Anomaly Detection in DiaFlux Models

Reinhard Hatko[1] and Gritje Meinke[2] and Joachim Baumeister[3] and Stefan Mersmann[2] and Frank Puppe[1]

[1] University of Würzburg, Institute of Computer Science, Dept. of Artificial Intelligence and Applied Informatics
97074 Würzburg, Germany
{hatko, puppe}@informatik.uni-wuerzburg.de
[2] Dräger Medical GmbH, 23558 Lübeck, Germany
{gritje.meinke, stefan.mersmann}@draeger.com
[3] denkbares GmbH, Friedrich-Bergius-Ring 15, 97076 Würzburg, Germany
joachim.baumeister@denkbares.com

Abstract. In recent years, the use of graphical knowledge representations more and more proved to be suitable for building diagnostic and therapeutic knowledge systems. When building such systems, the quality assurance of the knowledge base is an integral part of the development process. In this paper, we present the flowchart-based language DiaFlux and we describe a collection of anomalies, that can occur when using the language for knowledge base development. The naming of many shown anomalies was motivated by the experiences made in real-world projects.

1 Introduction

In recent years, intelligent systems have been established in a variety of domains. When building such systems the developers no longer depend on pure rule-based representations, but more and more use graphical approaches that often allow for a more intuitive knowledge elicitation process. In the medical domain, for instance, workflow-oriented representations emerged in the last years to build systems based on existing guidelines and standard operating procedures (SOPs), see for instance [1].
In an industrial setting, the development of such knowledge bases is integrated in a predefined knowledge engineering process, that shows similar phases to general software engineering processes, see for instance [2,3]. All these process models also propose a quality assurance phase, where the developed artifact is tested by validation and verification methods. Here, usually the expected system behavior is tested with regression-based methods, such as empirical tests [4], but also checks at the component level are performed. The most commonly used verification method for component-based tests is the detection of (already known) anomalies. In Software Engineering such anomalies are related to object-oriented metrics [5] and bad smells [6]. Some typical examples for general anomalies are cyclic dependencies between classes and packages, infinite recursion, and

long/unmaintainable methods. The automated detection by a static code analysis and the (manual) elimination of such anomalies can prevent serious malfunctions of the built application.
It is easy to see, that the ideas of anomalies in general software code can be transfered to the artifacts produced in a knowledge engineering process. Here, the knowledge base is investigated in order to find deficient parts of the knowledge. In the past, verification methods for detecting anomalies in different knowledge representations were introduced, for instance see [7,8].
Approaches for the verification of workflow models are described, e.g., in [9]. In addition, some of the anomalies we identified represent a mixture of data- and control-flow anomalies and also involve a Truth Maintenance System. In this paper, we introduce the workflow-based knowledge representation DiaFlux for building diagnostic and therapeutic knowledge systems. The language is presented in Section 2 and Section 3 describes possible anomalies. We report a small case study in Section 4 and conclude the paper with a discussion in Section 5.

2 Graphical Knowledge Models with DiaFlux

This section first describes the application scenario. Then, we introduce the representation language *DiaFlux*.

2.1 Application Scenario

DiaFlux is a graphical guideline language intended to be used in mixed-initiative devices, that continuously monitor, diagnose, and treat a patient in the setting of an Intensive Care Unit (ICU). The clinical user interacts with such a semi-closed loop system during the care process. Actions on the patient can be initiated by both parties, the clinician and the device. Continuous reasoning is performed, as some data is continuously available as a result of the monitoring task. An execution environment for automated clinical care in ICUs and the implementation of a guideline for weaning from mechanical ventilation are presented in [10].

2.2 Language Description

Two kinds of knowledge have to be effectively combined for the specification of a clinical protocol, namely declarative and procedural knowledge [11]. The declarative part encompasses the facts and their relationships. The procedural knowledge reflects how to perform a task, i.e., the correct sequence of actions. The declarative knowledge particularly consists of the terminology, i.e., findings, solutions, and sometimes also therapies and their interrelation. The procedural knowledge is responsible for the decision which action to perform in a given situation, e.g., asking a question or carrying out a test. The appropriate sequence of actions is mandatory for efficient diagnosis and treatment, as each action has a cost (monetary or associated risk) and a benefit (for establishing or excluding currently considered solutions) associated with it. For the representation of the

procedural aspects, guideline languages employ different kinds of Task Network Models [1]. They constrain the ordering of decisions and actions in a guideline plan. Flowcharts are a common formalism to explicitly express this control flow. In DiaFlux models, a domain-specific ontology represents the declarative knowledge. It contains the definition of findings and solutions. This application ontology extends the task ontology of diagnostic problem-solving, as described in [12]. Due to its strong formalization, it provides the semantics necessary for the execution of the guidelines. The procedural knowledge is represented by flowcharts, that consist of nodes and edges. Different types of actions are represented by nodes. Connecting edges create possible sequences of actions. To constrain these sequences, an edge can be guarded by a condition that evaluates the state of the current session and thus guides the course of the care process.

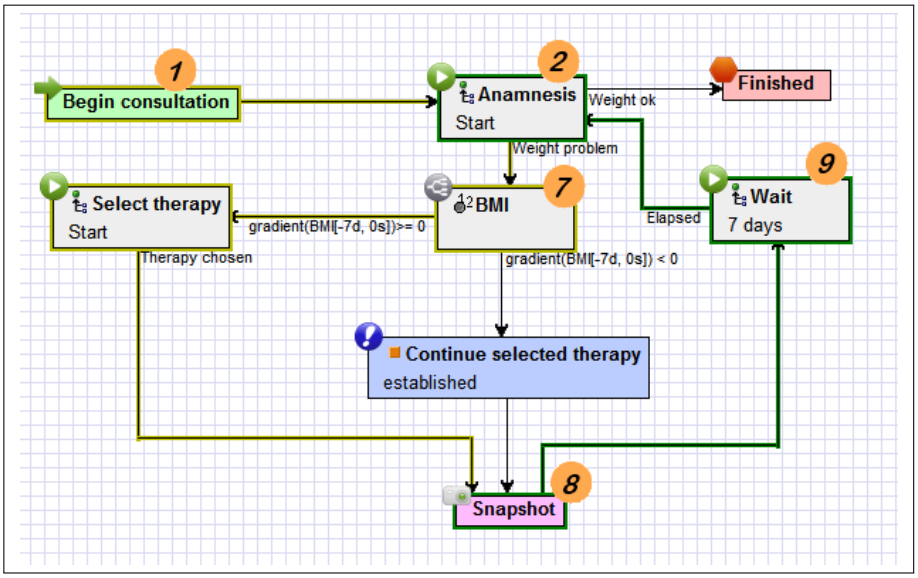

Fig. 1. The main model and starting point of a protocol for monitoring and treating overweight. The state of the current testing session is highlighted in green and yellow colors (black and grey in this figure, respectively).

In the following, we give a simple example of a protocol for the diagnosis and treatment of overweight, modeled in DiaFlux.
Figures 1 and 2 show parts of a protocol for the diagnosis and treatment of overweight modeled in DiaFlux. When a consultation session starts, the main module, as depicted in Figure 1, is activated. The execution begins at the *start node* (1), labeled "Begin consultation". It points to the *composed node* "Anamnesis" (2). When this node is reached, the according submodule (cf. Figure 2)

is called and its start node labeled "Start" is activated. The execution of the main module awaits the completion of the called submodule. Reaching the *test node* "Height" (3) data is acquired from the user. After entering the value for body height, the execution can continue to the next test node "Weight". As the weight is supposed to change from one session to the next, this test node acquires new data each time it is activated. Therefore, the specific testing action used is "always ask" instead of "ask". The first one triggers data acquisition even for inputs that are already known in order to update their value. After the value for "Weight" has been entered, the *abstraction node* (4) calculates the body mass index (BMI) from the acquired data and assigns the value to the input "BMI". An appropriate next action is chosen depending on the value of the BMI. For a value contained in the range of [25; 30[the execution progresses to the *solution node* (5) which establishes the solution "Overweight". The following *exit node* (6) labeled "Weight problem" terminates the execution of the module. The control flow then returns to the superordinate module. For other values of "BMI" the appropriate solution is established and the according exit node is returned as result of the "Anamnesis" protocol.

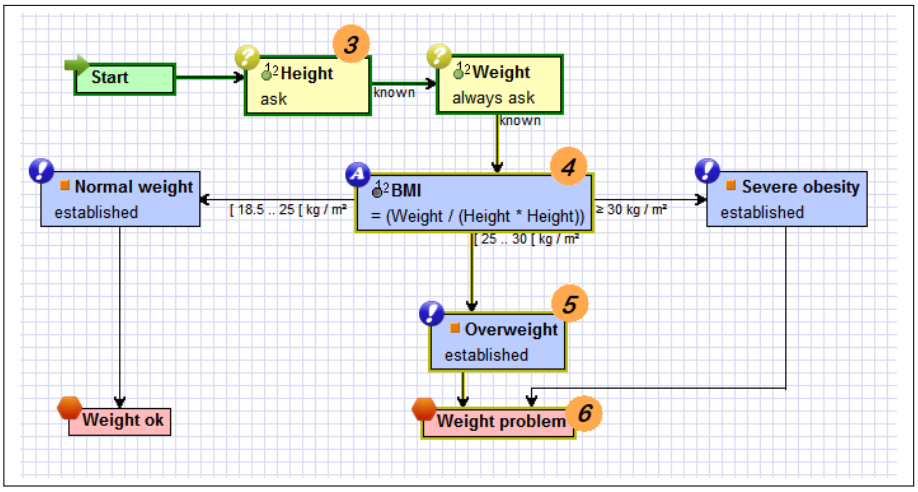

Fig. 2. The anamnesis module for acquiring data and establishing the current diagnosis.

Upon completion of the "Anamnesis" module, the appropriate successor node is chosen based on the returned result. In case of "Weight ok" the execution of the protocol ends by reaching the exit node "Finished", as there is no superordinate module to return to. Otherwise, a proper treatment is chosen based on the history of values of the BMI. The *decision node* (7) tests the gradient of BMI values. For a declining BMI (i.e., the patient is loosing weight), the previously selected therapy is continued. Otherwise, another therapy is chosen within the module

"Select Therapy"[1]. Both paths reach the *snapshot node* (8). On activation of this node, the execution state of the protocol is saved and truth maintenance will not retract any conclusion beyond this point. Furthermore all active nodes on the incoming path are deactivated, to allow their repeated execution. Next, the execution is suspended by the *wait node* (9), until the given time of 7 days has lapsed. Afterwards, a second anamnesis is conducted and the current BMI is calculated based on the newly acquired body weight. If it has decreased, so will the BMI and the current therapy is continued. Otherwise, a new therapy is selected and applied until a normal body weight is obtained.

A more detailed description of the DiaFlux language and the execution engine can be found in [13].

3 Anomaly Detection

There exists a large body of research concerning the detection of anomalies by verification methods, for instance for rule bases [14], for ontologies [15], for mixed verification of rules and ontologies [8]. In general, we distinguish the following types of anomalies for knowledge bases:

1. **Redundancy** defining duplicate or subsuming elements of the knowledge base
2. **Inconsistency** caused by contradicting elements of the knowledge base
3. **Missing knowledge** are absent parts of the knowledge base, that can prevent the proper execution of the knowledge
4. **Deficiency** comprising parts of the knowledge base, that worsen the design of the knowledge

In the following, we discuss these types in more detail and we introduce particular anomalies, that explain redundant, inconsistent, deficient, and missing knowledge especially in DiaFlux models.

It is important to notice, that the following presentation of anomalies is not an exhaustive set but more or less a collection of problems, that occurred during the development of industrial knowledge bases.

3.1 Redundancy

Redundant knowledge may be removed from the knowledge base without change in the semantics of the derivation behavior. Each found anomaly, however, needs to be considered carefully by a human knowledge engineer, since some kinds of redundancy can be used to increase the robustness of the knowledge base.

[1] The gradient of a single value is 0 and a therapy is chosen for the first time.

Redundant Calculation Abstraction nodes can be used to assign a value to a finding. That value can either be a constant number or can be calculated by a formula, aggregating the values of the others findings. The assignment of a constant value is redundant when the same value is assigned more than once on a given path. The assignment of a value derived from a formula is redundant, if the second calculation will yield the same result. This is the case if the second abstraction uses the same formula and if there is no path between the first and the second calculation that leads to the acquisition of new values for the findings used in the calculation.

Redundant Test Depending on the frequency the values of a finding may change, two different kinds of actions can be used for *test nodes*, "ask" and "always ask", respectively. The first one triggers the acquisition of data only if no value has been assigned to the finding so far. The latter demands new data each time the node is activated in the flowchart. If two test nodes are located on a connecting path and trigger an "ask" action on the same finding, the second test action is ignored and therefore redundant. In case the second node has more than one outgoing edge with different guards, the developer should consider to convert the node to a *decision node*.

3.2 Inconsistency

Inconsistent knowledge often yields unexpected and contradictory inferences during execution. Detected inconsistencies should be investigated thoroughly by the knowledge engineer and be considered for elimination in most cases.

Inconsistent Calculation As described in the anomaly *Redundant Calculation*, abstraction nodes can be used to assign a value to a finding. The assignment contains either a constant value or a formula that is evaluated. Such a calculation is inconsistent, if different values are assigned to one finding on a single, connected path of nodes. In the worst case, the assignment of the second value may force the truth maintenance system to illegally retract the followed path until the first assignment, and thus creates a truth maintenance cycle.

Inconsistent Test Action Two different types of testing actions are provided in DiaFlux for collecting data. For findings containing high frequency data (e.g. "blood pressure" in the medical domain), the testing action "always ask" is appropriate to be used; the action "ask" is appropriate for the single acquisition of data (e.g. when asking the age or sex of a patient). Using both types of testing actions for the same finding most likely hints to a design flaw. If the finding contains high frequency data, the value of the finding will not be updated upon reaching the node, that performs the "ask" action. Therefore an old value will be used, instead of acquiring new data. In the case of low frequency data, the value for the finding is acquired more often than necessary, if the action "always ask" is used.

3.3 Missing Knowledge

Some anomalies may point to unfinished areas of the knowledge base, for instance elements of the knowledge that are never used in problem-solving sessions.

Uninitialized Value Values of findings are calculated in the DiaFlux representation by using abstraction nodes. To conduct such a calculation, proper values have to be available for all findings that are included in the calculation. If at least one necessary finding is not acquired (or calculated itself) on at least one path leading to the abstraction node, then the calculation will not succeed and the execution of the path may stop at the abstraction node.

Missing Start Node A flowchart in DiaFlux can have several distinct entry points. Each one must begin with a *start node*. A flowchart not defining at least one start node, cannot be activated during execution and thus is isolated from the rest of the knowledge base.

Unconnected Node Every flowchart defines a process that begins at a *start node* and ends at an *exit node*. The activation of the nodes in between depends on the connecting edges and their respective guards. Any node (except a *start node*) that is missing an incoming edge cannot be activated during the problem-solving process. All successors of such a node are also unreachable unless they have an alternative incoming edge, which is itself connected to at least one *start node*.

Open Path End Every possible path in a flowchart has to be terminated by an *exit node*. Although, an open path end does not influence the execution of this particular flowchart, it will prevent the continuation of a superordinate flowchart. Thus, the flowchart is not returning to the super-flowchart, that called it. After reaching a *composed node* during execution, the calling flowchart awaits the termination of the called module by an *exit node*. If this does not exists, then the execution of the calling flowchart will not continue.

No Startup Flow Defined The execution of the knowledge base begins in a distinct flowchart, which has to be marked as *autostart* by the knowledge engineer. If no flowchart is marked accordingly, then none is activated at the start of a problem-solving session. Therefore, the execution will end immediately.

Unused Flowchart For improving the structure of the knowledge base, flowcharts can be nested. *Composed nodes* allow the execution of another flowchart module. A flowchart, that is neither marked as *autostart* nor is called by any *composed node* will never be executed during runtime.

Incompleteness of Edge Guards The definition of edge guards allows to select one of multiple outgoing paths at a node, depending on the current value of a finding. As the execution will continue only along an edge whose guard is evaluated to *true*, the entirety of guards defined at one node has to cover the complete range of possible values of the examined finding. Otherwise, the execution of the flowchart will stop at this node, if the current value does not match with an edge guard.

3.4 Deficiency

Deficiencies point to subtle parts in the knowledge base, that may benefit from a design improvement. The existence of such an anomaly, however, often does not affect the reasoning behavior in a bad manner.

Dead Path The possible paths through a flowchart are given by the edges between nodes. Every edge can be guarded by a condition that evaluates the values of findings entered into the system. An edge is activated, if its starting point is active and its condition evaluates to *true*. If a finding is used multiple times on a single path, then the guards at later edges have to be consistent to the possible values at that point. Otherwise such edges cannot be activated for certain values. An example is given in Figure 3.

Impossible Path When new findings are entered into the system, a truth maintenance system checks the state of all flowcharts. If the value of a finding has changed, all edges and nodes change their activation state according to the new values. In case an *abstraction node* calculates a value for a finding, that is used to guard an edge in the active path, the calculated value must not contradict that guard. Otherwise, the truth maintenance system will collapse the path to the *abstraction node* undoing its calculation. Therefore, the path starting at the *abstraction node* is impossible to continue.

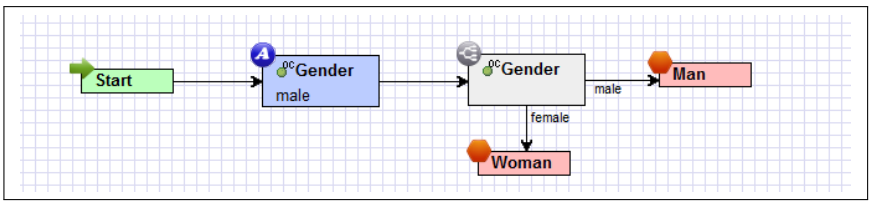

Fig. 3. A minimal example of a Dead Path. After setting the question "Gender" to "Male", the following decision node branches depending on its value. As it can only be "Male", the path leading to the exit node "Woman" can never be taken, and is therefore dead.

Disjointness of Edge Guards The guards on the outgoing edges of every node must be disjoint with respect to the possible outcomes of a node. If the domains of guards overlap, all belonging edges will be activated for according values. This easily happens, when defining intervals at a decision node that examine a numerical finding.

In this section, we introduced a selection of anomalies that can occur in DiaFlux knowledge bases. In the next section, we describe an implementation of a part of the shown anomalies and we report on some experiences.

4 Case Study

The DiaFlux development environment is integrated into the Semantic Wiki KnowWE [12]. KnowWE is a wiki aimed at building intelligent systems, offering methods to capture and execute strong problem-solving knowledge. A Continuous Integration (CI) tool supports the modeler during the development of the knowledge base by executing a configurable set of tests after each edit. The results of the recent build of the knowledge base are indicated to the user in an unintrusive manner. A detailed report is available on demand. The frequently running test procedures help to find modeling errors at an early stage.
We recently integrated detection algorithms for selected anomalies as described in Section 3 into the CI tool. The system was used in a couple of projects and received very positive feedback, from unexperienced as well as advanced users. A common mistake among modelers, that are new to the DiaFlux language, is to miss marking the *autostarting* flowchart. As a result the knowledge base seems to simply do nothing. In more complex knowledge bases, that are hierarchically structured and contain different possible paths of execution, the detection of anomalies like *Uninitialized Value* or *Dead Path* is very helpful as those are not only tested within each flowchart module but also across their boundaries along paths through composed nodes.

5 Conclusions

The development of knowledge-based software systems is similar to general software engineering approaches. We motivated that today's knowledge bases are often built using workflow-based languages; this especially holds in the medical domain, where existing guidelines and standard operating procedures are transfered into computer-interpretable models. In this paper, we discussed the problem of quality assurance of such models and we described the detection of anomalies in the models as an important aspect of quality assurance. We described the practical guideline language DiaFlux by an example protocol for overweight treatment. Furthermore, we introduced a selection of anomalies for this language. The selection of these anomalies is not exhaustive, but was motivated by our experiences in the development of industrial knowledge bases.

In the future, we plan to define a more exhaustive set of anomalies, including temporal ones, and relate the particular artifacts to anomalies already known in classical verification research. Often, a found defect is the start of a refactoring of the knowledge base. We are currently working also on refactoring methods for DiaFlux models, that are used to eliminate found deficiencies but also other kinds of anomalies.

References

1. Peleg, M., Tu, S., Bury, J., Ciccarese, P., Fox, J., Greenes, R.A., Miksch, S., Quaglini, S., Seyfang, A., Shortliffe, E.H., Stefanelli, M., et al.: Comparing computer-interpretable guideline models: A case-study approach. JAMIA **10** (2003) 2003
2. Schreiber, G., Akkermans, H., Anjewierden, A., de Hoog, R., Shadbolt, N., de Velde, W.V., Wielinga, B.: Knowledge Engineering and Management - The CommonKADS Methodology. 2 edn. MIT Press (2001)
3. Baumeister, J., Seipel, D., Puppe, F.: Agile development of rule systems. In Giurca, Gasevic, Taveter, eds.: Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches. IGI Publishing (2009)
4. Baumeister, J.: Advanced empirical testing. Knowledge-Based Systems **24**(1) (2011) 83–94
5. Simon, F., Steinbruckner, F., Lewerentz, C.: Metrics based refactoring. In: Software Maintenance and Reengineering, 2001. 5th European Conference on. (2001) 30–38
6. Fowler, M.: Refactoring. Improving the Design of Existing Code. Addison-Wesley (1999)
7. Ayel, M., Laurent, J.P.: Validation, Verification and Test of Knowledge-Based Systems. Wiley (1991)
8. Baumeister, J., Seipel, D.: Anomalies in ontologies with rules. Web Semantics: Science, Services and Agents on the World Wide Web **8**(1) (2010) 55–68
9. Aalst, W.M.P.v.d.: Workflow verification: Finding control-flow errors using petri-net-based techniques. In: Business Process Management, Models, Techniques, and Empirical Studies, London, UK, Springer-Verlag (2000) 161–183
10. Mersmann, S., Dojat, M.: SmartCare^{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, Valencia, Spain, IOS Press (2004) 745–749
11. de Clercq, P., Kaiser, K., Hasman, A.: Computer-interpretable guideline formalisms. In ten Teije, A., Miksch, S., Lucas, P., eds.: Computer-based Medical Guidelines and Protocols: A Primer and Current Trends. IOS Press, Amsterdam, The Netherlands (2008) 22–43
12. Baumeister, J., Reutelshoefer, J., Puppe, F.: KnowWE: A semantic wiki for knowledge engineering. Applied Intelligence (2011)
13. Hatko, R., Baumeister, J., Belli, V., Puppe, F.: Diaflux: A graphical language for computer-interpretable guidelines. In: KR4HC'11: Proceedings of the 3th International Workshop on Knowledge Representation for Health Care. (2011)
14. Preece, A., Shinghal, R.: Foundation and application of knowledge base verification. International Journal of Intelligent Systems **9** (1994) 683–702
15. Gómez-Pérez, A.: Towards a framework to verify knowledge sharing technology. Expert Systems with Applications **11**(4) (1996)