P^{ex}_{f,T}|}{|P_f|}, f \in F$$

Even with a proper modularization given, not every modeled path may be enactable, due to implicit dependencies between the guideline steps. If certain combinations of decisions and actions on a single path exist, that can not occur, the targeted value of Path Coverage has to be decreased accordingly.

As a path consists of consecutive nodes and edges, Path Coverage satisfies Node Coverage as well as Edge Coverage.

Path Coverage, as defined above, is a rather aggregated measurement and thus gives little advice of how to improve coverage with further tests. Therefore, Path Coverage can also be restricted to the paths through a specific node $n \in N$:

Let P_n be the set of paths containing n ($\forall p \in P_n : n \in p$), and $P^{ex}_{n,T}$ be the set of paths containing n exercised by test suite T. Accordingly, the Path Coverage of node n is defined as:

$$PC_T(n) = \frac{|P^{ex}_{n,T}|}{|P_n|}$$

A node with a low Path Coverage is only tested under a small fraction of the contexts in which it is contained. Again, further tests should be created for those scenarios, unless they expose dependencies that makes not every path feasible.

3.6 Value Coverage

The metrics presented so far assess the test coverage with respect to structural properties, each considering some kind of modeled element, like nodes and edges. However, the actual input data, that directs the execution of the guideline, is not assessed by these metrics in any way.

Beside the mentioned control-flow-based metrics, a second perspective on coverage in classic Software Engineering is given by data-flow-based metrics. Those measure the coverage of *definition-use (du)* sequences of variables, i.e., a block of instructions in which a variable is defined and subsequently used without a redefinition of the variable, e.g., [19]. Black-box testing strategies concerned with data usage are *Equivalence Partitioning* and *Boundary Value Analysis* [16]. As an exhaustive testing with all valid input data is most likely not tractable even for a small program, its specification can be used to partition the input space into equivalence classes. Under the assumption that each value of a partition is treated equally by the program, an arbitrary representative of each class can be chosen for a test case. As errors are more probable at the boundaries of an equivalence class, Boundary Value Analysis is often used to derive additional test cases at these spots [20], for example to find "off-by-one" errors (e.g., resulting from the use of the operator "$\leq$" rather than "$<$" in a numerical comparison).

DiaFlux models do not contain variables as they are common in procedural programming languages, and the input data is not modified during guideline execution. Therefore, a *definition-use* analysis is not applicable. Equivalence Partitioning and Boundary Value Analysis can also not be used as they are. Explicit equivalence classes usually can not be stated for inputs. Even thresholds for less determined assessments (like "low", "normal", "high") are often hard to specify for a medical expert. Those can furthermore vary between different types of patients, which, e.g., share an underlying disease. However, for each numerical input, a contiguous interval of possible values can typically be given, according to the human body's physiological system and/or the preconditions of guideline applicability. To assure a proper coverage of allowed input data regardless of concepts like equivalence classes, we define the metric *Input Coverage*: Let the interval $[min_i; max_i]$ be the domain D_i for numerical input i, and $n \in \mathbb{N}, n > 0$ be the number of equally-sized partitions of D_i. The function $cover(i, j)$ returns 1, iff the $j-th$ partition of D_i contains at least one input value in test suite T, and 0 otherwise. Then, the Input Coverage of i is given by:

$$IC_{n,T}(i) = \frac{\sum_{j=1}^{n} cover(i, j)}{n}$$

Clearly, the significance of Input Coverage depends on the actual value of n. It should be chosen to appropriately represent the sensitivity of the outcome of the guideline to changes in values of i. At later stages of test creation, the value can be increased stepwise to test more fine-granular in terms of i's input values.

Similarly, the output of the guideline (which mainly consists of numerical values of the host device's settings in predefined ranges) can be assessed by the analog defined metric *Output Coverage* $OC_{n,T}$.

3.7 Measuring Test Coverage

Commonly, there are two strategies to gather the data needed for calculating test coverage metrics. The first one is called *instrumentation*, which modifies the tested piece of software by including new code that collects the necessary information. The second strategy is *tracing*, which traces the executed elements by using some sort of debugging API (Application Programming Interface) of the execution environment. Clearly, both approaches have an effect on the execution time of the tests, as additional data has to be gathered. An advantage of tracing is, that it does not alter the executed artifact. Under certain circumstances this may also influence its behavior. Under this aspect, "tracing" seems preferable, if the necessary API is available.

3.8 Visualization of Test Coverage

The calculated metrics result in a numerical value representing the test coverage of the exercised artifact. This may very well be comprehensible for software and knowledge engineers, though for non-technical domain specialists, like medical experts, these sole numbers may not be accessible enough. Furthermore, only a proper composition of metrics yields a meaningful overall picture, as each metric represents a different aspect of coverage. Thus, an intuitive visualization as a means for communicating the reached coverage levels to domain specialists seems preferable. One approach to this need are so called "Polymetric Views" [13]. They allow to display various metrics in an aggregated view.

4 Test Coverage for DiaFlux Guidelines

This section describes an implementation of coverage metrics for DiaFlux guidelines and a small case study.

4.1 Implementation

The development environment for DiaFlux guidelines is integrated into the Semantic Wiki KnowWE. We created a plugin to calculate the test coverage of DiaFlux models, when executing a test suite. It employs the tracing approach to collect the information about exercised elements of the guideline. For each execution of the guideline, the chosen path according to the input data is recorded. After finishing the test suite, the metrics are calculated and can subsequently be visualized as CoverageCity, which can freely be scaled and rotated (cf. Figure 2). For creating the visualization, we used the WebGL[4]-based JavaScript library SceneJS[5]. It is hence accessible with every (modern) web browser from within KnowWE, not requiring any proprietary software.

4.2 The CoverageCity Visualization

For an accessible visualization of the reached coverage levels, we use the metaphor of a city. It has been introduced as graphical representation of static code metrics (e.g. number of methods,...) in the context of reverse-engineering of software systems ("CodeCity") [21]. Such a city consists of districts that represent the nesting structure of packages and buildings representing classes. Each building is located in the district corresponding to its package. The actual values of the metrics of each class determine the visual appearance of the matching building. A building can represent up to four different metrics, influencing its length, width, height, and color. Besides the package structure, districts can depict one metric by their color.

We adapted the city metaphor for the visual representation of test coverage of DiaFlux guidelines. Districts stand for individual flowcharts. They are nested as given by their call hierarchy. Buildings correspond to the nodes of the district's corresponding flowchart. Edges are not explicitly included but mapped to one of the buildings' dimensions. In particular, we used the following assignment of metrics to visual properties of buildings:

- *Length:* The number of outgoing edges of a node determines the length of the building.
- *Width:* The width of a building relates to the number of outgoing edges that are covered by the test suite.
- *Height:* The height of a building shows how often it was covered by the test suite.
- *Color:* The color represents how well the paths, that go through a particular node, are covered by a test suite. In case a building is "red", only few paths are executed. "Green" stands for a high Path Coverage.

[4] http://webgl.org
[5] http://scenejs.org

4.3 Interpretation of the Visualization

The complexity that a node introduces into a guideline, mostly comes from its number of outgoing edges. In case the associated action has numerous possible outcomes (each represented by an outgoing edge), it is more likely to contain errors. Thus, one dimension of the base area of a building is influenced by the number of outgoing edges of the corresponding node. The increased size makes the building easier to spot. The other dimension of the base area grows with the number of covered outgoing edges. As a result, a non-quadratic building stands for a node, whose outgoing edges are not completely covered. The lower the aspect ratio, the more uncovered edges exist. This deficiency in test coverage can easily be observed.

The color of a building depicts its Path Coverage. A red one, represents a node, that is contained in much more paths than were covered. With increasing Path Coverage, the color becomes green. The height of a building corresponds to the number of times, it has been exercised by the test suite. Clearly, those two properties are not independent. On the one hand, a small building is more likely to be contained only in a small number of paths, and thus be red. On the other hand, a building that is tall and red, implies that the corresponding node is often tested under similar circumstances. This can give hints for creating new test cases, that are not contained so far. Tall, green buildings have been thoroughly exercised and do not require additional test cases.

The nesting level of the districts helps in estimating the necessary effort for testing a specific flowchart or node. Deeply nested module are probably harder to reach by a test case, as each module may only be called under certain preconditions.

The aggregated view of CoverageCity makes it easy to spot deficiencies in test coverage quickly. Though, a more detailed view is necessary to identify the specific nodes and their test deficiencies. Hence, each building be selected by the user. Then, the corresponding flowchart is shown below the CoverageCity, and the node and its coverage is highlighted visually.

4.4 Case Study

Currently, we are involved in the implementation of a computerized guideline for automated mechanical ventilation [9]. The guideline is intended to run on a mechanical ventilator, and is able to derive new ventilatory settings in order to improve the ventilation of the patient. First testing efforts of the guideline were conducted using a physiological simulation. The guideline was run against a software tool that simulates a mechanically ventilated patient. It employs a physiological lung model to determine the effects of the current ventilation settings to the patient. The simulation is able to deliver the necessary data (ventilation settings and measured patient response) to the guideline execution engine. Based on this data, the guideline derives optimized settings and returns them to the simulation environment, that uses them for the further simulation. The simulation tool was used by medical experts to generate the test cases and review the derived ventilation settings. The generated test cases are saved to a file,

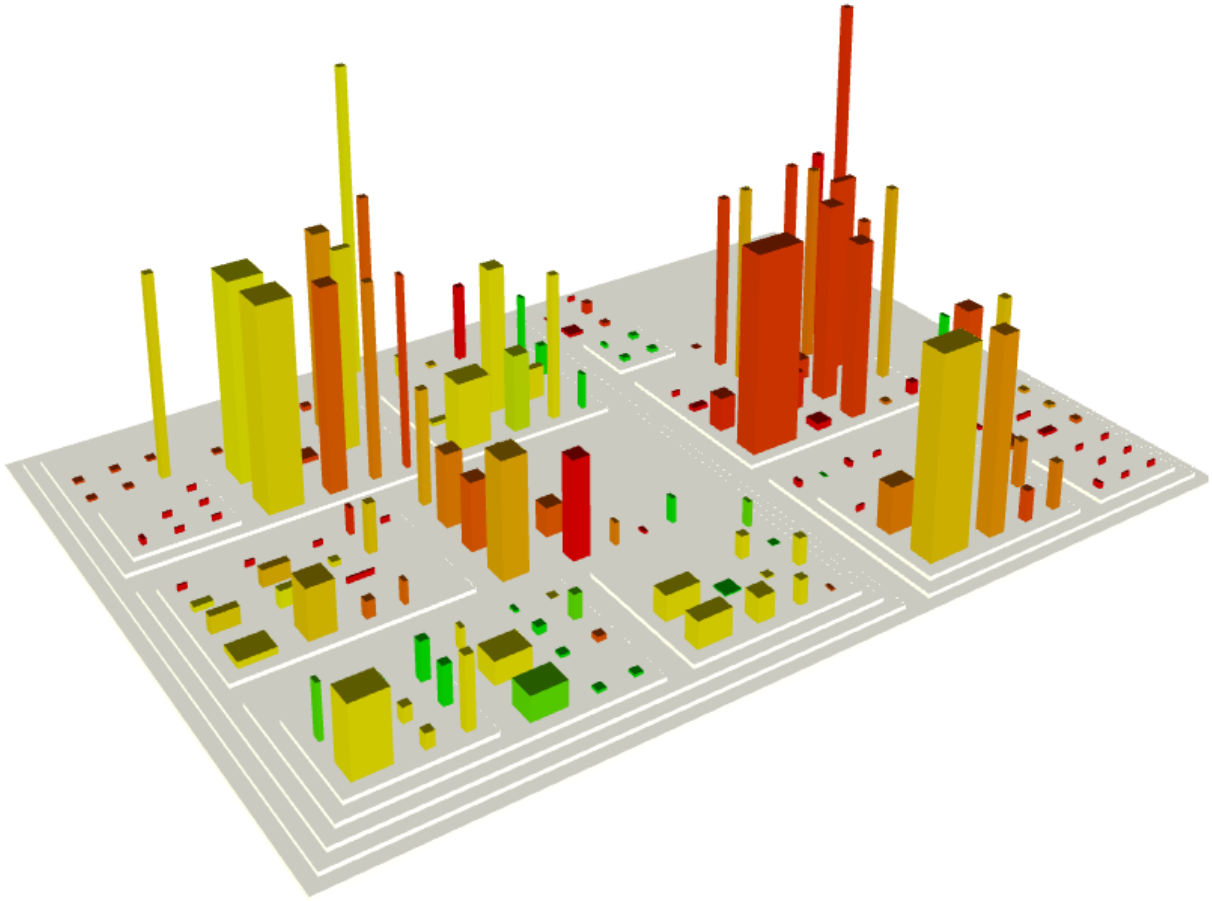

Figure 2. The test coverage visualization using the city metaphor. Nodes are represented by buildings, flowcharts by districts. The visual properties of building are determined by the coverage metrics of the corresponding node.

and are then uploaded to KnowWE for the introspection of guideline execution and coverage analysis.

We selected a sample of ten generated test cases. The visualization of the acquired coverage levels is shown in Figure 2. Currently, we are in the process of evaluating our visualization with medical experts. Furthermore, we identify other meaningful assignments of metrics to visual properties of the buildings.

5 Conclusion

In this paper, we formally defined different coverage metrics to assess the thoroughness of testing efforts for clinical guidelines. They can be used to identify insufficiently tested elements, and to improve the process of test creation in terms of efficiency, as this may involve domain specialists as well as knowledge engineers. An intuitive visualization method helps in communicating the acquired coverage levels to domain specialists, for which numerically expressed metrics probably are less helpful than for knowledge engineers.

Additional metrics could be defined over more dynamic aspects of a guideline. First, the distribution of values could be tracked for each activated flowchart element. As there clearly are dependencies between the actual values and the possible ones - given by the context (i.e. path) of the element - proper preprocessing would be necessary. Ultimately, this could give insight, if parts of the guideline were only tested, e.g., for a certain patient type. Second, it would be helpful to define scenarios with respect to the occurrence of a certain sequence of input data or therapeutic actions over time and trace their coverage by the test suite. In terms of the CoverageCity-visualization, we will evaluate different mappings of the coverage metrics to the visual properties of the city, to create new perspectives on test coverage.

One shortcoming of white-box testing in general is, that it is unable to detect errors by omission, i.e. some requirement may not have been included in the implementation under test. An approach to find this type of errors is *Requirements-based Test Coverage* [18]. It defines coverage with respect to implementation-independently defined requirements that are exercised by a test suite. Having formally defined requirements, this approach should be transferable to testing clinical guidelines.

Acknowledgements

University of Würzburg is funded by the German Federal Ministry for Education and Research under the project “WiM-Vent” (Knowledge- and Model-based Ventilation), grant number 01IB10002E.

REFERENCES

[1] Valerie Barr, 'Applications of rule-base coverage measures to expert system evaluation', *Knowledge-Based Systems*, **12**, 27–35, (1999).

[2] Joachim Baumeister, 'Advanced empirical testing', *Knowledge-Based Systems*, **24**(1), 83–94, (2011).

[3] Joachim Baumeister, Jochen Reutelshoefer, and Frank Puppe, 'KnowWE: A semantic wiki for knowledge engineering', *Applied Intelligence*, **35**(3), 323–344, (2011).

[4] Aziz A. Boxwala, Mor Peleg, Samson Tu, Omolola Ogunyemi, Qing T. Zeng, Dongwen Wang, Vimla L. Patel, Robert A. Greenes, and Edward H. Shortliffe, 'GLIF3: a representation format for sharable computer-interpretable clinical practice guidelines', *J. of Biomedical Informatics*, **37**(3), 147–161, (2004).

[5] J.J. Chilenski and S.P. Miller, 'Applicability of modified condition/decision coverage to software testing', *Software Engineering Journal*, **9**(5), 193–200, (1994).

[6] Paul de Clercq, Katharina Kaiser, and Arie Hasman, 'Computer-interpretable guideline formalisms', in *Computer-based Medical Guidelines and Protocols: A Primer and Current Trends*, eds., Annette ten Teije, Silvia Miksch, and Peter Lucas, 22–43, IOS Press, Amsterdam, The Netherlands, (2008).

[7] Reinhard Hatko, Joachim Baumeister, Volker Belli, and Frank Puppe, 'DiaFlux: A graphical language for computer-interpretable guidelines', in *Knowledge Representation for Health-Care*, eds., David Riaño, Annette ten Teije, and Silvia Miksch, volume 6924 of *Lecture Notes in Computer Science*, 94–107, Springer, Berlin / Heidelberg, (2012).

[8] Reinhard Hatko, Joachim Baumeister, Gritje Meinke, Stefan Mersmann, and Frank Puppe, 'Anomaly detection in DiaFlux models', in *KESE7: 7th Workshop on Knowledge Engineering and Software Engineering, San Cristobal de La Laguna, Spain, November 10, 2011*, volume 805 of *CEUR Workshop Proceedings*, Tenerife, Spain, (2011). CEUR-WS.org.

[9] Reinhard Hatko, Dirk Schädler, Stefan Mersmann, Joachim Baumeister, Norbert Weiler, and Frank Puppe, 'Implementing an automated ventilation guideline using the semantic wiki knowwe', in *EKAW 2012: 18th International Conference on Knowledge Engineering and Knowledge Management*, eds., Heiner Stuckenschmidt, Annette ten Teije, and Johanna Voelker, (2012).

[10] Arjen Hommersom, Perry Groot, Michael Balser, and Peter Lucas, 'Formal methods for verification of clinical practice guidelines', in *Computer-based Medical Guidelines and Protocols: A Primer and Current Trends*, eds., Annette ten Teije, Silvia Miksch, and Peter Lucas, 63–80, IOS Press, Amsterdam, The Netherlands, (2008).

[11] J.A. Jones, M.J. Harrold, and J. Stasko, 'Visualization of test information to assist fault localization', in *Proceedings of the 24th international conference on Software engineering*, pp. 467–477. ACM, (2002).

[12] H.F Kwok, D.A Linkens, M Mahfouf, and G.H Mills, 'Rule-base derivation for intensive care ventilator control using ANFIS', *Artificial Intelligence in Medicine*, **29**(3), 185 – 201, (2003).

[13] Michele Lanza and Stéphane Ducasse, 'Polymetric views - a lightweight visual approach to reverse engineering', *IEEE Trans. Software Eng.*, **29**(9), 782–795, (2003).

[14] Daniel Lübke, Leif Singer, and Alex Salnikow, 'Calculating bpel test coverage through instrumentation', in *AST*, eds., Dimitris Dranidis, Stephen P. Masticola, and Paul A. Strooper, pp. 115–122. IEEE, (2009).

[15] Stefan Mersmann and Michel Dojat, 'SmartCare$^{\text{tm}}$ - automated clinical guidelines in critical care', in *ECAI'04/PAIS'04: Proceedings of the 16th European Conference on Artificial Intelligence, including Prestigious Applications of Intelligent Systems*, pp. 745–749, Valencia, Spain, (2004). IOS Press.

[16] Glenford J. Myers, Corey Sandler, and Tom Badgett, *The art of software testing*, John Wiley & Sons, Hoboken, N.J., 3 edn., 2011.

[17] Mor Peleg, Samson Tu, Jonathan Bury, Paolo Ciccarese, John Fox, Robert A Greenes, Silvia Miksch, Silvana Quaglini, Andreas Seyfang, Edward H Shortliffe, Mario Stefanelli, and et al., 'Comparing computer-interpretable guideline models: A case-study approach', *JAMIA*, **10**, (2003).

[18] A. Rajan, 'Coverage metrics to measure adequacy of black-box test suites', in *Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on*, pp. 335 –338, (sept. 2006).

[19] Sandra Rapps and Elaine J. Weyuker, 'Selecting software test data using data flow information', *IEEE Trans. Softw. Eng.*, **11**(4), 367–375, (April 1985).

[20] S.C. Reid, 'An empirical analysis of equivalence partitioning, boundary value analysis and random testing', in *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pp. 64 –73, (November 1997).

[21] Richard Wettel and Michele Lanza, 'Visualizing software systems as cities', in *VISSOFT*, eds., Jonathan I. Maletic, Alexandru Telea, and Andrian Marcus, pp. 92–99. IEEE Computer Society, (2007).

[22] Richard Wettel and Michele Lanza, 'CodeCity: 3D visualization of large-scale software', in *ICSE Companion*, eds., Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, pp. 921–922. ACM, (2008).