On the Definition of Self-service Systems

Corentin Burnay[1,2,3], Joseph Gillain[2,3], Ivan J. Jureta[1,2,3], and Stéphane Faulkner[2,3]

[1] Fonds de la Recherche Scientifique – FNRS, Brussels
[2] Department of Business Administration, University of Namur
[3] PReCISE Research Center, University of Namur

Abstract. Changing requirements are common in today's organizations, and have been a central concern in Requirements Engineering (RE). Over time, methods have been developed to deal with such variability. Yet, the latter often require considerable amount of time to be applied. As time-to-value is becoming a critical requirement of users, new types of systems have been developed to deal more efficiently with changing requirements: the Self-Service Systems. In this paper, we provide an overall discussion about what self systems are, what they imply in terms of engineering, how they can be designed, and what type of questions they raise to RE.

1 Introduction

Information Systems (IS) are becoming central in contemporary organizations, given that they intervene in most, if not all value chain activities. They do so by helping organizations to collect, treat and diffuse data and information, thereby enabling them to know more about, and hopefully influence more precisely and relevantly the business activities. The engineering of IS is the responsibility of systems engineers, which today involves various professions, including requirements engineers (or business analysts), software engineers, system architects, etc. They are expected to define clear and relevant requirements, then engineer, develop, deploy, maintain, and change solutions to satisfy the requirements.

It is well-known that it is hard to get the requirements right [1]. Part of the difficulty comes from the fact that requirements that are available *before* using a system or interacting with its prototype and *after* using it, can, and often are different. The difference comes from various factors, including simply that people learn through experience. Through learning, their expectations change, and so do their requirements. Such issue, of changing requirements, has been a topic of research in adaptive systems and requirements evolution [2].

In this paper, we are interested in one specific approach to deal with requirements changes. We observed it in industry, and our aim is to look at the theoretical issues that arise in relation to it, how it relates to research in Requirements Engineering (RE), and what new issues, if any, it gives rise to.

Specifically, we are interested in the so-called Self-Service Systems. The basic idea is that the system engineers will not produce a system that satisfies all the specific requirements that the various stakeholders may have. Instead, they will

M. Indulska and S. Purao (Eds.): ER Workshops 2014, LNCS 8823, pp. 107–116, 2014.

engineer a system which can, if properly configured *by the users*, satisfy these requirements, whichever they are at system design time, and whichever they may end up being at run-time. If you think of requirements as being represented, as usual in goal-oriented requirements engineering [3,4], as a goal forest, where goals close to roots are abstract and general requirements, and goals close to leaves are more specific requirements, then Self-Service Systems are engineered to satisfy some goals midway in the goal forest, between the leaves and the roots, rather than satisfy the leaves. As we will see later, this view is simplistic, but it gives an initial idea that we build from in the rest of the paper.

We refer to Self-Service Systems as *Selfs* hereafter. We first motivate the use of Selfs in Section 2 and 3. We then provide a more detailed definition of what Selfs are, with examples in Section 4. We discuss how requirements from traditional systems differ from those from Selfs in Section 5. Finally, we suggest that designing a Self raises new challenges for RE in Section 6. We conclude with a discussion about tradeoffs and future works respectively in Section 7 and 8.

2 Illustration - Self-service in Business Intelligence

We observed Self-Service Systems for Business Intelligence (BI). In general, IS for BI gather business data in order to provide information to business decision-makers, under the form of reports, dashboards, or any other relevant output [5,6]. Despite there being frameworks for doing RE of BI systems [7,8], there is a practical difficulty that is hard to overcome in terms of methodology. It can be formulated as follows: the data types and sources, and the information relevant for business decision-making may not be the same at all times, which means that, for example, the content of reports, the analyses applied to data, and so on, need to be changed regularly. In practice, it may be too costly (or take too much time) to have business analysts elicit new requirements on reports and analyses, and propagate these new requirements through the specifications, system architects to architecture, and software engineers to code each time a change occurs.

From the standpoint of those who make and sell BI systems, it may also be more interesting to avoid changing the systems so much, because, for example, it means maintaining systems that become different from each other over time, even if they started from the same set of functionality. Self-Service Systems are a response to this, in that they do not try to satisfy the most specific requirements, but give features whose combinations could satisfy various specific requirements. For example, a Self-Service BI (SSBI) system will have features that allow its users (they can include business analysts as well, not only end-users) to change analyses applied to data, create new reports or change existing ones, and so on.

In [9], SSBI is presented as an important promise of BI, *despite the current difficulties in making those softwares easy to use for business people.* SSBI has been the center of some attention from specialized institutions (e.g. TDWI [10], Gartner [11] or Forrester [12]), which is another clue that business users are actually interested in achieving shorter time-to-value and doing BI on their own.

3 Why Make Self-service Systems?

Selfs are ways to deal with the change of requirements. Instead of making a system that satisfies exactly all of the most specific requirements identified at design-time, we consider these merely as examples of requirements that may arise at run-time, and we engineer features whose combinations can satisfy these anticipated, and perhaps some of the unanticipated run-time requirements.

In fact, the problem with design-time requirements is not that they are changing. In practice, many RE approaches exist, that can be used to identify new requirements [13] or track evolution of existing ones [14,15]. The translation of those requirements into specifications for the system-to-be is not a problem either, as it has also been discussed at length in RE [16]. *The problem is rather on the time it takes to go from there being a new run-time requirement, to the time when the system has been changed to satisfy it.*

In the case of BI systems, the IT department is expected to be very responsive, and to provide quickly adapted solutions to business requirements. This is a generic requirement from BI systems, also known as time-to-value: once a BI system is implemented, users must gain easy and rapid access to information, so that decision making process remains efficient [10]. Traditional RE approaches can appear limited in that regard, because they assume elicitation, modeling, analysis, verification, negotiation, validation, and so on, have to be done for new requirements. This can influence time-to-value negatively.

As a response to delays due to *changing requirements*, Selfs attempt to transfer some of the design responsibility to end-users, who are therefore in charge of understanding what they need, and directly designing what is required to satisfy these needs using a Self. For example, in BI, Self-Service Business Intelligence (SSBI) is used to enable end-users to select some data sources and decide about a visualization tool to view it, all by themselves. Ultimately, they choose what to include in the report that they need the system to make.

Unlike for classical BI systems, SSBI does not require the usual RE processes to occur each time a user has a new requirement. There is therefore a tendency to make systems with generic features, and expect users to combine / configure the latter by themselves, in order to satisfy their new, specific requirements on-demand. They do so with the support of the system, but without the intervention of system engineers. This approach has the main advantage of reducing time-to-value, and hence improving users experience of the BI platform [10].

4 Indirect Requirements Satisfaction

In this section, we suggest and discuss an essential property held by typical requirements of Selfs. We consider that any requirement which satisfies this property can be characterized as a *Self requirement*. The more a system is specified by such Selfs requirements, the more the system can be considered as a Self.

It implies that the distinction between Selfs and non Selfs is not clear cut, and that any system can be placed on a "Self dimension". Besides, Selfs can be

distributed or centralized, adaptive or not, agent-based or otherwise, and so on - all such system categorizations are orthogonal to the Self dimension.

To introduce the property in question, it is important to distinguish users from other stakeholders of a Self. The reason lies in the difference between how a Self can satisfy user requirements, and how it can satisfy those of other stakeholders. Just as any system, a Self may satisfy the requirement of a stakeholder who is not a user, e.g., a stakeholder may not be involved in using the system, but may require that the license to use the Self costs below some amount per year. If the annual license costs X, and is smaller than Y (the maximal amount that this stakeholder set), then the system satisfies the requirement. We say that the requirement is *directly* satisfied, *since this stakeholder does not need to invest any more effort in using the system, in order to satisfy that requirement.*

The story is different for most users of Selfs, i.e., individuals who will interact with the system at run-time, precisely in order to satisfy their own requirements. If a system directly satisfies the requirements of its users, then it does not belong to Selfs. It does belong to Selfs, if it satisfies these requirements *indirectly.* We say that a system satisfies a requirement indirectly, if there exists a scenario for using that system, such that if the system is so used, then the requirement will be satisfied, *but that is not the only possible scenario for using that system, and that scenario is not necessarily known by either the users or the system designers.*

It looks like quite a lot of software only indirectly satisfies their users' requirements. An operating system satisfies indirectly a user's requirement to print out a document, if the user needs to find, install, and configure by herself a printer driver. A word processor indirectly satisfies the requirement to format a text according to some formatting guidelines, because it is the user who has to figure out how to use headers, footers, front pages, blank pages, and so on, in order to ensure that the document does indeed follow the guidelines. A web browser, in contrast, looks to be directly satisfying its main requirement, which is to display content on the World Wide Web. But it indirectly satisfies the requirement to play specific kind of video files, if the user has first to find, download and install the relevant plugin.The usual calendar applications satisfy directly the requirements to add events and reminders, invite people to join events, and such. But they only indirectly satisfy the requirement to find the slots that suit everyone, when organizing a meeting. To satisfy that requirement, the user has to find some clever way to use the various existing features in her calendar application.

It is an essential property of Selfs that *the intention in designing them is to satisfy many user requirements indirectly.* The consequence is that a user will have to do the work of finding the appropriate scenario that mobilizes the features of the system, in a way which will satisfy this user's requirements. The scenario should not already built into the system in such a way that a user can, without much effort, activate it. If the intention in designing a system is primarily to enable the indirect satisfaction of many requirements, which were anticipated or unanticipated at design-time, we will say that the system is *undetermined.* We will say that the system is *determined*, if the intention in designing it is to satisfy exactly some specific set of requirements identified at design-time.

Table 1. Examples of Software

	Determined	Undertermined
Business Users	Google Calendar, Microsoft Outlook, Microsoft Word, Skype	Blogger, Joomla, Matlab, QlikView, Excel PowerPivot/View
IT Experts	Visual Paradigm, FileZilla, AVG Anti-Virus, Apache Web Server	MicroStrategy, Pentaho, Symfony Framework, .NET Framework, Java

5 Requirements from Selfs and Non-selfs

In order to analyze the way RE happens in the particular case of Selfs, we first suggest to distinguish between two types of requirements. Then, using this distinction, we provide a more accurate, RE-oriented, definition of Selfs.

We start by distinguishing two kinds of requirements, called Stakeholder Requirements (R_{Stk}) and Derived Stakeholder Requirements (R_{DS}). As their name suggests, we obtain R_{Stk} from stakeholders and through requirements elicitation, which may involve interviews, observation, documentation analysis, and so on. R_{DS} are the requirements that a requirements engineer defines herself, on the basis of R_{Stk}. R_{DS} are made, for example, by refining, decomposing, disambiguating, or otherwise manipulating R_{Stk}, in the aim of identifying such R_{DS} which are operational. A requirement is operational when there is a specification which can be implemented (that is, its implementation is judged feasible), and there are good reasons to believe that, if implemented according to that specification, the resulting system will satisfy that R_{DS}. Various frameworks exist to identify variability in goal models, and could be of particular interest in the identification of R_{DS} [17,18]. We will say that R_{Stk} and R_{DS} together are *Ground Requirements* (R^G), i.e, R^G is the union of R_{Stk} and R_{DS}.

We then distinguish R^G from *Self Requirements* (R^{Self}), which are defined by requirements engineers, in order to ensure that the system has generic features, whose various combinations could satisfy potentially many R^G. For example, the user of a BI system can have the requirement r_1 to "Display average sales margin per product". Such requirement has been elicited explicitly from that business user (through, for example, an interview): r_1 is therefore part of R_{Stk}, and hence of R^G. That requirement provides some direction to derive requirement variants. For instance, r_2, "Display sales revenue repartition per vendors", can be derived from r_1 and is then part of R_{DS}, hence also of R^G.

Selfs are, however, not made to satisfy specifically R^G. Instead, they are made to satisfy a requirement r_3, which is "Be able to *[displaying]* an *[arithmetic function]* over one *[business fact]* for one *[business dimension]*". The latter requirement is such that, if it is satisfied, then both r_1 and r_2 will be, but also potentially many others similar to r_1 and r_2. The requirement r_3 is in R^{Self}. The difference between r_1 and r_3, and between r_2 and r_3, is that r_3 is obtained by looking at r_1 and r_2, and finding what is common to them, in order to formulate a new requirement which, if satisfied, would lead us to conclude that both r_1 and r_2 are satisfiable, provided that the users find out how by themselves.

6 Requirement Engineering for Selfs

6.1 Selfs vs Non-self: An Illustration

Previous RE definition of Selfs offers a support for distinguishing between the design concerns of Selfs and non-Selfs. Consider previous example r_1, where a user wants to "Display average sales margin by product". To obtain such result, the user can use a BI solution, and has two alternatives, called A and B below.

Alternative A: she could use a classical BI system. In that case, she would have to ask the IT to design a report, which shows the average margin by product. This results in the stakeholder requirement r_1 which is part of R^G. One could model that requirement via a goal model such as in Figure 1. With R^G, the IT could decide about the design of a new report, with no room for self-service: user might simply need to select a product to obtain the information she needs. Here, there are no features to select: everything is decided for the user in advance, so that the system can be said to be Determined.

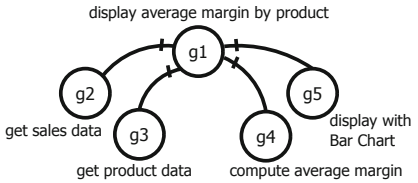

Fig. 1. Goal refinement of r_1

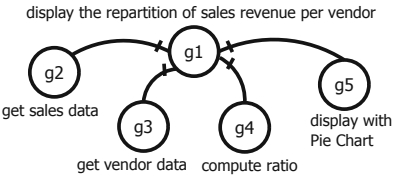

Fig. 2. Goal refinement of r_2

As discussed in our Introduction section, the requirements from a BI solution are likely to change rapidly. Previous user could for instance have the new requirement r_2, illustrated with a goal model in Figure 2. To be achieved, that goal would require a new round of elicitation and operationalization, as r_2 would be added to R^G. Based on that new R^G, IT would have to design a new report. This repetitive RE process can increase time-to-value.

Alternative B: The user has access to an SSBI solution, and she is responsible for satisfying her requirements. Suppose that this is a simple spreadsheet software, such as Excel. Let there be a spreadsheet, which satisfies r_3 mentioned earlier. Starting with her first requirement r_1 (Figure 1), the stakeholder could for example select some rows from a data set she judged relevant, sum the cells and divide the result by the count of rows. She could also select all the data, apply a filter to it to keep only the last six months, and compute the average using the function for computing the mean, and so on. This system is Undetermined, as it is up to the stakeholder to find and design a solution to her problem. If a new requirement arises, let's say r_2, there is no need to re-engineer the SSBI solution. The user, or someone helping her, would simply adapt some part of her initial solution to design a new solution that satisfies the new requirement.

6.2 A RE Process Adapted for Selfs

From an RE perspective, Alternative A and B imply different design approaches. This is illustrated in Figure 3. In Alternative A, engineers have to decide about a specification that satisfies the Ground Requirements they elicit from business users. Only R^G is used to design the system-to-be. If R^G changes (due for example to a new variant of a requirement), then engineers must redesign the existing software to satisfy that new set of requirements. In Alternative B, engineers must identify user requirements, and then try to anticipate any other possible requirement. This results in a set of generic requirements R^{Self}. The design based on R^{Self} must offer sufficient features for the user to satisfy by herself the requirements that may appear at run-time in R^G.

Actually, the design of a Self cannot work on R^G since operationalizing R^G would consist in delivering a determined system providing business users with all required features in a single design. It is illustrated in the goal model GV in Figure 4. Identifying Selfs Requirements is more than only taking into account of all possible variability in users requirements. Although possible Self configuration would be able to eventually operationalize each leaf node of GV, a system which directly operationalize all leaf nodes of GV is not a Self. From there on, non-Self systems build from the operationalization of requirements in R^G, while Self systems build from the operationalization of requirements in R^{Self}.

Nonetheless, setting up R^{Self} from R^G is currently still a research challenge. Current methodologies only focus on R^G, i.e. how to gather and model R_{Stk} as well as how to derive R_{SD}. To the best of our knowledge, little attention has been paid as to how R^{Self} can be *abstracted* from R^G.

Notice that Self requirements open the way to some unanticipated uses of the system. Consider the case of MS Word. Word proposes to its user a mailing functionality, in which users are capable of selecting themselves fields, displaying the latter on a form, defining the layout for these fields, etc. Word is therefore

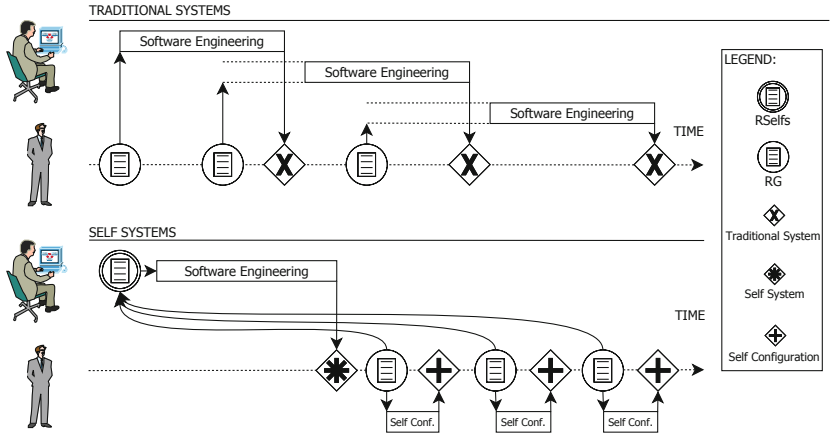

Fig. 3. Comparison of RE process for traditional and Self systems

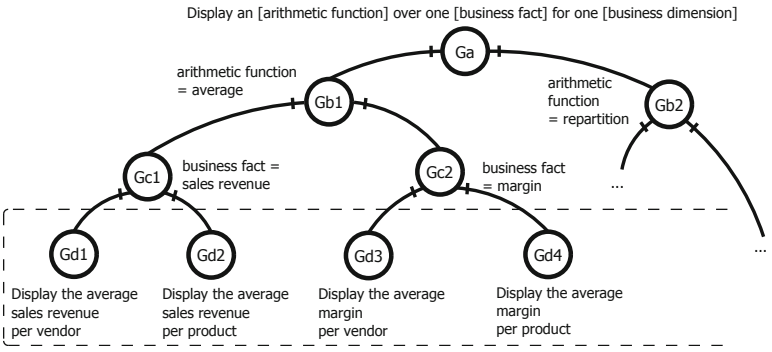

Fig. 4. The goal model GV

somewhere on the Self dimension between pure Selfs and pure non-Selfs, because it satisfies at least one Self requirement, i.e., the user is able to define a mail on her own. A side effect of being a Self system is that users may be creative in using the software, e.g., defining a mailing is not the only way to use the Word functionality. In practice, it could be used in some other, unanticipated, ways.

Imagine for example a professor who wants to create several exercises for her students. Each exercise will be the same except some values and some words which will be changed. For this purpose, she could use the mailing functionality to design a template with print labels, and generate several exercises by giving different values for each label. In this context, she used an Undetermined functionality of Word in order to design her *own solution*. The use of a Self therefore depends on the creativity of its users, and should not be limited to the use cases for which it was initially designed.

7 Challenges of Selfs for Requirements Engineering

Although critical to ensure a Self anticipates as much as feasible of future stakeholders' requirements, the identification of R^{Self} based on R^G (using, for example, an abstraction mechanism) presents some risks. Using the full set of R^{Self} to decide about the specification of a Self can lead to systems with numerous features, and hence to relatively complex Selfs. By complex, we mean that they provide many features to end-users. The problem with the complexity of a Self is that it can be negatively correlated with its usability: end-users, who have relatively low (sometimes no) IT background, may not be capable of understanding and combining the features consistently to build custom solutions. That problem of complexity is typical of Selfs: regular systems avoid such complexity by directly satisfying the specific R^G.

Consider again the requirement of a user who wants to "Display average margin per product". Imagine the user has to satisfy that requirement using a Self. She is given a spreadsheet software such as Microsoft Excel. Excel contains Self requirements because the user is in charge of designing its own solution to compute her average margin from a range of data. In Microsoft Excel, she may have

the choice between five, maybe six, features (average, count, sum functions, etc.) to be combined in order to compute an average margin. Imagine the same user is exposed to a new, more complex system, with hundreds of features that could be used to compute that same result. That system would be less usable for the business user. There would be risks that the user gets lost, or uses inappropriately some features of the software, with the ultimate risk that this business user does not satisfy properly her requirements.

This threat is important, and is reflected in existing SSBI solutions, where users are often discouraged because the Self is too complex for them. For example, Weber emphasized that "In an effort to give users what they want, IT sometimes errs on the side of giving users everything" which he claims is a typical problem of SSBI system [19]. SSBI experts also highlighted that "It turns out that most users found the tools too difficult to use. Even when the tools migrated from Windows to the Web, simplifying user interfaces and easing installation and maintenance burdens, it was not enough to transform BI tools from specialty software for power users to general-purpose analytical tools for everyone in the organization." [9]. In that regard, we consider there is a gap in current RE approach to Selfs: designers should not only be interested in creating systems that satisfy the set of requirements R^{Self} (such as in SSBI). They should also account for the fact that Selfs must be usable for business users. Therefore, they should pay attention to the number of feature they provide.

Note finally that research has been conducted to bring variability into software development. One of the most important research regarding variability is Software Product Line Engineering [20]. Although it aims to build a base system which can be customized to particular needs, this customization still requires IT intervention. Moreover, it does not aim to transfer the design responsibility of the users. Consequently, RE is traditionally about R^G and how to derive products which implement sub-parts of R^G. To the best of our knowledge, no research has gone on the business-user intervention in the resolution of R^G.

8 Conclusion and Future Work

In this paper, we provided an overall discussion about the use of Self-Service systems in organizations. We first discussed the rationale for such system, claiming that Selfs are valuable solutions to the problem of changing requirements and long time-to-value for business users. We defined Self-Service systems as being systems which contains operationalization of Self requirements. With such operationalization business users are in charge for configuring themselves the system in order to design their proper solution to some requirements. We then provided a deeper RE perspective on Selfs, by distinguishing between Ground Requirements, obtained requirements elicitation, and Self Requirements, which are requirements to be able to solve other, forthcoming, Ground Requirements. We concluded on a discussion about the trade-off that may appears, during RE for Selfs, between the completeness of a Self platform (in terms of features) and the usability of the latter.

References

1. Brooks Jr., F.P.: No silver bullet - essence and accidents of software engineering. Computer 20, 10–19 (1987)
2. Silva Souza, V.E., Lapouchnian, A., Robinson, W.N., Mylopoulos, J.: Awareness requirements for adaptive systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 60–69. ACM (2011)
3. Dardenne, A., Van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. Science of Computer Programming 20(1), 3–50 (1993)
4. Van Lamsweerde, A.: Goal-oriented requirements engineering: A guided tour. In: Proc. 5th IEEE International Symposium on Requirements Engineering, pp. 249–262 (2001)
5. Golfarelli, M., Rizzi, S., Cella, I.: Beyond data warehousing: what's next in business intelligence? In: Proc. 7th ACM International Workshop on Data Warehousing and OLAP, p. 1 (2004)
6. Negash, S.: Business Intelligence. Communications of the Association for Information Systems 13, 177–195 (2004)
7. Pourshahid, A., Richards, G., Amyot, D.: Toward a goal-oriented, business intelligence decision-making framework. E-Technologies: Transformation in a Connected World, 100–115 (2011)
8. Burnay, C., Jureta, I.J., Faulkner, S.: A Framework for the Operationalization of Monitoring in Business Intelligence Requirements Engineering. Software and System Modeling (SoSym) (in press)
9. Eckerson, W.W.: Performance Dashboards: Measuring, Monitoring, and Managing Your Business. John Wiley & Sons (May 2008)
10. Imhoff, C., White, C.: Self-Service: Empowering Users to Generate Insights. tech. rep., The Data Warehouse Institute, TDWI (2011)
11. Richardson, J., Schlegel, K., Sallam, R.L., Hostmann, B.: Magic quadrant for business intelligence platforms. Core Research Note ... (2008)
12. Evelson, B.: The Forrester Wave: Self-Service Business Intelligence Platforms, Q2 2012, tech. rep., Forrester (2012)
13. Zowghi, D., Coulin, C.: Requirements Elicitation: A Survey of Techniques, Approaches, and Tools. In: Engineering and Managing Software Requirements, pp. 19–46. Springer, Heidelberg (2005)
14. Zowghi, D., Offen, R.: A logical framework for modeling and reasoning about the evolution of requirements. In: Proc. 3rd IEEE International Symposium on Requirements Engineering, pp. 247–257 (1997)
15. Rolland, C., Salinesi, C., Etien, A.: Eliciting gaps in requirements change. Requirements Engineering 9(1), 1–15 (2004)
16. Van Lamsweerde, A.: Requirements engineering: from system goals to uml models to software specifications (2009)
17. Gonzales-Baixauli, B., Prado Leite, J., Mylopoulos, J.: Visual variability analysis for goal models. In: Proceedings of the 12th IEEE International Requirements Engineering Conference, pp. 198–207. IEEE (2004)
18. Liaskos, S., Lapouchnian, A., Yu, Y.: On goal-based variability acquisition and analysis. In: Proc. 14th IEEE International Conference on Requirements Engineering, pp. 79–88 (2006)
19. Weber, M.: Keys to Sustainable Self-Service Business Intelligence. Business Intelligence Journal 18, 18–24 (2013)
20. Pohl, K., Böckle, G., Van Der Linden, F.: Software product line engineering, vol. 10. Springer (2005)