On Restaurants and Requirements: How Requirements Engineering may be Facilitated by Scripts

Christoph Peylo

Deutsche Telekom Laboratories, Berlin
christoph.peylo@telekom.com

Abstract. Requirements engineering is a central part of software projects. It is assumed that two third of all errors in software projects are caused by forgotten requirements or mutual misunderstandings in the requirement gathering process. Due to the inherent structure of project planning and the project management process, it is very unlikely that this problem will be solved unless the process itself is changed or we develop tools that possess some intelligence to facilitate the assessment of requirements. In this paper a position for the latter approach is formulated. It is argued that it is feasible to establish a domain ontology based on meta information and explanations that are represented as scripts. It is shown that this ontology has to be constructed in a dynamic way to reflect the dynamics of the requirements engineering process. Finally, it is sketched how use cases and test cases can be derived from this ontology.

1 The Difficulty of Creating Novel Things in Time and Budget

A project is a temporary endeavour undertaken to create a unique product, service or result [1]. Consequently, a project has a limited time frame and creates unique deliverables like services, products or results. The success of a project depends on the meeting of the outcome with its expected result. Thus, the deliverables of a project have to fit in their intended niche.

Accordingly, identifying the project requirements is of uttermost importance for any project. The project requirements describe the characteristics, i. e. conditions or capabilities, that must be met by the deliverables [1]. Further steps of project management such as establishing objectives, balancing the demands for scope, time, cost and quality, etc., have the clear and comprehensive definition of requirements as a prerequisite.

Hence, the identification of requirements (requirements engineering) is critical for the success or failure of a project. It is assumed that two thirds of all errors in system development are caused by forgotten requirements or misunderstandings.

1.1 Project Requirements and Project Risks

The understanding of project requirements must be reached (according to the classical project management approach) in the very early stages of the project (initiation stage), in order to formulate the project scope statement.

The project scope statement describes the project's deliverables and the work that is to be accomplished to create them. Frequently, it is quite a coarse grained level of detail in which the conditions and capabilities of the intended system are described at that time. Nevertheless, this will build the basis for further planning during the next stages.

From a business point of view, this is a quite disadvantageous setting: the gathering and analysis of the key project deliverables and the decision of how to supply them is in a project phase before there is a signed contract. Consequently, the time and work in this phase is not payed by the intended customer-to-be. It is quite common in large projects to get compensation for feasibility studies, but this does not apply to all projects.

In summary, if this stage is not performed well, it is unlikely that the project will be successful in meeting the expectations of the stakeholders.

2 On the Difficulty to Represent Understanding with Imprecise Formalisms

To understand the project requirements it is important to understand the background, i. e. the business processes from which the need of a new product or service has arisen. This helps to understand the stakeholder's needs, wants and expectations and what has to be achieved to satisfy a contract.

Alas, the setting of such a task is often quite complex. The general background is often quite specialized and not easy to understand, business processes are sometimes ill defined. The terminology used by the stakeholders may differ from the vocabulary of the requirements engineer or, even worse some terms may be used with a slightly different semantics (which usually becomes apparent later in the project). Last but not least, some terms may turn out to be not terms at all but business processes (cf. [2]).

Thus, there are numerous and well known reasons (cf. [3]) why requirements are left out or not fully understood:

- Business processes are ill defined. A business process may consist of several sub-processes which may be too trivial for the stakeholders to mention.
- Terms are used with (slightly) different semantics.
- Business processes of a new (innovative) setting are ill defined.
- Novel business processes may interfere with existing business processes.
- There are contradictionary processes involved.

It would be necessary to describe the system and its background in a comprehensive and almost complete way to eliminate all sources of those mistakes. There are hardly any projects where there is time and budget for such a comprehensive

approach. It is important to remember that in the initiation phase of a project a contract is not signed, i. e. there is no or insufficient compensation for an in-depth approach available.

Consequently, it is quite common to concentrate on the representation of the functional requirements with use cases. A use case describes the interaction between a system and a request that originates from outside of that system. Use cases represent that interaction as a sequence of single steps and events to achieve a specific goal. There are several representation schemes (most common: the UML use case diagram in various versions) to graphically express this kind of interaction. The meaning (or semantics) of the use case is not represented by the well defined building blocks of the formalism [4], [5], but shall constitute itself (helped by various annotations) in the mind of the reader. This approach is quite common but prone to misunderstandings.

Admittedly, those representation formalisms have a certain beauty: they represent complex interactions in a compact way that may be perceived quickly (at least in comparison to lengthy (and often tiresome to read) definitions in natural language). Due to their seeming clarity and formality they are often over-estimated. Nevertheless, they are deceptive with respect to their precision and expressiveness. There main limitations are:

1. Weak and not well defined semantics of relations.[1]
2. The expressiveness of graphical representation schemes is limited per se to a fragment of first order logic (existential quantified, conjunctive connected). Trying to extend the symbology by annotations (to cover modal or second order constructs) will increase the confusion, not the expressiveness.
3. Use cases represent the interaction in the communication between user and system. Commonly, they refer to sub processes and documents that are interchanged during those process steps without explaining the content in full detail. Thus, generally, it is not possible to decide by the study of a use case whether the process flow may lead to the desired result (i. e. the system output may be achieved, given the set of input).

After this synopsis of the requirements gathering process and the difficulties that exist in avoiding misunderstandings it is concluded that either this process should be upvalued considerably (in terms of time and money), or tools for facilitating this process on a more semantic level should be applied.[2]

In the remaining part of this paper, an approach is sketched how existing AI concepts could be deployed for this purpose.

---

[1] This is no new insight, as shown by Woods [6].

[2] There are several approaches that try to support the requirements engineering process in linking use cases to contextual scenarios (e. g. [7] and [8]). But the representation of scenarios is done in natural language and suffers from the known problems connected with that approach (cf. sec. 2 and sec. 4).

3 Contributions from AI

How can AI facilitate the requirements engineering process, and, more specifically, how can AI contribute to avoid misunderstandings? From an AI perspective, the problem is situated in the context of formalizing domain knowledge and to explain and to communicate this knowledge (cf. [9], [10]).

3.1 On Explanations and Understanding

Explanation can be regarded as the process by which we make sense of the world [11]. Thus, we construct structures from which knowledge can be derived at a later stage. Explanations seem to be linked to the process where knowledge structures are constructed or communicated. Thus, explanations connect phenomenon in a systematic way to make outcomes predictable. Whereas explanations reflect the process of understanding, knowledge seems to be more about the management of realization. Thus, knowledge can be understood as a tertiary relation (someone assigns someone knowledge about something).[3]

At a very basic level explanation equals understanding: We believe that we understand why a person is doing what he is doing if we can point to a script that he or she is following [11]. A script is a structured representation describing a stereotyped sequence of events in a particular context and forms a very basic knowledge structure [11]. In that sense the famous restaurant script [13] was used to understand the basic interactions in a restaurant. This script-type of understanding is equivalent to *making sense* and is to be distinguished from the deeper (*cognitive*) understanding, of course. For the purpose of this paper *making sense* will do. Thus, scripts as condensed or compiled explanations may be considered as suitable building blocks for a system with a shallow degree of self-awareness.[4]

3.2 The Structure of a Script

As stated above, a script models a flow of action in a specific setting. To understand this setting, i. e. to interact in a limited communication [11], it is necessary to tag structural information to its building blocks to facilitate the semantic processing in a computer system. The components of scripts are

- Entry conditions that must be met before the script may be started.[5]
- Results or conditions that are true once the script has terminated. Thus, a script has a specific setting as an entry condition and after the script terminated, the setting (the state of affairs) is different from the initial setting.[6]

[3] This holds true especially in educational contexts, see [12].

[4] A script resembles goals and scenarios in the COSMOD-RE approach of Pohl [14]. But this approach is a methodology (cf. [15]) which does not result in a system as proposed here.

[5] In the famous restaurant script these include a restaurant that is open and a customer that is hungry.

[6] In the restaurant script effects of the script are that the customer is not hungry and has less money.

- Roles as placeholders for actors or objects in actions that the individual participants perform.
- Scenes that reflect temporary aspects of a script. A scene works like a script in a script. It encapsulates operations that change the state of affairs.
- The entities involved as objects or passive parts in that script.
- A set of well defined actions. Actions are distinguished by the arity of the relation (e. g. transitive verbs are a binary relation, double-transitive verbs a tertiary relation), and a type restriction with respect to roles and entities.

Those components offer the tools, by which a *lightweight* understanding may be modeled. Logically, entities may be modeled as predicates, actions may be represented as relations on roles. Roles are variables (with type restrictions) for entities. A state of affairs may be represented as a list of predicates that hold in that moment.

3.3 Example: the Restaurant Script

The classic example of Schank's theory is the restaurant script. The script theory is closely related to Schank's concept of *conceptual dependencies* [13]. According to that concept, the meaning of natural language sentences should be expressed by using conceptual primitives. In the example given below the conceptual dependencies are marked using uppercase letters and a typewriter font. The meaning of these primitives is as follows:

`PTRANS`**:** Transfer of the physical location of an object (i.e. go).
`MBUILD`**:** Building new information of old information (i.e. decide).
`MTRANS`**:** Transfer of mental information (i.e. tell).
`ATRANS`**:** Transfer of an abstract relationship (i.e. give).
`MOVE`**:** Movement of a body part by its owner.
`ATTEND`**:** Focusing of a sense organ toward a stimulus (e.g. listen).

The subject of the sentences is represented by `S` which is a role and con be instatiated by any agent. The script consists out of four scenes:

Scene 1: Entering: `S PTRANS S` into restaurant, `S ATTEND` eyes to tables, `S MBUILD` where to sit, `S PTRANS S` to table, `S MOVE S` to sitting position.
Scene 2: Ordering: `S PTRANS` menu to `S` (menu already on table), `S MBUILD` choice of food, `S MTRANS` signal to waiter, waiter `PTRANS` to table, `S MTRANS` 'I want food' to waiter, waiter `PTRANS` to cook.
Scene 3: Eating: Cook `ATRANS` food to waiter, waiter `PTRANS` food to `S`, `S INGEST` food.
Scene 4: Exiting: waiter `MOVE` write check, waiter `PTRANS` to `S`, waiter `ATRANS` check to `S`, `S ATRANS` money to waiter, `S PTRANS` out of restaurant.

It is not compulsory to adopt the concept of conceptual dependencies to utilize scripts. Nevertheless, it is necessary to define entities, roles and operations (i. e. actions) in a way, that offers some generality and transferability. Thus, a set of universals with predefined semantics and support for roles seems to be helpful.[7]

[7] Further work will include an analysis how ongoing efforts on semantic modeling could be integrated in this approach (cf. [16]).

3.4 Semantic Expressiveness

As sketched above, a script transforms one state of affairs into another state. States may be modeled adequately with a fragment of first order logic (existential quantified, conjunctive connected predicates). Accordingly, entities, e. g. objects or actors, that are referred to in a script may be formalized as well by a set of attributes, i. e. as predicates. Generally, first order logic is not sufficient to model the dynamic interdependencies and actions in a domain, due to the necessity of modal, temporal or second order (quantification about predicates) constructs. These language constructs have to be provided by modeling the actions and roles accordingly. Thus, type restrictions and quantifications on predicates or attributes have to be considered in the process of defining and implementing roles.

Consequently, the static aspects (situations as constellations of entities at a given point in time) are modeled with a fragment of first order logic. Actions, as well as operations on entities, permit more advanced constructs like quantification on predicates and additional qualifiers. This augments the expressiveness of the whole formalism considerably. A script forms a context in which the semantics for at least one - and to avoid misunderstandings: exactly one - valid assignment and interpretation is provided.

This approach is computationally feasible, since the domain of the variables of the predicates are restricted in most application scenarios to reasonably sized sets of possible instantiations.

3.5 Scripts and Ontologies

Generally, a software system is intended to be representationally and inferentially adequate with respect to its application area. Thus, the entities in the software system and their real world counterparts shall be describable by the same attributes. Inferences over attributes and entities in the system shall hold in reality and vice versa. Thus, the conceptualization of the application domain in the software system shall model those entities, relationships and processes that are essential for achieving the intended level of adequacy. Such a conceptualization may be referred to as an ontology [17]. In this setting the role of an ontology is twofold. It shall represent the body of knowledge from which the deliverables of the system shall be derived and it shall provide the vocabulary and the rules from which the interactions with the system may be described (cf. [9]).

The ontology has to be dynamic: the project goals may be subject to change and therefore the underlying ontology respectively. Since a project is an unique endeavor we can not take some ontology from the shelf, but it is more likely that each project (even if located roughly in the same domain) will need its own ontology.

Such an ontology will be referred to as an *agreed* ontology to express that it shall represent the common understanding of the domain by all stakeholders of the project. The term *agreed* implies a certain dynamics as well in the process

of defining and refining the ontology. It reflects the processes as mutual understanding as the project group grows and implies that the formalism should be mighty enough to tackle well known problems with respect to knowledge bases (frame problem, non-monotone logic, etc.).

3.6 The Building Blocks of an Agreed Ontology

Accordingly, there have to be building mechanisms for both: scripts as constituents of an ontolgy and the ontology itself. Given the inherent structure of a script as outlined above it lies at hand that scripts can be defined by a context-free grammar. Basically, a context-free grammar has four components (cf. [18]):

- A set of terminal symbols. These are the elementary symbols of the language defined by this grammar.
- A set of nonterminal symbols or syntactic variables.
- A set of rules, where each rule consists of a nonterminal (head) and a sequence of terminals or nonterminals (body), by which the head may be replaced.
- A designation of one of the nonterminals as start symbol.

Applied to this context, it is evident that scenes, roles and actions form the nonterminal symbols of this grammar. The terminal symbols are either domains of the syntactic variables, such as instantiations of roles (i. e. a specific user or a specific entity) or outcomes of a script, i. e. a state of affairs.

This shall be illustrated with a scenario where a device fails, and calls for a technician. This scenario is taken from a setting that has been accomplished by the Deutsche Telekom Laboratories [19]. It is situated in a context where machine-to-machine techniques are deployed to automate facility management processes. This scenario is built up from several scenes. The scenes of the scenario have to be applied in a distinct order, thus the scenes are numbered.

1. A device fails. A notification is sent calling for the technician. His credentials are activated, so that he may enter the room where the device is located. A process is triggered which waits for the technician. If the technician has not arrived during an interval, the call is sent again.
2. The technician arrives at the building. The technician has to authorize himself with his credentials to be able to enter the room where the device is located. This will trigger another event. This event includes a success parameter, stating whether the door opened or not. This can generate an alarm, should the technician enter the wrong credentials.
3. During the repair process the device has to be queried several times. Since the technician is authorized, the conditions for a repair process are met and no further call is sent.
4. Once the device works again and the technician is finished the authorization lifespan will be ended. Again, an alarm is triggered if the technician fails to authenticate himself when leaving the location. The credentials are deactivated to ensure that the technician may enter the location only in the course of a repairing process.

The scenario is comprised of several actions that result in specific situations, i.e. events. The scenario is represented with a grammar in Backus-Naur form as given below. The terminal symbols are enclosed with quotes. Head and tails of the rules are separated by a '::=', ',' is used as concatenation and ';' as termination symbol.

```
 1. accessAndServiceControl::=queryStateOfDevice;
 2. queryStateOfDevice::= device, deviceWorks;
 3. queryStateOfDevice::= device, deviceFails, isAuthorized,
    queryStateOfDevice;
 4. queryStateOfDevice::= device, deviceFails, callTechnician,
    waitingForTechnician;
 5. device ::= "specificDevice";
 6. deviceWorks ::= "deviceWorks";
 7. deviceFails ::= "deviceFails";
 8. callTechnician::= technician, notifyTechnician;
 9. technician ::= "specificTechnician";
10. notifyTechnician ::= "technicianNotified";
11. setCredentials::= "techniciansCredentialsActivated";
12. unsetCredentials ::= "techniciansCredentialsDeactivated";
13. waitingForTechnician::= openDoor;
14. waitingForTechnician::= callTechnician, waitingForTechnician;
15. openDoor::= arrivesTechnician, setCredentials, authenticate,
    isAuthorized;
16. soundAlarm::= setCredentials, authenticate, isNotAuthorized;
17. arrivesTechnician= "technicianHasArrived";
18. authenticate::= credentials, isAuthorized;
19. authenticate::= invalidInput, isNotAuthorized;
20. credentials ::= "credentials";
21. invalidInput::= "invalidInput";
22. isAuthorized ::= "isAuthorized";
23. isNotAuthorized ::= "isNotAuthorized";
24. repairDevice ::= queryStateOfDevice, leaveRoom;
25. leaveRoom::= authenticate, unsetCredentials;
```

The terminal symbols in this example are placeholders for specific instantiations of roles, e. g. a specific technician, location or device, or situations. A situation s is a configuration (cf. sec. 3.4) which holds true for all state of affairs Δ in the system at a given point t in time.[8] Thus: $\Delta_t \vDash s \wedge \Delta_t \nvDash \neg s$. It surely is a challenge to model the action and role part in a way that supports explanations and allows quantifications. Nevertheless, although the exact definition and modeling of actions and roles may be demanding in specific cases, it is considered that this does not present an obstacle in principle to this approach. Since there are several approaches documented and available, where this problem has been solved (c.f. [11]).

[8] For example, Δ_t may be the set of all relations in a database system at a specific point of time t.

3.7 Constructing Agreed Ontologies

Scripts form the nonterminal vocabulary of an agreed ontoloy, the terminals are representations of outcomes of scripts. The challenge to define the rules for the ontology is quite demanding, since they model the order and interdependencies of scripts and the dynamics of the ontology depends on them.

The dynamics is achieved by adding new scripts to the ontology, removing and modifying existing scripts or changing the order of the scripts. This shall be sketched by extending the service scenario given above with further use cases from that domain. [9]

Access and Service Control: Maintenance personnel are given key (e. g. RFID tags, access cards) for accessing facilities and identification at devices to be maintained. Tags store employee credentials, doors to be used, work orders as well as operations carried out. The usage of doors is monitored and alerts are generated if needed.

Inventory Management: Every asset may have one or more unique identifier. This provides knowledge of the connected devices, their functionalities, and attributes. Automatic inventory of assets using fixed and handheld readers helps locating displaced and mobile assets. Absence of a reading event can be used to detect stolen equipment.

Predictive Maintenance: Continuous monitoring of operational (e.g. load) and non-operational (e.g. temperature) parameters using sensors to predict breakdowns. Estimate individual maintenance intervals for different equipments. Using maintenance history (data logging) to analyze tradeoffs between cost to maintain old equipment and investment in new equipment.

Remote Control: Remotely monitor and query about the status of individual persons (in terms of location) and devices. Devices shall be reconfigured remotely.

The script for access and service control was explained in detail in sec. 3.6. The other use cases may be represented as scripts in analogous way.[10] A simple example grammar that models an ontology for this facillity management setting is given below:[11]

```
1. FacilityManagementOntology::= InventoryManagement, PredictiveMaintenance,
   RemoteControl, AccessAndServiceControl;
2. InventoryManagement ::= establishedAutoID;
3. PredictiveMaintenance ::= establishedPMProcesses, remoteControl;
4. AccessAndServiceControl ::= establishedAASProcesses, remoteControl;
5. RemoteControl::= "establishedRemoteControl";
```

[9] Additionally, it might be necessary to ensure global consistency of the outcomes of the different scripts. The approach of assumption based truth maintenance systems (ATMS) [20] can be deployed to solve this problem. A script of this approach roughly plays the role of an assumption in de Kleer's concept.

[10] In this context use cases are subsumed by scripts.

[11] The grammar is represented in BNF, terminal symbols are enclosed with quotes.

It is obvious that the scenarios are not independent from each other but imply an order. Thus, to enable predictive maintenance remote monitoring has to be established. The granularity can be increased, by splitting the scenario of access and service control in two scenarios: an access control and a service control scenario. This leads to a change in the definition of the inventory management process:

```
1. AccessAndServiceControl ::= remoteControl, AccessControl;
2. AccessControl ::= "establishedACProcesses";
3. InventoryManagement::= AccessControl, establishedAutoId;
```

It is possible to change a sequence or to edit the starting conditions or the outcome of a script. Thus, by manipulating the grammar the ontology of the domain of interest can be managed. In addition an ATMS could administer the consistency of the general state of affairs of the model of the application domain. Since each script affects the state of affairs of the whole system and the changes that have been applied by the operations of a script are recorded, each state of affairs can be tracked down to the scripts involved. Thus, it may be easily discovered, if starting conditions of a script never appear or if scripts lead to inconsistencies.

Since scripts not only model the interaction flow but are more detailed with respect to the semantics of the process and the outcome, scripts may even act as blue print for test cases. Consequently, this approach enables to transfer the methodology of test driven development to the requirement engineering process. As production code in this setting has to pass the predefined test cases, new requirements would have to be formulated as scripts and checked against the so far agreed ontology. Consequently, it can be decided for each requirement (that is well-formed in terms of the ontology's grammar) whether it may be derived, is subsumed, leads to contradictions or augments the set of requirements gathered so far.

4 Conclusion

In this paper a position was formulated that points out some general deficits in requirements engineering. It was argued that the tools to represent functional requirements of a non-trivial software system are commonly restricted in their semantic expressiveness and thus inept to establish mutual understanding concerning those processes. Misunderstandings will easily arise due to the ill defined semantics of the representation formalism: each stakeholder will underlay his or her individual semantic reference system for understanding. To describe the system‘s functional requirements in a comprehensive way in natural language is possible in theory but not feasible either. It is known from experience that large volumes are scarcely read by their target audience.

Thus, it is argued that this problem can be solved by an agreed ontology that models the understanding of the target system in a way that explanations on functional requirements can be given. This ontology should be implemented in a system that forms the semantic grounding of all assumptions about the

domain of interest and the interactions within the future software system. Misunderstandings and contradictions can be managed due to its semantic-enabled constituents, i. e. scripts, and the internal management of the system.

5 Consequences and Further Work

Although the argumentation in this paper did not provide detailed examples, it should be comprehensible, that this approach is technically feasible and will hold true. Future work will provide comprehensive examples and flesh out the approach.

Nevertheless, applying this approach to software projects will result in a major change in the administrative setting of a project and communication between the stakeholders that will hinder its acceptance. The biggest issue in that respect is that presently, the documentation of all relevant organisational stipulations is essentially paper-based. Utilizing this approach would mean to transfer an essential part of the project documentation, i. e. the written and signed requirements specification, to a different medium. The legal issues are no hindrance, since a system that implements this approach could be serialized and signed with the certificates of the stakeholders. Nevertheless, major shifts in administrative procedures are not done lightly. Thus, future work will have to prove that the benefit of this approach to software engineering will outweigh the inconvenience of changing an administrative process.

References

1. Project Management Institute, ed.: A guide to the project management body of knowledge: PMBOK guide. 3 edn. Project Management Institute, Inc. (2004)
2. Knauss, E.: Einsatz computergestützter Kritiken für Anforderungen. Softwaretechnik-Trends **27** (2007)
3. Wiegers, K.: Software Requirements. Microsoft Press (2005)
4. Jeckle, M., Rupp, C., Zengler, B., Queins, S., Hahn, J.: Uml 2.0 - Neue Möglichkeiten und alte Probleme. Informatik Spektrum **27** (2004) 323 – 332
5. Nalepa, G., Wojnicki, I.: Using UML for Knowledge Engineering - A Critical Overview. In Baumeister, J., Seipel, D., eds.: 3rd Workshop on Knowledge Engineering and Software Engineering (KESE 2007) at the 30th Annual German Conference on Artificial Intelligence. (2007) 37 – 47
6. Woods, W.: What's in a Link: Foundations for Semantic Networks. In Borow, D., Collins, A., eds.: Representation and Understanding. Academic-Press, New York (1975) 36–81
7. Sutcliffe, A.: Scenario-based requirements analysis. Requirements Engineering Journal **3** (1998) 48 – 65
8. Allmann, C.: Situations- und szenariobasierte Entwicklung von Anforderungen in der technischen Entwicklung. Softwaretechnik-Trends **28** (2008)
9. Walton, D.: Can Argumentation Help AI to Understand Explanation. Künstliche Intelligenz **2** (2008) 8 – 11
10. Richter, M.: Logik versus Approximation. Künstliche Intelligenz **4** (2004) 62 – 64

11. Schank, R., Kaas, A., Riesbeck, C.: Inside case-based Explanation. Lawrence Erlbaum Associates, Inc. (1994)
12. Peylo, C.: Wissen und Wissensvermittlung im Kontext von internetbasierten intelligenten Lehr- und Lernumgebungen. Volume 257 of Dissertationen zur künstlichen Intelligenz. Akad. Verl.- Ges. Aka, Berlin (2002)
13. Schank, R., Abelson, R.: Scripts, Plans, Goals and Understanding. Lawrence Erlbaum Associates, Hilsdale, New Jersey (1977)
14. Pohl, K., Sikora, E.: The Co-Development of System Requirements and Functional Architecture. In Krogstie, J., Opdahl, A., Brinkkemper, S., eds.: Conceptual Modelling in Information Systems Engineering. Springer, Berlin, Heidelberg, New York (2007)
15. Bramsiepe, N., Sikora, E., K.Pohl: Ableitung von Systemfunktionen aus Zielen und Szenarien. Softwaretechnik-Trends **28** (2008)
16. Colomb, R.: Ontology and the Semantic Web. IOS Press, Amsterdam (2007)
17. Guarino, N.: Formal Ontology and Information Systems. In Guarino, N., ed.: Formal Ontology in Information Systems. Proceedings of the First International Conference, June 6-8, Trento, Italy, Amsterdam, Berlin, Oxford, Tokyo, Washington, IOS Press (1998) 3–19
18. Aho, A., Lam, M., Sethi, R., Ullman, J.: Compilers, Principles, Techniques, & Tools. Addison-Wesley, Reading, Massachussets (2007)
19. Krishnamurthy, S., Anson, O., Sapir, L., Glezer, C., Rois, M., Shub, H., Schlöder, K.: Automation of Facility Management Processes using Machine-to-Machine Technologies. In: The Internet of Things. Volume 4952 of LNCS. Springer, Berlin, Heidelberg, New York (2008) 68 – 86
20. de Kleer, J.: An assumption based truth maintenance system. Artificial Intelligence (1986)