Applied Ontology 3 (2017) 1–3
IOS Press

An Ontology of Requirements at Runtime

Bruno Borlini Duarte [a], Andre Luiz de Castro Leal [b], Ricardo de Almeida Falbo [a], Giancarlo Guizzardi [a]
Renata S. S. Guizzardi [a], and Vítor E. Silva Souza [a,*]

[a] *Ontology & Conceptual Modeling Research Group (NEMO) – Department of Computer Science – Federal University of Espírito Santo (UFES)*
E-mail: bruno.b.duarte@ufes.br, {falbo, gguizzardi, rguizzardi, vitorsouza}@inf.ufes.br
[b] *Department of Mathematics – Federal Rural University of Rio de Janeiro (UFRRJ)*
E-mail: andrecastr@gmail.com

Abstract. The use of *Requirements at Runtime* (RRT) is an emerging research area. Many methods and frameworks that make use of requirements models during the execution of software can be found in the literature. However, there is still a lack of a formal and explicit representation of what RRT are and what are the primary goals of their use. Still, most RRT proposals have their own modeling languages and ways to represent, specify and make use of requirements at runtime, thus resulting in a domain with overloaded concepts. In this paper, we present the Runtime Requirements Ontology (RRO), a domain ontology that intends to represent the nature and context of RRT. RRO is integrated into the Software Engineering Ontology Network (SEON), extending a general ontology about what requirements are, called Reference Software Requirements Ontology (RSRO), also presented in this paper. For developing both ontologies (RSRO and RRO), we follow a well-established Ontology Engineering method. In particular, both ontologies are grounded in the Unified Foundational Ontology (UFO) and are evaluated using verification and validation techniques.

Keywords: Requirements, Runtime, Ontology, UFO, RRO, RSRO, SEON

1. Introduction

In recent years, we have witnessed a growing interest in software systems that can monitor their environment and, if necessary, change their requirements in order to continue to fulfill their purpose (Dalpiaz et al., 2013; Souza et al., 2013a). This particular kind of software usually consists of a base system responsible for the main functionality, along with a component that monitors the base system, analyzes the data and then reacts appropriately to make sure that the system continues to execute its required functions.

There are many works in the literature that propose different solutions to this issue, such as adaptive or autonomic systems, e.g., (Huebscher and McCann, 2008; Cheng et al., 2009; de Lemos et al., 2013). We are especially interested in those that deal with this monitoring–adaptation loop using requirements models at runtime. In this context, proposals use different kinds of models and terms to represent what are the system requirements, specify what is to be monitored and prescribe how to adapt. As result, the vocabulary used by these methods is very similar, but the semantics of the entities present in their models are not always the same, thus resulting in a domain with overloaded concepts. In other words, that means that the same name could be used to identify things that are ontologically completely different, and because of that, they have different natures and identities in the real world. This is the case of the term requirement. One can refer to requirement as an artifact, described in a requirements specification; someone else can understand a requirement as an intention of a stakeholder. Although it seems that we are talking about the same entity, that is not true because they are not ontologically the same.

The construct overload problem, i.e., the same construct used to represent two or more domain entities, and other fairly common problems such as construct redundancy, i.e., a single entity is represented by two or more constructs, have motivated us to develop a domain reference ontology on the use of *Requirements at Runtime* (RRT). By *domain reference ontology* we mean a domain ontology developed with the goal of

[*]Corresponding Author: UFES - Departamento de Informática (CT7), Av. Fernando Ferrari, 514, Goiabieras, Vitória, ES, Brazil; E-mail: vitorsouza@inf.ufes.br.

making a clear and precise description of domain entities for the purposes of communication, learning and problem-solving. A reference ontology is a special kind of conceptual model, representing the consensus within a community. It is a solution-independent specification and, thus, does not take computational properties (e.g., decidability or computational tractability) into account. In other words, when developing a reference ontology, quality attributes such as precision and truthfulness to the underlying domain are never sacrificed in favor of these computational properties (Guizzardi, 2007).

The goal of this work is to provide a formal representation of the use of RRT, giving a precise description of the domain entities and establishing a common vocabulary to be used by software engineers and stakeholders within the RRT domain, including the distinctions underlying the nature of requirements during the execution of a software system.

For capturing the conceptualization underlying the use of RRT, we need to understand what requirements really are. Thus, we decided to develop two ontologies. The first, called Reference Software Requirements Ontologies (RSRO), focuses on requirements in general, their main types, and how they are documented. The second, called Runtime Requirements Ontology (RRO), extends the first, focusing on the use of RRT. Both ontologies are integrated into the Software Engineering Ontology Network (SEON) (Ruy et al., 2016). RSRO and RRO were developed following the process defined by the SABiO method (Falbo, 2014) and using UFO (Guizzardi, 2005) as foundational ontology.

To extract consensual information on the concepts present in RSRO, we conducted a study of relevant standards, such as CMMI (SEI/CMU, 2010) and SWEBOK (Bourque et al., 2014), and selected publications, including classical books on Requirements Engineering (Kotonya and Sommerville, 1998; Wohlin et al., 2005; Robertson and Robertson, 2012). For RRO, we performed an extensive Systematic Mapping of the literature (Kitchenham et al., 2010; Kitchenham and Charters, 2007) and discussed the results with a group of specialists. Finally, RSRO and RRO were evaluated by verification and validation activities, as proposed by SABiO. The verification process was based on the Competency Questions initially raised and validation was done via instantiations of the ontology based on three frameworks that propose the use of requirements at runtime, namely FLAGS (Baresi et al., 2010), ReqMon (Robinson, 2006) and *Zanshin* (Souza, 2012).

The ontologies presented here are intended to be used for conceptual clarification and terminological systematization of the existing approaches, in sub-areas like adaptive systems, requirements monitoring, context-aware systems and so on, as well as for serving as a knowledge framework for developing and reengineering RRT approaches. Furthermore, it can be used as a conceptual model for tools developed for these purposes. Operational versions of the ontologies could be implemented in languages like OWL, for a formal annotation of the models used by these proposals, providing traceability of the requirements during their life-cycle, from design-time to runtime. Lastly, the ontologies can be used for interoperability purposes, allowing frameworks that make use of requirements at runtime to interoperate.

This paper is an extended version of (Duarte et al., 2016). For this version, we integrated RRO with an ontology of requirements (RSRO) and into an ontology network (SEON). Moreover, we provide a formal characterization for the domain specific categories as well as constraints involving them. Lastly, we instantiated the ontology using other two RRT frameworks, improving its validation. Besides, based on these instantiations and the competency questions, we defined test cases for RRO.

The rest of the paper is structured as follows. Section 2 summarizes the general concepts of the RRT domain. Section 3 describes the methodology used to build RSRO and RRO. Section 4 introduces the ontological foundations for the proposed ontologies. Sections 5 and 6 present RSRO and RRO, the main contributions of this paper. Section 7 discusses how the ontologies were evaluated. Section 8 compares related work. Finally, Section 9 concludes the paper.

2. Background

This section presents the basic concepts about the use of requirements at runtime that are responsible to provide the theoretical background for the development of this work. It also presents three frameworks that propose the use of requirements at runtime: *Zanshin*, ReqMon and FLAGS. These frameworks will be used to instantiate RSRO and RRO, as a form of validation in Section 7. They were chosen because: (i) they are all present in the systematic mapping of the literature on Requirements at Runtime (RRT) that was used as a knowledge base for the ontologies; (ii) they are widely accepted by the RRT community, with many publications over the years; and (iii) because they differ in the way they treat/deal with RRT.

2.1. Requirements at Runtime

Requirement Engineering (RE) is the field of Software Engineering (SE) concerned with understanding, modeling, analyzing, negotiating, documenting, validating and managing requirements for software-based systems (Cheng and Atlee, 2007). In the RE literature, there is a strong overloading of the term *requirement* and, hence, different texts refer to requirements in different possible senses, for example, as an artifact, as an intention, as a desire or as a property of a system.

The seminal work by Zave and Jackson (1997) defines the term *requirement* as a desired property of an environment, which comprehends the software system and its surroundings, that intends to express the desires and intentions of the stakeholders concerning a given software development project. A requirement, however, can be seen as an intention or a desire when acting as a high-level requirement, or as an artifact, when documented as part of a specification. In turn, such documentation can have different levels of abstraction, for instance, depending on whether it is part of an early requirements analysis or a machine-readable artifact (file) that allows reasoning over requirements during the execution of the software. Features such as the latter have motivated research on the topic of *Requirements at Runtime* (RRT) (Bencomo et al., 2011).

For instance, Feather et al. (1998) monitor violation of requirements using Linear Temporal Logic expressions that are observed by a monitoring component at runtime in order to try and reconcile system requirements and behavior. Requirements are documented using a Goal-Oriented RE (GORE) approach for design-time purposes and as the aforementioned logical expressions for runtime purposes. The *Requirements Reflection* approach (Bencomo et al., 2010a) proposes not only that requirements be reified as runtime entities but that they keep traceability to the architectural models and all the way back to high-level goals.

Souza et al. (2013b) define a *requirement at runtime* as the classes or scripts that represent the requirements being instantiated during the execution of a program, i.e., a compiled code artifact loaded in memory to represent the fact that someone (or something) is using the system in order to satisfy a requirement. Qureshi and Perini (2010) have a similar vision, as they understand that a *requirement at runtime* is expressed by the users requests, which are captured by the software user interface and also monitored in order to check if it is in an agreement with the original specifications. More recently, Dalpiaz et al. (2013) propose, in the context of GORE, the distinction between a *Design-time Goal Model*—used to design a system—and a *Runtime Goal Model*—used to analyze a system's runtime behavior with respect to its requirements.

These and other works illustrate the diversity of concepts, models and features that have been proposed in the field of RRT, showing us that there is no consensual definition about what is a runtime requirement and that *Requirement* is a complex entity that can exist during the entire software life-cycle, since design-time until runtime. However, the use of requirements as a runtime entity is fairly recent and there is no formal representation about this domain in the literature. This lack of a formal and consensus-based description has motivated us to develop a reference ontology about this domain. The purpose is to unify an interpretation for requirements and then to characterize how it relates to other intimately connected elements in the RRT domain.

2.2. The Zanshin Framework

Zanshin (Souza, 2012; Souza et al., 2013a,b) is an RE-based framework created for the development of adaptive systems. Its main idea is to make elements of the feedback loop responsible to provide adap-

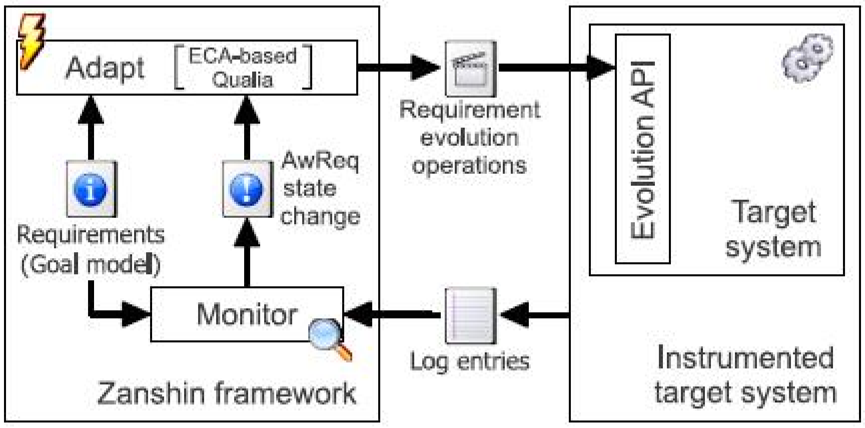

Fig. 1. *Zanshin*'s architecture representation (Souza, 2012)

tivity important entities in the requirements models used by the framework. To do that, *Zanshin* uses requirements models based on GORE (Goal-Oriented Requirements Engineering) to represent system requirements. *Zanshin*'s specific models are augmented with new elements, called Awareness Requirements (*AwReqs*) and Evolution Requirements (*EvoReqs*) that, when combined with classic GORE constructs (goals, softgoals, tasks, AND-OR refinements, and so on), are able to represent monitoring and adaptation strategies during the execution of a software system.

Figure 1 presents *Zanshin*'s basic architecture. *AwReqs* are requirements that refer to the state of other requirements of a software system at runtime. In other words, *AwReqs* are responsible for representing the parts of the system that the stakeholders want it to be able to adapt. *EvoReqs* are requirements that describe how other requirements are supposed to adapt/evolve in response to an *AwReq* failure. *EvoReqs* act directly over system requirements through a set of adaptation strategies. During runtime, they are responsible to ensure that the system is fulfilling what was previously specified by the stakeholders.

Figure 2 represents the requirements model of a Meeting Scheduler system created as a proof-of-concept for the *Zanshin* framework. *AwReqs* appear in the Meeting Scheduler requirements model as small bold circles with arrows, pointing out to the elements of the system that need to have its states monitored. *EvoReqs* are not graphically represented in the requirements model, however, they are implemented as a code artifact that will be executed by *Zanshin* when an *AwReq* fails. For instance, if, for some reason, a meeting cannot be properly scheduled in the system, AR1 (*NeverFail*, at the top-left corner of Figure 2) will trigger the *EvoReq Retry Characterize Meeting*, that will make the system perform a rollback, wait for 5 seconds, and then try to schedule the meeting again, represented in Figure 3.

2.3. ReqMon Framework

ReqMon (Robinson, 2006) is a framework for monitoring software systems requirements at runtime. ReqMon was developed with the main objective of improving the visibility of conformity policies inside a software system and to be used by system users that have permissions to receive information about the satisfaction of the system requirements. The ReqMon framework presents a requirements definition language based on KAOS (Van Lamsweerde et al., 1991), a systematic methodology for requirements elicitation and analysis to be used in design-time, and a software tool built to provide a requirements monitoring service during runtime.

Figure 4 presents ReqMon's architecture and its main components. The Event Capturer is responsible to monitor and capture, during runtime, any deviation in the requirements of the main system, i.e, the system being monitored. The Analyzer updates the status of the main system monitors and the Repository acts like a database of events, persisting a history of the monitored indicators.

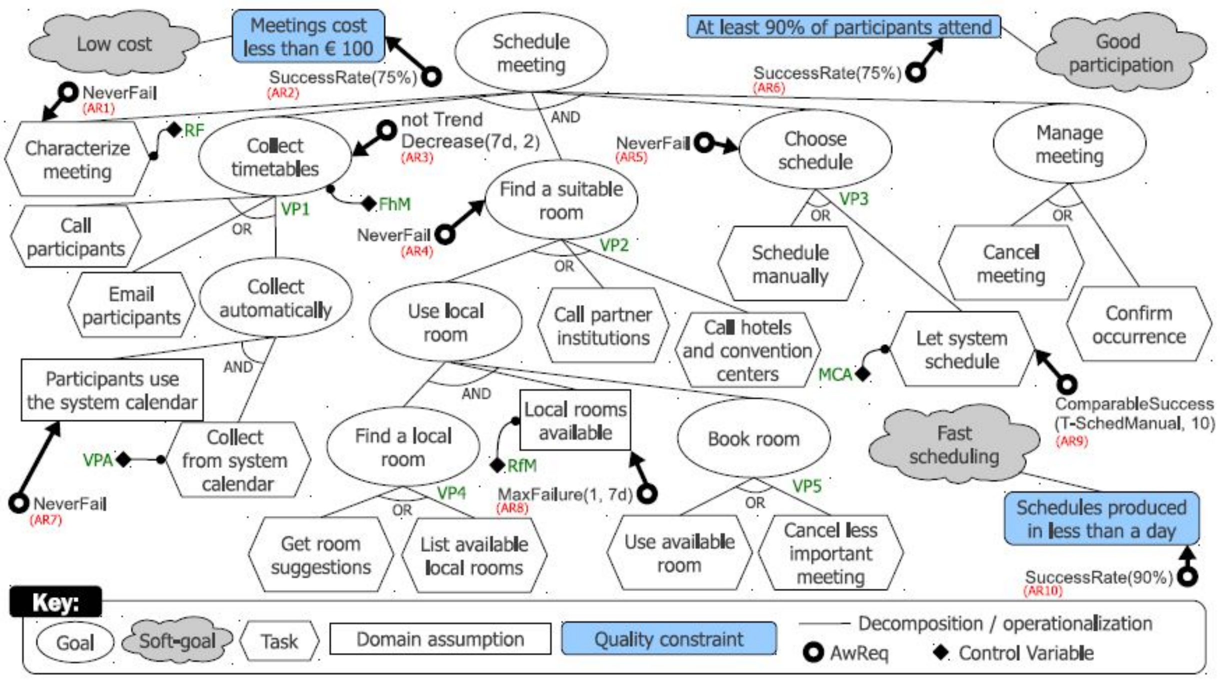

Fig. 2. Requirements model of the Meeting Scheduler, created with the *Zanshin* approach (Souza, 2012)

```
t' = new-instance(T_CharactMeet);
copy-data(t, t');
terminate(t);
rollback(t);
wait(5s);
initiate(t');
```

Fig. 3. *EvoReq* Retry Characterize Meeting after 5 seconds (Souza, 2012)

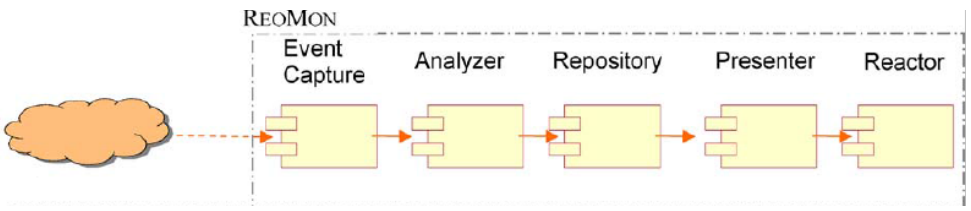

Fig. 4. ReqMon framework axrchitecture (Robinson, 2006)

Like *Zanshin*, ReqMon characterizes requirements as goals, also understanding them as complex entities, that exists both at design-time and runtime of a software system, organizing them in requirements models that aim to provide requirements traceability during the software life-cycle. However, *Zanshin* and ReqMon are different in the sense that the first one is focused on adaptations/evolution of a software system through changes in its requirements, while the last one is focused on monitoring the system and verifying if the requirements are being fulfilled. These similarities and differences are the main reasons why they were chosen to be instantiated to validate RRO.

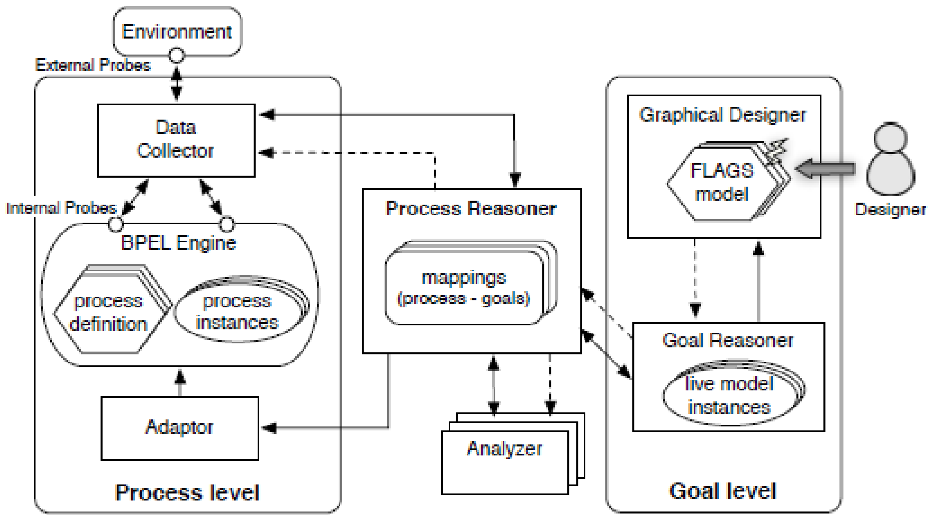

Fig. 5. FLAGS Runtime infrastructure (Pasquale et al., 2011)

2.4. FLAGS Framework

The Fuzzy Live Adaptive Goals for Self-Adaptive Systems (FLAGS) (Baresi et al., 2010; Pasquale et al., 2011) is a GORE-based framework for designing self-adaptive systems which extends the KAOS model (Van Lamsweerde et al., 1991), adding the concept of *adaptive goals*. Adaptive goals are live runtime entities, consisting of requirements that are responsible for defining the countermeasures that must be performed in order to keep system goals satisfied. Countermeasures may act by updating an existing requirement (relaxing or tightening), replacing it for a new one or even preventing it from being violated. In other words, countermeasures are responsible to change the live goal model that is kept by the system.

Figure 5 presents the FLAGS runtime infrastructure that allows requirements being used as runtime entities that provide adaptation capabilities to the system. In order to support requirements evolution, FLAGS works at the process and goal levels, managing two models at runtime: the FLAGS model and the implementation model. The first has requirements and adaptation goals, while the second includes the definition of the process and the monitoring and adaptation capabilities that are necessary at the process level.

The Process Reasoner is the core component of the architecture, being responsible for controlling the adaptation at the process level. To accomplish that, it monitors the runtime data that is collected to check if the conditions for an adaptation are satisfied. It is also responsible to send to the Goal Reasoner, runtime data that is used to change elements in the goal model. At the goal level, if a designer creates a new version of the FLAGS model using the Graphical Designer, live instances of this model are sent to the Process Reasoner, which propagates the changes to the running and next process instance of the system.

To illustrate the method, Figure 6 presents the FLAGS model for a web application called Click&Eat. White clouds are the main goals for Click&Eat, gray clouds represent the adaptation goals that define the adaptation capabilities of the system. The small circles represent the operationalizations of the adaptive goals, which define the exact actions that must be performed when an adaptation condition is triggered. In comparison to *Zanshin* and ReqMon, FLAGS also treats requirements as goals, and live goals models are also used in runtime as first-class citizens. However, FLAGS is closer to *Zanshin*, since it also deals with runtime requirements-based adaptation, even creating a runtime entity that defines how the requirements needs to change.

Lastly, it is important to mention that adaptation goals proposed by FLAGS, monitoring requirements used by ReqMon and *AwReqs* and *EvoReqs* proposed by *Zanshin* are examples of requirements being used at runtime, whose characteristics and properties are discussed in Section 6.

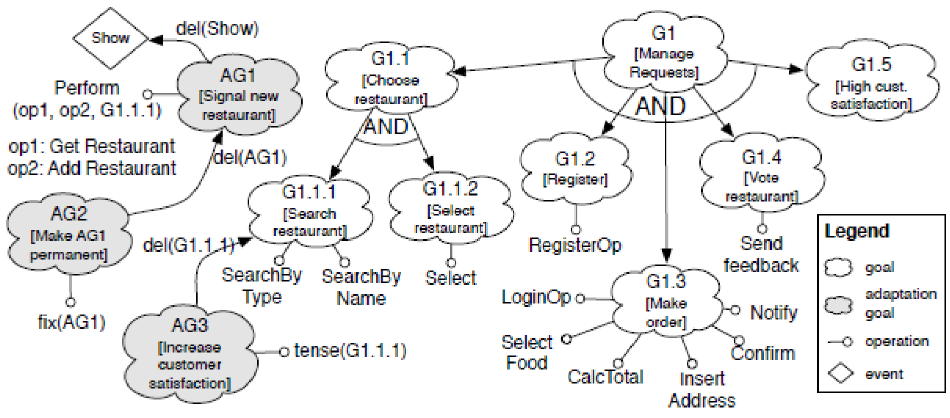

Fig. 6. FLAGS model for a Click&Eat web application (Pasquale et al., 2011)

3. Methodology

To develop RSRO and RRO, we used SABiO, a Systematic Approach for Building Ontologies (Falbo, 2014). SABiO supports the development of domain reference ontologies, as well as the design and coding of operational ontologies. We chose SABiO because it has been successfully used to develop domain ontologies, in particular software engineering reference domain ontologies, including the ones already integrated in SEON, such as: (Barcellos et al., 2010), (Bringuente et al., 2011), and (de Souza et al., 2017).

SABiO's development process comprises five main phases, namely (Falbo, 2014): (1) purpose identification and requirements elicitation; (2) ontology capture and formalization; (3) design; (4) implementation; and (5) test. These phases are supported by support processes, such as knowledge acquisition, reuse, documentation and evaluation. SABiO aims at developing both reference ontologies (phases 1 and 2) and operational ontologies (phases 3, 4 and 5). In this work, we are interested in building only domain reference ontologies, thus we performed only the first two phases. Although we did not implement the ontologies, we designed test cases for them.

The purpose of RSRO and RRO is to improve human knowledge and problem-solving in the Requirements domain, in general, and in the use of Requirements at Runtime (RRT), in particular. As non-functional requirements for RSRO/RRO, we defined that it would: be grounded on a well-known foundational ontology (**NFR1**); be based on consensual work from the literature (**NFR2**); and, to be integrated to SEON, reusing existing networked ontologies, or other relevant ontologies already published in the literature (**NFR3**).

Competency Questions (CQs) were elicited as functional requirements for RSRO/RRO. CQs are questions that the ontology should be able to answer (Grüninger and Fox, 1995) and they help to determine the scope of the ontology (Falbo, 2014). In total, thirteen CQs were identified, four for RSRO (cf. Section 5) and nine for RRO (cf. Section 6). They were refined using a bottom-up strategy, starting with simpler questions and proceeding to find more complex ones.

To identify proposals that use RRT and, thus, to satisfy NFR2, we performed a systematic mapping of the literature as a knowledge acquisition activity (as opposed to relying solely on the opinion of a few domain experts from our research group). Kitchenham and Charters (2007) define a Systematic Mapping as an extensive study on a specific topic that intends to identify evidence available on this theme. Following their method, we applied the steps illustrated in Figure 7.

First, we defined a *search string* that intended to cover all the relevant aspects of our research, and applied it in several search engines, in an attempt to find most studies existing in the literature about the research domain. In total, 1581 studies were returned by the search string. The search string was validated by checking if the *control articles* (6) that were chosen beforehand were retrieved from the databases. Next, papers returned from all search engines were combined and duplicate entries were removed. As result of this step, 912 studies remained. Then, we applied two filters, considering inclusion and exclusion

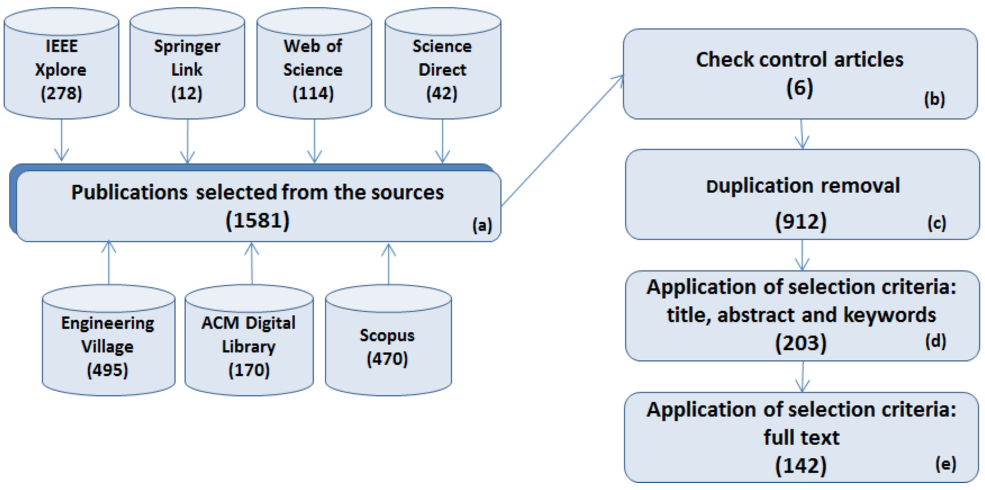
IEEE Xplore (278)
Springer Link (12)
Web of Science (114)
Science Direct (42)
Publications selected from the sources (1581) (a)
Engineering Village (495)
ACM Digital Library (170)
Scopus (470)
Check control articles (6) (b)
Duplication removal (912) (c)
Application of selection criteria: title, abstract and keywords (203) (d)
Application of selection criteria: full text (142) (e)

Fig. 7. Steps of the Systematic Mapping process on RRT.

criteria established at the beginning of the process. In this first filter, only the abstracts were read to evaluate if a paper should be selected or excluded from the mapping. Then, the selected publications (203) were once again analyzed against the selection criteria, but now considering the full text of the publication. To help reduce bias, publications were analyzed in the second filter by different specialist from the first. As result, 142 publications were found to satisfy the selection criteria, which accounts for a 91,02% reduction from the starting 1581 results.

During the mapping, we classified publications by purpose of use of RRT, separating them in two major categories: *Monitor Requirements* and *Change Requirements*. The former covers publications that propose checking if requirements defined at design time are being achieved at runtime, whereas the latter covers papers in which requirements are used not only as guidelines to monitoring but also as rules on how the system should adapt to keep satisfying its requirements. Although the results of the mapping are briefly summarized here, the mapping is not in the scope of this paper.

Returning to the ontology development process, we conducted weekly meetings in which the primary aspects and the scope of the ontologies were defined. These meetings were attended by several specialists in the areas of Requirements Engineering and Ontology Engineering (expanding the initial group of four engineers), who played different roles depending on the context: in requirement elicitation meetings, they acted as *Domain Experts* on Requirements and RRT, whereas in verification meetings, in which the many versions of the ontologies were analyzed and polished, they also played the role of *Ontology Engineers*.

Given the requirements for RSRO and RRO and the results of the Systematic Mapping, we started capturing the concepts and relations of RSRO/RRO. This capture process was highly iterative, using the aforementioned weekly meetings with specialists to refine the models. The resulting models are presented in sections 5 and 6.

For building RSRO and RRO, we reused the Ontology of Software Artifacts by Wang et al. (2014) and the interpretation of Non-Functional Requirements by Guizzardi et al. (2014) (addressing NFR3). Moreover, we grounded both ontologies in the Unified Foundational Ontology (UFO) (Guizzardi, 2005; Guizzardi et al., 2008) (aiming at satisfying NFR1), and latter we integrated them into the Software Engineering Ontology Network (SEON) (Ruy et al., 2016). In Section 4, we present the fragment of UFO used to provide the ontological foundations for RSRO and RRO and the reused ontologies.

As prescribed by SABiO, once the reference ontologies are developed, they need to be evaluated. SABiO's Evaluation Process comprises two main perspectives: (i) *Ontology Verification*: aims to ensure that the ontology is being built correctly, in the sense that the output artifacts of an activity meet the specifications imposed on them in previous activities. (ii) *Ontology Validation*: aims to ensure that the right ontology is being built, that is, the ontology fulfills its specific intended purpose.

To evaluate RSRO and RRO, we applied two evaluation approaches, namely: assessment by humans, and data-driven evaluation (Brank et al., 2005). For evaluating the reference ontology as a whole (intended

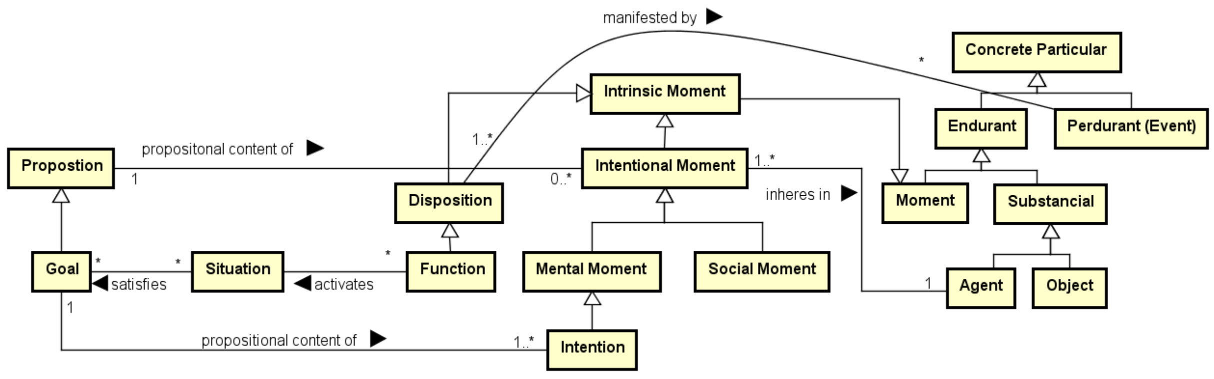

Fig. 8. Fragment of UFO showing Goals, Agents and Intentions.

purposes, competency questions and conceptual models), expert judgment was performed. In particular, for addressing ontology verification, we built tables indicating which ontology elements (concepts, relations, and axioms) are able to answer each competency question, as suggested by SABiO. For evaluating if the reference ontologies are able to represent real world situations, concepts and relations of RSRO/RRO were instantiated using data extracted from examples of methods described in Section 2, in a data-driven approach to ontology validation. Finally, we defined test cases for the ontologies, showing how the competency questions should be answered when using the instantiations as test case inputs.

4. Ontological Foundations

As mentioned before, RSRO and RRO were grounded in the Unified Foundational Ontology (UFO). We chose UFO because it has been constructed with the primary goal of developing foundations for conceptual modeling. Consequently, UFO addresses many essential aspects for conceptual modeling, which have not received a sufficiently detailed attention in other foundational ontologies (Guizzardi, 2005). Examples are the notions of material relations and relational properties. For instance, this issue did not receive up to now a treatment in DOLCE (Masolo et al., 2003), which focuses solely on intrinsic properties (qualities). Moreover, UFO offers a complete set of categories to tackle the specificities of the targeted domain and it has been successfully employed in a number of semantic analyses, such as the one conducted here (see detailed discussion in (Guizzardi et al., 2015)).

UFO is composed of three main parts: UFO-A, an ontology of endurants (Guizzardi, 2005); UFO-B, an ontology of perdurants/events (Guizzardi et al., 2013); and UFO-C, an ontology of social entities (both endurants and perdurants) built on the top of UFO-A and UFO-B (Guizzardi et al., 2008). In Figure 8, we present only a fragment of UFO containing the categories that are germane for the purposes of this article. Moreover, we illustrate these categories and some contextually relevant relations using UML (Unified Modeling Language) class diagrams. These diagrams are used here primarily for visualization. The reader interested in an in-depth discussion and formal characterization of UFO is referred to (Guizzardi, 2005; Guizzardi et al., 2008, 2013; Benevides et al., 2010).

Endurants and Perdurants are Concrete Individuals, entities that exist in reality and possess an identity that is unique. Endurants are entities that do not have temporal parts, but persist in time while keeping their identity (e.g., a person). Perdurants, also called Events, are composed by temporal parts (e.g., a trip) (Guizzardi, 2005).

Substantials are existentially independent Endurants. They can be agentive (Agent), i.e., bear intentional properties (states) such as beliefs, desires, intentions; or non-agentive (Object). Agents can bear special kinds of moments called Intentional Moments. An Intention is a specific type of Intentional Moment that refers to a desired state of affairs for which an Agent commits to pursuing (e.g., the intention of a student to take an exam). A Goal is a Proposition (the propositional content of an Intention) that

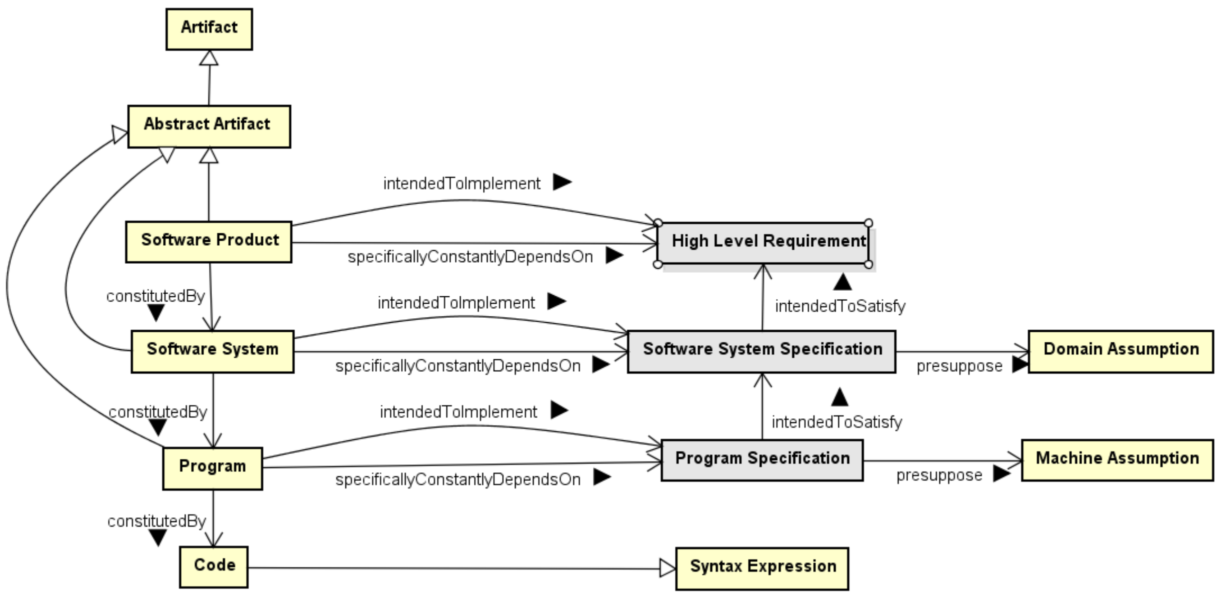

Fig. 9. Fragment of the Ontology of Software Artifacts (Wang et al., 2014).

can be satisfied by a Situation, i.e., a portion of the reality that can be comprehended as a whole, iff the Situation is the truthmaker of the Proposition expressed by the Goal (Guizzardi et al., 2008). Functions and Dispositions are Intrinsic Moments, i.e., existentially dependent entities that have potential to be realizable through the occurrence of an Event. This occurrence brings about a Situation.

Based on UFO, Guizzardi et al. (2014) present an interpretation of the difference between Functional and Non-Functional Requirements (FRs/NFRs), a frequently used categorization scheme in Requirements Engineering. According to the authors, requirements are Goals (as in UFO) and can be *functional* and/or *non-functional requirements*. Functional Requirements are those which refer to Functions, whereas NFRs refer to Qualities taking Quality Values in particular Quality Regions. Because the distinction between FRs/NFRs is usually accepted in the Requirements Engineering community and, also, because it is very relevant for the RRT domain, we decided to reuse this ontology in our work. The fact that the ontology is also grounded in UFO made the reuse simpler.

Finally, we also reused the conceptualization established in Wang et al.'s Ontology of Software Artifacts (Wang et al., 2014), which we here refer to with the acronym OSA, for reasons of brevity. According to them, software is a special kind of entity that is capable of changing while maintaining its numerical identity. These changes (i.e., a simple bug fixing or an entirely new release) are necessary for the natural evolution of software and are also one of the engines that move the software industry. When we start to think in specific software such as, e.g., Microsoft Excel, this fact becomes clearer: Excel has many releases and even more versions throughout its almost 30 years of existence, but has always maintained its identity as Microsoft's spreadsheet software.

To try to answer questions like "what does it mean for software to change?", "What is the difference between a new release and a new version?" and to address the ontological reasons for software being able to change while maintaining its nature, Wang et al. (2014) propose an ontology of software inspired by the Requirements Engineering literature, depicted in Figure 9, which shows a fragment of the original ontology. OSA makes the distinction between three artifacts: Software Product, Software System and Program. It also deals with the differences between a Program seen as a piece of Code and as a process, running in a medium.

A Program is a special type of Abstract Artifact, i.e., a non-physical entity created by humans (Irmak, 2013), with temporal properties and constituted by Code (a sequence of machine instructions). A Program is created with the purpose of performing a function of a given type, which is specified in a Program Specification. The Program is considered an Abstract Artifact because of its complex nature:

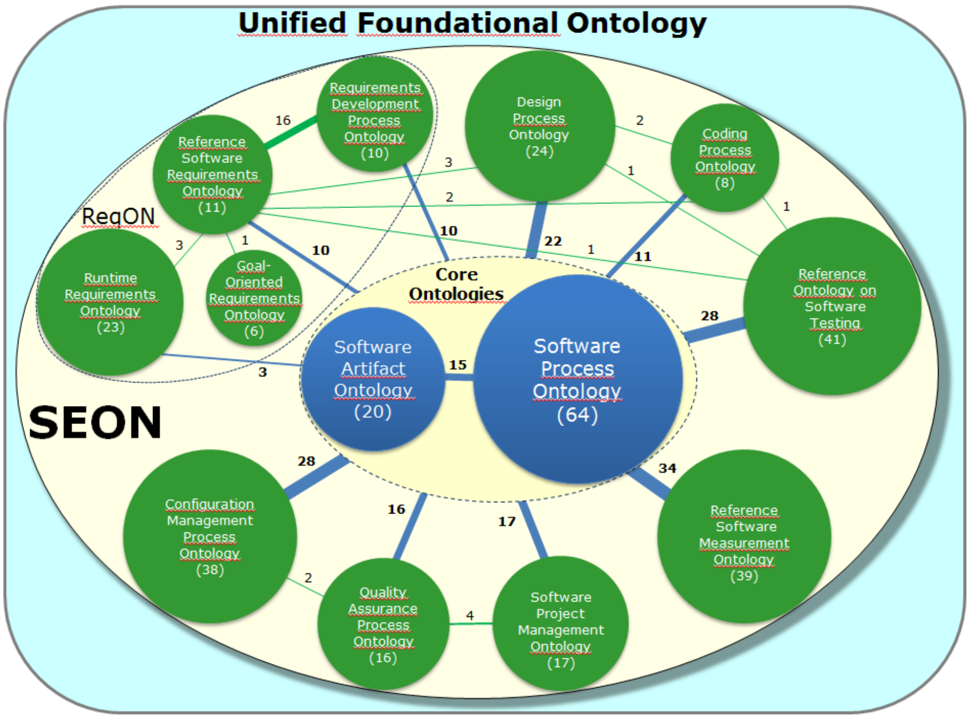
Unified Foundational Ontology
SEON
RegON
Core Ontologies
Requirements Development Process Ontology (10)
Design Process Ontology (24)
Coding Process Ontology (8)
Reference Software Requirements Ontology (11)
Runtime Requirements Ontology (23)
Goal-Oriented Requirements Ontology (6)
Software Artifact Ontology (20)
Software Process Ontology (64)
Reference Ontology on Software Testing (41)
Configuration Management Process Ontology (38)
Quality Assurance Process Ontology (16)
Software Project Management Ontology (17)
Reference Software Measurement Ontology (39)
16
3
2
10
22
1
11
28
15
34
17
4
2
1
3
1
3
10
2
28
16

Fig. 10. SEON: The Network view (Ruy et al., 2016).

it behaves as a universal since it has characteristics that are repeatable in its copies, but it also does not exists outside space and time, unlike a universal.

A Software System is constituted by a set of Programs and intends to determine the behavior of the machine towards the external environment. This behavior is specified by the Software System Specification. The Software Product is considered an Abstract Artifact that intends to implement the High Level Requirements, which represent the stakeholders intentions and goals.

It is important to mention that, although OSA is grounded in DOLCE, it is based in a part of DOLCE that is aligned with UFO. Thus, we could reuse it with no major problems. Moreover, we decided to reuse OSA and the interpretation of NFRs because the subjects they cover are within our scope of interest.

The ontologies developed in this work were integrated to the Software Engineering Ontology Network (SEON) (Ruy et al., 2016). An ontology network is a collection of ontologies related together through a variety of relationships, such as alignment, modularization and dependency. A networked ontology, in turn, is an ontology included in such a network, sharing concepts and relations with other ontologies (Suárez-Figueroa et al., 2012). SEON is designed seeking for: (i) taking advantage of well-founded ontologies (all its ontologies are ultimately grounded in UFO); (ii) providing ontology re-usability and productivity, supported by core ontologies organized as Ontology Pattern Languages; and (iii) solving ontology integration problems by providing integration mechanisms (Ruy et al., 2016).

In its current version, SEON includes core ontologies for software artifacts and processes, as well as domain ontologies for the main technical software engineering sub-domains, namely requirements, architectural design, coding and testing, and for some management sub-domains, namely software measurement, configuration management, and quality assurance. Figure 10 presents an overview of the SEON network. Each circle represents an ontology or an ontology subnetwork. The circles in the center are the core ontologies, the ones closer to the borders are domain ontologies already integrated in SEON. The circles' sizes vary according to the ontologies' sizes in terms of number of concepts (represented inside the circles in parenthesis). Lines represent links between networked ontologies, and line thickness represents the coupling level between them.

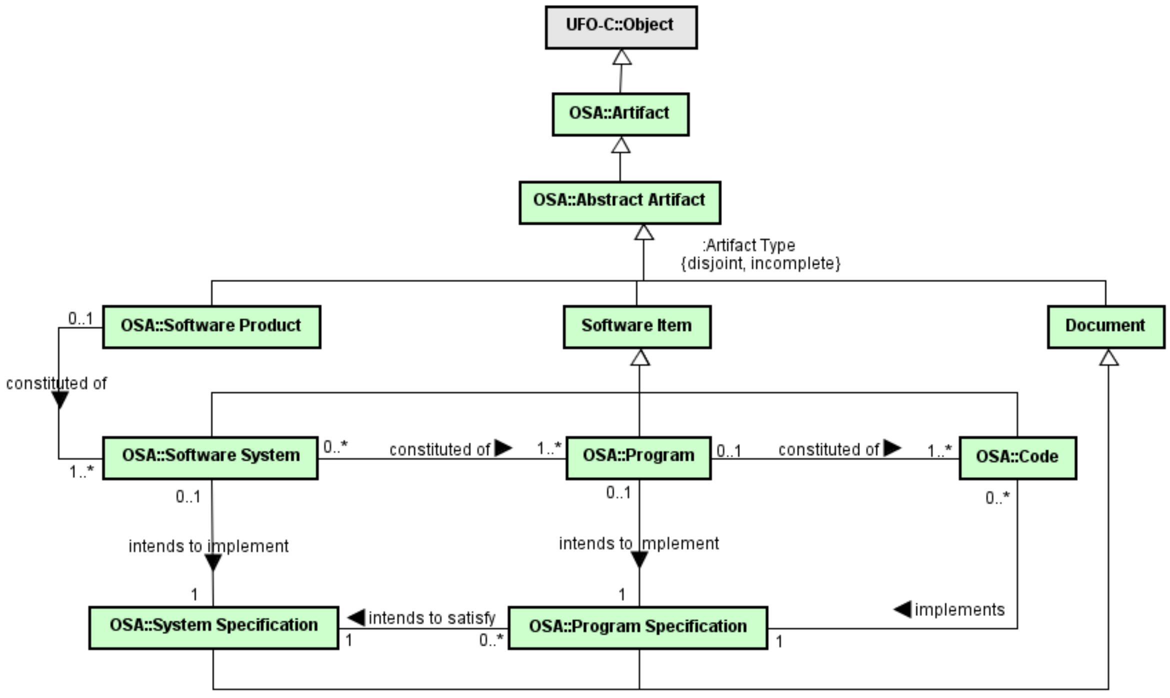

Fig. 11. Fragment of SEON's Software Artifacts Ontology.

Concerning requirements, ReqON is the SEON's ontology subnetwork devoted to requirements. The Reference Software Requirements Ontology (RSRO), presented in the next section, is the main SEON's ontology in the requirements domain. It captures the most general notions regarding requirements, which are valid for many Requirements Engineering approaches. The Goal-Oriented Requirements Ontology (GORO) (Negri et al., 2017) focuses on the basic notions of Goal-Oriented Requirements Engineering. The Requirements Development Process Ontology (RDPO) (Ruy et al., 2016) aims at representing the activities, artifacts and stakeholders involved in the Software Requirements Development Process. Finally, the Runtime Requirements Ontology (RRO), which is presented in Section 6, addresses the use of Requirements at Runtime.

RSRO extends SEON's Software Artifact Ontology (SAO), shown in Figure 11. As this figure shows, SAO is based on Wang et al.'s Ontology of Software Artifacts (Wang et al., 2014) (concepts from this ontology are prefixed by the acronym OSA). The main differences are: (i) in SEON, Code is also considered an Artifact (instead of a Syntactic Expression), in the sense that it is produced by an activity of the software process; and (ii) the introduction of two types of artifacts, namely Software Item and Document. In SEON, a Software Item is a piece of software, produced during the execution of a software process, not considered a complete Software Product, but an intermediary result. A Document, in turn, is any written or pictorial, uniquely identified, information related to the software development, usually presented in a predefined format. The specification of SEON is available at `https://nemo.inf.ufes.br/projects/seon/`.

5. Reference Software Requirements Ontology (RSRO)

The Reference Software Requirements Ontology (RSRO) aims to capture the most general notions regarding requirements, which are valid for Requirements Engineering in general. It is developed to be reused and extended to represent the many facets of Requirements Engineering, such as for modeling Goal-Oriented Requirements Engineering approaches (Negri et al., 2017) or the use of requirements at

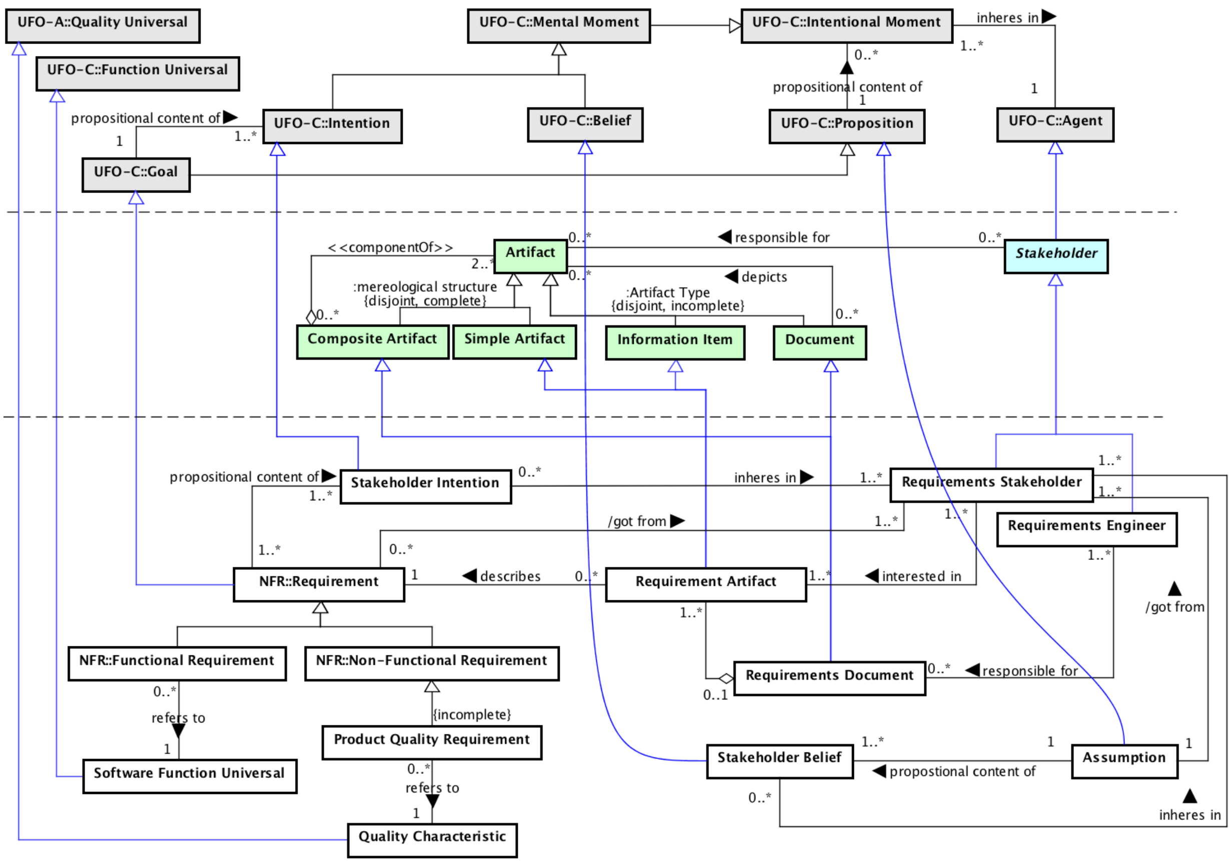

Fig. 12. Conceptual Model of RSRO.

runtime. To be as general as possible, the scope of RSRO is very narrow, and it aims to answer only the following competency questions:

- **CQ1:** What is a requirement?
- **CQ2:** What are the main types of requirements?
- **CQ3:** How are requirements documented?
- **CQ4:** Who are the main stakeholders involved with requirements?

Figure 12 shows RSRO's conceptual model. This ontology is strongly based on the ontological interpretation of Non-Functional Requirements made by Guizzardi et al. (2014). Concepts reused from (Guizzardi et al., 2014) are prefixed by NFR. Concepts from UFO, when necessary, are also present to show how RSRO is grounded in UFO.

As in (Guizzardi et al., 2014), a Requirement is a Goal in the sense of UFO, i.e., the propositional content of an Intention that inheres in an Agent, which in the Requirements Engineering domain is represented by a Stakeholder. Thus, Requirements Stakeholder is the role played by a Stakeholder, when requirements are got from her. Axiom A1 describes the derived relation[1] *got from*:

$$\begin{gathered}\forall r: Requirement, ri: RequirementIntention, rs: RequirementStakeholder\\ propositionalContentOf(r, ri) \wedge inheresIn(ri, rs) \rightarrow gotFrom(r, rs)\end{gathered} \tag{A1}$$

When a Requirement is documented in some kind of requirements document (e.g., as result of a requirements documentation activity in the Requirements Engineering process), there is a Requirement Artifact describing the Requirement. A Requirement Artifact is an Information Item that is responsible

[1]Derived relations are represented in the diagrams preceded by a "/".

for keeping relevant information for human use. Requirement Artifacts are put together in a document, called Requirements Document by a Requirements Engineer, who is said responsible for this document. Moreover, Axiom A2 holds: if a Requirement *r* is got from a Requirements Stakeholder *rs*, and *r* is described in the Requirements Artifact *ra*, then *rs* is interested in *ra*.

$$\begin{aligned}&\forall r\colon Requirement, rs\colon RequirementStakeholder, ra\colon RequirementArtifact,\\&\quad gotFrom(r, rs) \wedge describes(ra, r) \rightarrow interestedIn(rs, ra)\end{aligned} \tag{A2}$$

It is important to say that, although Axiom A2 holds, the relation *interested in* is not a derived relation, since there may be other stakeholders who are also interested in a requirement artifact, other than those from which the corresponding requirement has been got.

Requirements are categorized into Functional and Non-Functional Requirements (NFRs) (Guizzardi et al., 2014). Functional Requirements are those which refer to Software Function Universals. In other words, they are directly related to what a system needs to do, i.e., to the functionalities it needs to provide for its users. For instance, a very likely Functional Requirement in a meeting scheduler software system would be that the system needs to provide a way to schedule meetings. Regarding NFRs, there are several types of them. In particular, a Product Quality Requirement is a NFR that refers to a Quality Characteristic of the system, i.e., they are related to qualities of the software and specify how well the system executes its functionalities. For example, in the meeting scheduler, a possible NFR would be that the Meeting Scheduler needs to be 100% compliant with any HTML 5.1 Internet browser. This distinction between functional and non-functional requirements is also reflected in Requirement Artifacts, and thus, axioms A3 and A4 hold:

$$\begin{aligned}&\forall fr\colon FunctionalRequirement, ra\colon RequirementArtifact,\\&\quad describes(ra, fr) \rightarrow FunctionalRequirementArtifact(ra)\end{aligned} \tag{A3}$$

$$\begin{aligned}&\forall nfr\colon NonFunctionalRequirement, ra\colon RequirementArtifact,\\&\quad describes(ra, nfr) \rightarrow NonFunctionalRequirementArtifact(ra)\end{aligned} \tag{A4}$$

Finally, Assumptions describe states-of-affairs in the environment of interest that the Stakeholders believe to be true, i.e., they are the propositional content of Stakeholder Beliefs. They do not express the Intention of an Agent as Goals do, but a Belief that a specific situation exists in the environment. Representing such situations is sometimes necessary because they need to be considered in a given solution to a specific problem. We further elaborate on the subject of assumptions in a related ontology (Negri et al., 2017), out of the scope of this paper.

6. Runtime Requirements Ontology (RRO)

The Runtime Requirements Ontology (RRO) focuses on the notions related to the use of Requirements at Runtime. The purposes and intended uses of RRO include, among others, the following: to establish a common understanding of the main concepts used in several RRT frameworks, aiming at allowing them to interoperate; to provide a common conceptual model to be used for developing and reengineering RRT frameworks and systems. Taking these purposes and intended uses for RRO, the following competency questions were defined for it:

- **CQ5:** What is a running program?
- **CQ6:** Where does a running program execute?
- **CQ7:** What can be observed from a running program execution?
- **CQ8:** What is the relation between a running program and its requirements?

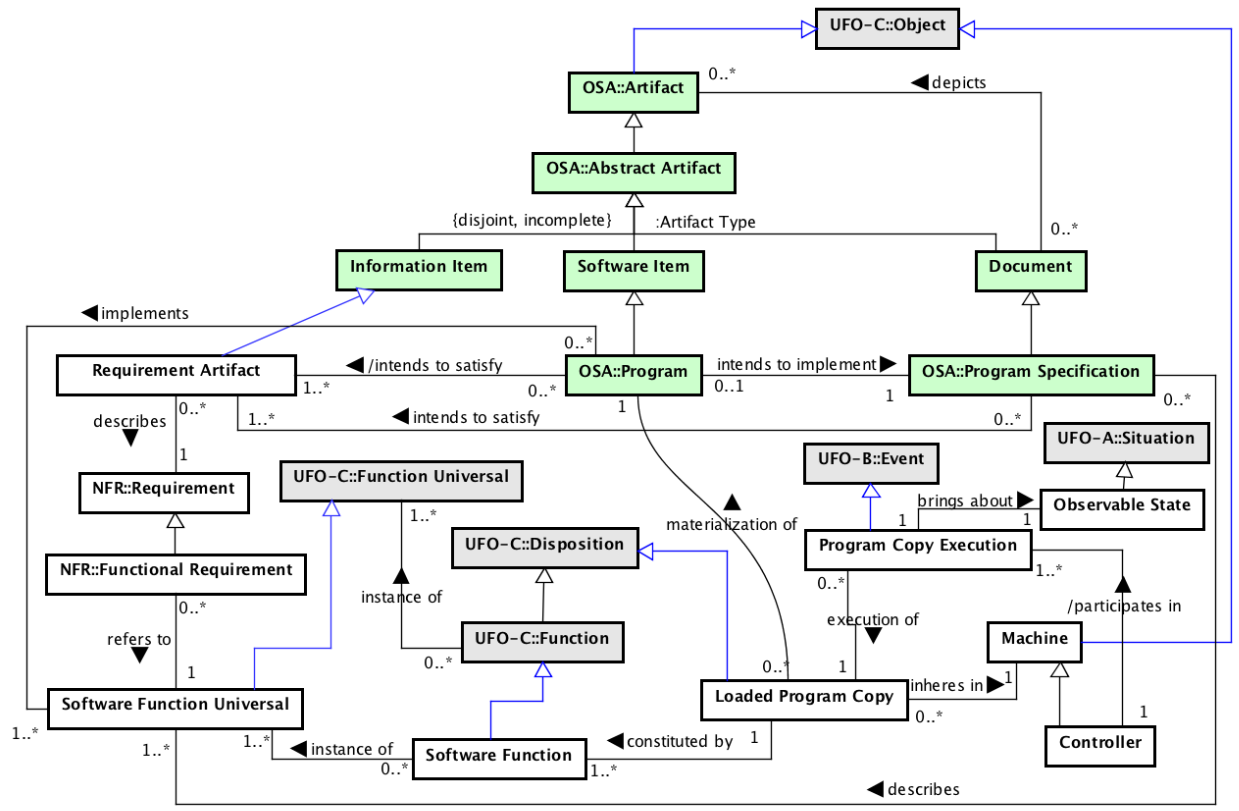

Fig. 13. Conceptual model fragment of RRO showing the ontological nature of RRO's concepts.

- **CQ9:** What is a requirement at runtime?
- **CQ10:** How are requirements used at runtime?
- **CQ11:** Which program is the target of a program that uses runtime requirements?
- **CQ12:** When is a program functionally in conformance with its specification?
- **CQ13:** When is a program that uses RRT considered internal or external to a system?

To manage the complexity of the model, it is divided in two parts, shown in figures 13 and 14. Concepts from the ontologies described in Section 4 are prefixed by the acronym of their original ontology, using NFR for Guizzardi et al.'s ontological interpretation of Non-Functional Requirements (Guizzardi et al., 2014) and OSA for Wang. et al.'s Ontology of Software Artifacts (Wang et al., 2014).

The first fragment, presented in Figure 13, is more general and focuses on capturing the ontological nature of program execution, which is the basis for the use of requirements at runtime. A Program, as defined by Wang et al. (2014), is an Abstract Artifact constituted by code written for a specific machine environment (e.g., Microsoft Excel for MacOS). To run a program, one must have a *copy* of the program and execute it. Irmak (2013) defines such *copy* as physical dispositions of particular computer components (e.g., the hard drive) to do certain things. He then describes the *execution* of the program as a kind of *event*, the physical manifestation of the aforementioned Dispositions. This Copy, being a Disposition, is a kind of Endurant and, thus, can change qualitatively while maintaining its identity, unlike an event, which cannot change (Moltmann, 2007).

In RRO, a Program is defined as a Software Item, a piece of software, produced during the software process, not considered a complete Software Product, but which aims at producing a certain result through execution on a computer, in a particular way, given by the Program Specification. In other words, a Program intends to implement a Program Specification, that intends to satisfy a set of Requirement Artifacts. Thus, we can say that there is a relation between Program and Requirement Artifact that is derived from the existing relation between Program Specification and Requirement Artifact and thus, Axiom A5 holds: if a Program *p* intends to implement a Program Specification *ps*, and *ps* intends to satisfy a Requirement Artifact *ra*, then *p* intends to satisfy *ra*.

$$\forall p\colon Program, ps\colon ProgramSpecification, ra\colon RequirementArtifact \\ intendsToSatisfy(ps, ra) \land intendsToImplement(p, ps) \rightarrow intendsToSatisfy(p, ra) \tag{A5}$$

Since a Program Specification describes a set of Software Function Universals, we can introduce another important notion: the satisfaction of the specification. Axiom A6 captures this notion in RRO: a Program *p* is functionally in conformance to its Program Specification *ps* if and only if *p* implements exactly the same Software Function Universals that are described by *ps*. In other words, for a program to be functionally in conformance with its specification, it must implement exactly the same set of Software Function Universals that are described by the Program Specification.

$$\forall p\colon Program, ps\colon ProgramSpecification \quad functionallyConformant(p, ps) \\ \iff (\forall sfu\colon SoftwareFunctionUniversal \quad describes(ps, sfu) \\ \rightarrow implements(p, sfu)) \tag{A6}$$

In the RRT domain (cf. Section 2), requirements are used not only to monitor (which can be done by observing the *events* referred to by Irmak) but also to adapt (i.e., change) a program. In that case, neither *events* (which are immutable) nor the *copy* at the hard drive (which is not running) can help. We are, thus, interested in the Loaded Program Copy as the materialization of a Program, inhering in a Machine, e.g., a copy of Microsoft Excel loaded in primary memory by MacOS on my MacBook.

The Loaded Program Copy is a complex Disposition that is constituted by one or more Software Functions, which are, in turn, instances of Software Function Universals. When the software development process is accomplished correctly, the Software Functions that constitute the (loaded) program copy are instances of the exact Software Function Universals described by the Program Specification and successfully implemented by the Program. With that said, the relations *constituted by* and *implements* must respect Axiom A7: if a Loaded Program Copy *lpc* is the materialization of a Program *p* that implements a Software Function Universal *sfu*, then there must exist a Software Function *sf* that is instance of *sfu* and constitutes *lpc*.

$$\forall lpc\colon LoadedProgramCopy, p\colon Program \quad materializationOf(lpc, p) \\ \rightarrow (\forall sfu\colon SoftwareFunctionUniversal \quad implements(p, sfu) \\ \rightarrow \exists sf\colon SoftwareFunction \quad instanceOf(sf, sfu) \land constitutedBy(lpc, sf)) \tag{A7}$$

Finally, as in (Irmak, 2013), a Program Copy Execution is an Event representing the execution of the Loaded Program Copy running in a Machine. In other words, the Program Copy Execution is the event of the physical manifestation of the dispositions, represented as the complex disposition Loaded Program Copy, that inhere in the Machine. In this context, the Machine participates in the Program Copy Execution playing the role of Controller, as defined Axiom A8.

$$\forall pce\colon ProgramCopyExecution, lpc\colon LoadedProgramCopy, m\colon Machine \\ executionOf(pce, lpc) \land inheresIn(lpc, m) \rightarrow participatesIn(m, pce) \land Controller(m)) \tag{A8}$$

Moreover, a Program Copy Execution, as an event, brings about a particular situation (the post-state of the event in the sense of UFO-B, as discussed in (Guizzardi et al., 2013)). We term this situation here an Observable State. As discussed in (Guizzardi et al., 2013), a situation is a particular configuration of a part of reality that can be understood as a whole, akin to notion of state of affairs in the philosophical literature. Situations can be characterized by the presence of objects, their intrinsic and relational moments and by the values that the qualities of these objects assume in certain quality regions, and so on.

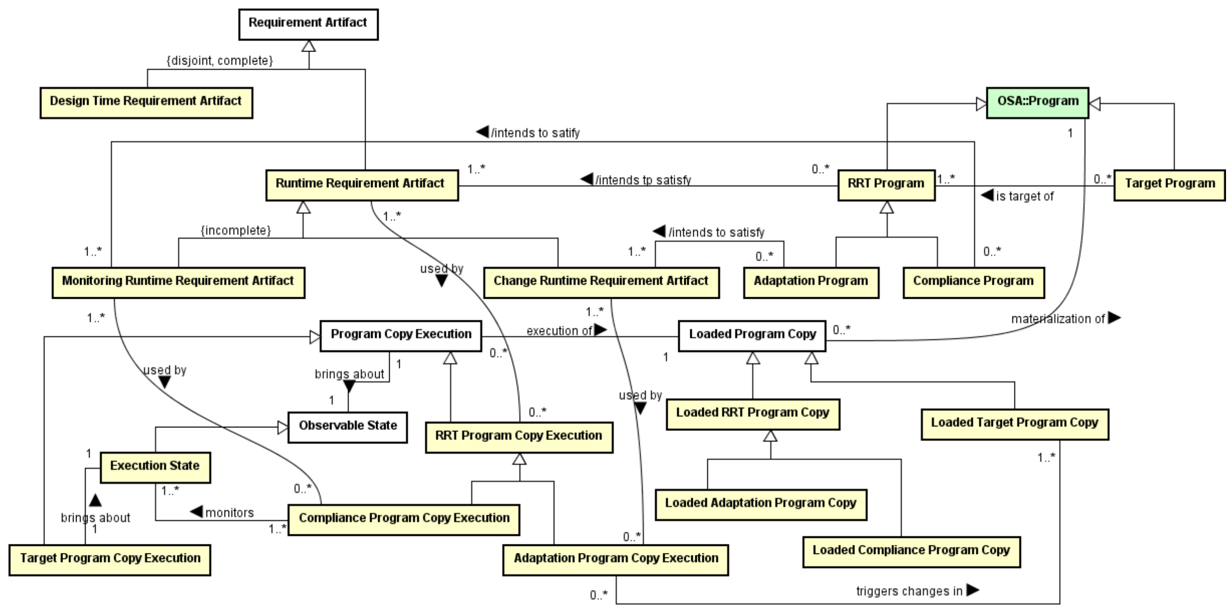

Fig. 14. Fragment of RRO that describes Runtime Requirement Artifacts, their nature and their relations with Programs and their executions.

We assume here that an Observable State is a situation resulting from the execution of a program copy, which involves qualities and quality values (qualia) of the Machine in which the Loaded Program Copy inheres, as well as of entities residing in that Machine (including the Loaded Program Copy itself).

The second fragment of RRO, shown in Figure 14, is more specific and focuses on runtime aspects. The first important distinction is between Runtime Requirement Artifacts, those that can be used by Program Copy Executions at runtime, from their non-runtime counterparts, named Design Time Requirement Artifacts. This distinction captures the fact that the description of requirements in artifacts can occur in several ways, like a text in natural language that is written in the software requirements specification, during design-time (represented in RRO as a Design Time Requirement Artifact), or in a computational file that could be processed by a program. The latter, if meant to be used at runtime, constitutes a Runtime Requirement Artifact.

Having both Design Time and Runtime Requirement Artifact accounts for an important distinction. Although most of the classic Requirements Engineering literature deals with requirements as entities that exist only in design time, this particular distinction exists, and it is widely accepted in the Runtime Requirements literature, as it can be seen in works of Dalpiaz et al. (2013), Souza et al. (2013a), Whittle et al. (2010), Borgida et al. (2013), Bencomo et al. (2010b), among others.

Runtime Requirement Artifact is further specialized into two specific subtypes: Monitoring Runtime Requirement Artifact defines the criteria for verifying if a requirement is being satisfied or not at runtime; on the other hand, Change Runtime Requirement Artifact specifies changes on the system's behavior, in order for the software system to keep fulfilling the goals it has been designed for. As examples of Monitoring Runtime Requirement Artifact we can mention the monitoring requirements used by ReqMon framework and the *AwReqs* proposed by Zanshin. As example of a Change Runtime Requirement Artifact, we can cite the *EvoReqs*, also from Zanshin (see Section 2). Also, although we did not find, during the systematic mapping, any proposal that collapses a Monitoring Runtime Requirement Artifact and a Change Runtime Requirement Artifact in a single entity, we believe that this is possible. In fact, there is no ontological constraint that forbids the combined runtime requirement artifact to exist. Moreover, the *incomplete* meta-property applied to this generalization set leaves open the possibility of existence of other types of requirements at runtime that were not addressed by the current version of RRO.

An RRT Program is a Program that is built intending to satisfy at least one Runtime Requirement Artifact. The aforementioned distinction between runtime requirement artifacts also reflects in RRT Programs (and in the corresponding Loaded Program Copies that materialize them and in the Program Copy Executions that are executions of the latter). Thus, RRT Program is specialized in Compliance Program , representing those RRT Programs that intend to satisfy at least one Monitoring Runtime Requirement Artifact, and Adaptation Program, representing those RRT Programs that intend to satisfy at least one Change Runtime Requirement Artifact. The generalization set of RRT Program is also not disjoint, since the same program can intend to satisfy, at the same time, both a Monitoring Runtime Requirement Artifact and a Change Runtime Requirement Artifact. In fact, the findings from the mapping study we performed show that many RRT Programs intend to satisfy requirement artifacts of these two types. Finally, some RRT Programs are built to act over (monitor/change) specific programs. When a Program is the target of an RRT Program, we say that it plays the role of a Target Program.

A Loaded RRT Program Copy is the materialization of an RRT Program, while an RRT Program Copy Execution is the execution of a Loaded RRT Program Copy. An RRT Program Copy Execution uses Runtime Requirement Artifacts as resources. A Compliance Program Copy Execution uses Monitoring Runtime Requirement Artifacts, and monitors Observable States produced by another Program Copy Execution, said Target Program Copy Execution. While using requirements at runtime, we are not interested in situations arising from the execution of any program, but only in those arising from the Target Program Copy Executions. For this reason, a Compliance Program Copy Execution monitors Execution States, i.e., the situations arising from the Target Program Copy Execution. The Compliance Program Copy Execution monitors the Execution States to verify if the runtime requirements comply with the criteria specified in the Monitoring Runtime Requirement Artifacts. If one or more requirements are not being fulfilled accordingly, an Adaptation Program Copy Execution can trigger changes in the the Loaded Program Copy being monitored, which is said a Loaded Target Program Copy, by using Change Runtime Requirement Artifacts.

Like the RRT Program generalization set, the generalization set of RRT Program Copy Execution (and also of Loaded RRT Program Copy) is not disjoint, since the same RRT Program Copy Execution can use, at the same time, both a Monitoring Runtime Requirement Artifact and a Change Runtime Requirement Artifact. Moreover, it is important to say that the very same software system (as it is defined in (Wang et al., 2014) — a set of programs, with distinct functionalities, working together and constituting a complex system) can be constituted by programs of any types (namely: Compliance, Adaptation and Target Programs) at the same time. In other words, we can have the case of a complex software that is responsible for a specific main task, but which is also built with aggregated modules that are responsible for monitoring the execution of this target functionality and trigger changes, whenever they are necessary. Based on that, it is important to notice that compliance and adaptation programs can be internal components of a *software system* (which can be constituted by many programs with many functionalities) or external applications that communicates with the Loaded Target Program Copy through specific channels. This distinction between external and internal RRT Programs is reflected in axioms A9 and A10:

$$\begin{aligned}&\forall rrtp\colon RRTProgram, tp\colon TargetProgram, s\colon SoftwareSystem\\ &\quad actsOver(rrtp, tp) \land constitutes(tp, s) \land constitutes(rrtp, s) \rightarrow internalTo(rrtp, s)\end{aligned} \tag{A9}$$

$$\begin{aligned}&\forall rrtp\colon RRTProgram, tp\colon TargetProgram, s\colon SoftwareSystem\\ &\quad actsOver(rrtp, tp) \land constitutes(tp, s) \land \neg constitutes(rrtp, s) \rightarrow externalTo(rrtp, s)\end{aligned} \tag{A10}$$

It is important to notice that these formulae do no exclude the case of an RRT Program that is internal to one system and external to another. Furthermore, compliance and adaptation program copies, when executing, are also subject to being monitored/changed by other executions of loaded compliance/adaptation programs, forming hierarchies of monitoring/adaptation systems. This happens because the type Target Program can be understood as a role played by a Program in the scope of the *is target of* relation.

Table 1

RSRO Verification against its Competency Questions.

CQs	Description, Concepts and *Relations*	Axioms
CQ1	**What is a requirement?** Requirement is a Goal which is the *propositional content of* a Stakeholder Intention that *inheres in* a Requirements Stakeholder, a role played by a Stakeholder.	A1
CQ2	**What are the main types of requirements?** Functional Requirement is a *subtype of* Requirement that *refers to* a Software Function Universal. Non-Functional Requirement is a *subtype of* Requirement that refers to a Quality Universal. Product Quality Requirement is a *subtype of* Non-Functional Requirement that *refers to* a Quality Characteristic.	–
CQ3	**How are requirements documented?** Requirements Document is composed of Requirements Artifacts that *describe* Requirements. Functional and Non-Functional Requirement are *subtypes of* Requirement.	A3, A4
CQ4	**Who are the main stakeholders involved with requirements?** Requirements Stakeholder is a *subtype of* (in fact a role played by) Stakeholder, who is *interested in* Requirements Artifacts. Requirements Engineer is a *subtype of* Stakeholder, who is *responsible for* a Requirements Document.	A2

7. Evaluation

To evaluate RSRO and RRO, we performed Ontology Verification & Validation (V&V) activities. Considering the guidelines proposed by SABiO (Falbo, 2014), RSRO and RRO were evaluated in three steps. First, in an **assessment by human approach to ontology evaluation** (Brank et al., 2005), we performed a verification activity by means of expert judgment, in which we checked whether the concepts, relations and axioms defined in the ontologies are able to answer their competency questions. Next, since a reference ontology should be able to represent real world situations, to validate RSRO/RRO, in a **data-driven approach to ontology evaluation** (Brank et al., 2005), we instantiated their concepts and relations using data extracted from projects using the different frameworks presented in Section 2. Finally, using the results of the two previous evaluation activities, we designed test cases to check if the ontologies answer the competency questions when instantiating them with data extracted from real case situations. Following each one of these evaluation steps are described in more details.

7.1. Verification by Experts

For verifying both RSRO and RRO, we started by manually checking if the concepts, relations and axioms defined in them are able to answer their competency questions (CQs). This approach enabled us to check not only if the CQs were answered, but also whether there were irrelevant elements in the ontology, i.e. elements that do not contribute to answer any of the questions. Table 1 illustrates this verification process for RSRO, showing which elements of the ontology (concepts, relations, properties and axioms) answer each one of the Competency Questions (CQs) of RSRO. Table 2 is analogous to Table 1, but showing the evaluation of RRO. These tables can also be used as a traceability tool, supporting ontology change management.

7.2. Validation

Concerning ontology validation, SABiO suggests that the ontology should be capable to properly represent real world situations. Based on that, we instantiated the ontologies using data extracted from three examples: the *Zanshin* Meeting Scheduler example (Souza, 2012), the E-Commerce Application used by (Robinson, 2006), and FLAGS Click&Eat Web portal example (Pasquale et al., 2011). These three examples were already presented in Section 2.

Table 3 illustrates the results of the instantiation with the Zanshin Meeting Scheduler example. Table 4 presents the instantiation for the e-commerce application, and Table 5 presents the instantiation using the

Table 2

RRO Verification against its Competency Questions.

CQs	Description, Concepts and *Relations*	Axioms
CQ5	**What is a running program?** Loaded Program Copy is a *materialization of* Program that *inheres in* a Machine. Program Copy Execution is an *execution of* Loaded Program Copy.	A7, A8
CQ6	**Where does a running program execute?** Loaded Program Copy *inheres in* Machine. Machine *participates in* Program Copy Execution, when it plays the role (*subtype of*) of a Controller.	–
CQ7	**What can be observed from a running program execution?** Program Copy Execution *brings about* an Observable State. Target Program Copy Execution *brings about* an Target Execution State.	–
CQ8	**What is the relation between a running program and its requirements?** Runtime Requirement Artifact is *used by* a RRT Program Copy Execution.	–
CQ9	**What is a requirement at runtime?** Runtime Requirement Artifact is a *subtype of* Requirement Artifact. Requirement Artifact is *subtype of* Information Item.	–
CQ10	**How are requirements used at runtime?** Monitoring Runtime Requirement Artifact is a *subtype of* Runtime Requirement Artifact and is *used by* a Compliance Program Copy Execution to *monitor* Target Execution State. Change Runtime Requirement Artifact is a *subtype of* Runtime Requirement Artifact and is *used by* an Adaptation Program Copy Execution to *change* the Loaded Target Program Copy.	–
CQ11	**Which program is the target of a program that uses runtime requirements?** Target Program is a *subtype of* Program. RRT Program *acts over* Target Program.	–
CQ12	**When is a program functionally in conformance with its specification?** A Program is *in conformance to* a Program Specification that it *intends to implement* when it *implements* all Software Function Universals that are described by that Program Specification.	A5, A6
CQ13	**When is a program that uses RRT considered internal or external to a system?** The RRT Program is considered *internal* if it is part of the same Software System constituted of the Target Program. Otherwise, it is considered *external*.	A9, A10

Flags framework. As mentioned in Section 2, these frameworks were chosen because they are well-defined approaches that we found during the systematic mapping of the literature, and because they propose distinct ways to represent and use requirements at runtime.

The successful instantiation of RRO with data coming from these three well-accepted Runtime Requirements frameworks gave us indications of the appropriateness of the proposed ontology as a reference model of this domain.

7.3. Test Case Designs

Considering the instances of concepts used to validate RSRO and RRO (as discussed above), test cases can be designed for the competency questions, in a competency question-driven approach for ontology testing (Falbo, 2014). In such approach, a test case comprises a competency question (the specification to be tested), plus the instantiation data from a fragment of the ontology being tested (input), and the expected result based on the considered instantiation (expected output). Table 6 shows examples of test cases for CQ11.

It is important to say that, if RSRO and RRO were implemented in some computational language (such as OWL), the test cases could be implemented (as SPARQL queries) and the resulting operational ontologies could be tested in a running environment (e.g., Protégé). However, test implementation and execution are out of the scope of this work. Furthermore, CQ 11 was chosen as an example to demonstrate test cases, whose should be implemented for all the CQs of an ontology.

Table 3
Results of RRO instantiation using the Meeting Scheduler example presented in (Souza, 2012).

Concept	**Instance**
Requirement	A stakeholder wants a software system that allows users to (among other things) characterize meetings before scheduling them.
Software Function Universal	The function of producing meeting characterizations from appropriate user input as required by the stakeholders.
Program	An implementation of the Meeting Scheduler system for a specific machine environment (e.g., a specific operating system).
Software Function	The disposition of the specific Meeting Scheduler implementation to produce characterizations when given proper inputs.
Loaded Program Copy	Materialization of a Meeting Scheduler implementation loaded in a machine's main memory.
Program Copy Execution	The event of the loaded copy of the Meeting Scheduler executing in the machine.
Observable State / Execution State	When given the characteristics of a meeting, the Meeting Scheduler produces the record of a new meeting (e.g., in a database).
Requirement Artifact	*Characterize Meeting* task, from the requirements (goal) model.
Runtime Requirement Artifact	*Characterize Meeting* task, represented in XML to be consumed by *Zanshin* components at runtime.
Compliance Program Copy Execution	The Monitor component of *Zanshin* running in some machine.
Monitoring Runtime Requirement Artifact	*Characterize meeting should never fail* Awareness Requirement (Souza et al., 2013a), represented in XML.
Adaptation Program Copy Execution	The Adapt component of *Zanshin* running in some machine.
Change Runtime Requirement Artifact	*Retry Characterize Meeting after 5 seconds* Evolution Requirement (Souza et al., 2013b), represented in XML.

Table 4
Results of RRO instantiation using the e-commerce example based on ReqMon and presented in (Robinson, 2006).

Concept	**Instance**
Requirement	A customer wants to be able to buy an specific item on-line
Program	An implementation of an e-commerce page for a specific machine (e.g: a computer with a Java EE compliant application server)
Software Function	The disposition of the specific implementation of a set of pages that represents a purchase being made.
Loaded Program Copy	Materialization of an e-commerce page implementation loaded in a machine's main memory.
Program Copy Execution	The event of the loaded copy of the e-commerce being executed in a machine (ex: an application server).
Observable State / Execution State	An instance of a complete execution of the e-commerce system done by a customer.
Requirement Artifact	The goal *CreateVendorPurchaseOrder* present in the requirements model of the e-commerce system.
Runtime Requirement Artifact	The goal *CreateVendorPurchaseOrder* represented by a XML file and ready to be used by ReqMon monitor component.
Compliance Program Copy Execution	*ReqMon Monitoring Tools* being executed in a machine.
Monitoring Runtime Requirement Artifact	The monitor *IMonPurchaseOrderDenialOfService* derived from the requirement with the same name.
Adaptation Program Copy Execution	This category is not presented by ReqMon, since it does not consider adaptation.
Change Runtime Requirement Artifact	This category is not presented by ReqMon, since it does not consider a requirement that is able to change other requirements

Table 5

Results of RRO instantiation using the Click&Eat example created with FLAGS and presented in (Pasquale et al., 2011).

Concept	**Instance**
Requirement	A customer wants to find a restaurant.
Program	An implementation of the Click&Eat application for a specific machine (e.g: an iPhone App).
Software Function	The disposition of the specific Click&Eat implementation to produce the desired outputs when given the proper input.
Loaded Program Copy	Materialization of a Click&Eat implementation loaded in a machine's (e.g: an iPhone) main memory.
Program Copy Execution	The event of the execution of a Click&Eat application in a machine.
Observable State / Execution State	An instance of a complete execution of Click&Eat done by a customer.
Requirement Artifact	The goal *Manage Request* present in the *FLAGS Model* of the Click&Eat application.
Runtime Requirement Artifact	Adaptation goal *Make New Restaurant found permanent* implemented in a specific language (e.g: jBPM).
Compliance Program Copy Execution	*Data Collector* and *Process Reasoner* components of a FLAGS implementation being executed in a machine, monitoring runtime data.
Monitoring Runtime Requirement Artifact	An internal probe of FLAGS implemented in a Business Process Execution Language (e.g: jBPM).
Adaptation Program Copy Execution	*Adaptor* component of a FLAGS implementation being executed in a machine.
Change Runtime Requirement Artifact	Operationalization *Get Restaurant* from adaptation goal *Signal New Restaurant* implemented in a Business Process Execution Language (e.g: jBPM).

Table 6

Example test cases for RRO competency question CQ11 - *Which program is the target of a program that uses runtime requirements?*

Test Case Id	**Example**	**Input**	**Expected Output**
T11.1	Meeting Scheduler (Table 3)	Adapt component of *Zanshin* (RRT Program)	An implementation of the Meeting Scheduler system for a specific machine environment (e.g., a specific operating system) (Target Program).
T11.2	e-Commerce (Table 4)	ReqMon Monitoring Tool (RRT Program)	An implementation of an e-commerce page for a specific machine (e.g: An Android Phone) (Target Program).
T11.3	Click&Eat (Table 5)	Adaptor component (RRT Program)	The Click&Eat application for a specific machine (e.g: an iPhone App) (Target Program).

8. Related Works

During the systematic mapping of the literature and the early stages of the RRO development process we have taken in consideration ontologies and ontological analysis of requirements. In this section we will present some of these ontologies that were relevant to our research.

By definition, a core ontology is a mid-term ontology, that is not as specific as a domain ontology but also not so domain-independent as a foundational ontology. Jureta et al. (2009) propose a new core ontology for requirements (CORE), based on the seminal work of Zave and Jackson (1997) and grounded in DOLCE (Masolo et al., 2003). The authors extend Zave and Jackson's formulation of the requirements problem, in order to "establish new criteria for determining whether RE has been successfully completed" (Jureta et al., 2008). CORE covers all types of basic concerns that stakeholders communicate to requirements engineers, thus establishing a new framework for the Requirements Engineering process. CORE was, by far, the ontology that was used the most as basis for works included in the results of the systematic mapping, including, e.g., *Zanshin* (Souza, 2012; Gillain et al., 2013; Qureshi et al., 2011). However, CORE does not cover concepts that are specific to the RRT domain.

Guizzardi et al. (2014) propose an ontological interpretation of non-functional requirements (NFRs) based on of UFO. As briefly mentioned in Section 4, NFRs and functional requirements (FRs) are seen as goals, with the major difference that the former refers to qualities and the latter to functions. In UFO, qualities and functions are both sub-categories of intrinsic moments, but qualities are properties that are manifested whenever they exist, whereas functions are dispositional properties that are manifested in certain circumstances and through the execution of an event. In their work, the authors advance the work of CORE and motivate the choice for adopting UFO as opposed to DOLCE in this domain. In our work, we have imported their distinction of FRs and NFRs in order to relate runtime requirement artifacts to their counterparts in design time through this widely accepted classification, emphasizing that both functional and non-functional design time requirements can give birth to runtime requirement artifacts.

Other ontologies discovered during the systematic mapping process include OWL-based ontologies used to support dynamic reconfiguration of service-oriented applications (Kim et al., 2007; Liu and Feng, 2012); a BDI-based ontology used to implement an inference mechanism of human intentions in context-aware systems (Oyama et al., 2008); an enterprise ontology used to monitor services (de Alencar Silva and Weigand, 2011); an ontology of communicative acts for monitoring service-based systems (Robinson and Purao, 2011); among others. None of the approaches found during the mapping, however, provided a well-founded domain reference ontology to aid in establishing precise descriptions for the concepts in the RRT domain. Our work aims at filling this gap in the literature.

9. Conclusions

The main contributions of this paper is the definition of RSRO and RRO. The former is a domain reference ontology about the use of requirements and the artifacts that describe them. The latter is a domain reference ontology about the use of different types of requirement artifacts at runtime. To develop them, we followed the SABiO approach for identifying the purpose, eliciting requirements, capturing, formalizing, verifying and validating the ontologies. Both ontologies are integrated into the Software Engineering Ontology Network (SEON) and had domain specific categories and constraints formally characterized.

As future work, we intend to create an ontology that combines concepts of RRO with concepts from Goal-Oriented Requirements Engineering (GORE), a paradigm that is very popular in Requirements Engineering research (fact that is confirmed by the systematic mapping results). We plan to use this new ontology to derive a new set of properly grounded meta-models in order to develop a new version of the *Zanshin* framework for adaptive systems. In this current version of RRO, we have decided not to base our ontology in GORE or any other specific RE approach. By doing this, we maintain the generality of our approach, not excluding any potential users of our contribution.

We will also continue SABiO's development process for RSRO and RRO, creating operational ontologies (e.g. in OWL) that could be used for the proposals that make use of requirements at runtime. This will also allow us to further evaluate the ontology, by using queries (e.g., in SPARQL) to actually run the test cases for the ontologies.

Acknowledgements

NEMO (`http://nemo.inf.ufes.br`) is currently supported by Brazilian research agencies FAPES (# 0969/2015) and CNPq (# 485368/2013-7, # 461777/2014-2). We would like to thank Nicola Guarino, Beatriz Martins and Pedro Negri for their participation during discussions about the research contained herein. Special thanks to Fabiano Ruy for the help with the integration of the ontologies into SEON.

References

Barcellos, M.P., de Almeida Falbo, R. & Dal Moro, R. (2010). A Well-Founded Software Measurement Ontology. In *Proc. of the 6th International Conference on Formal Ontology in Information Systems (FOIS 2010)* (pp. 213–226).

Baresi, L., Pasquale, L. & Spoletini, P. (2010). Fuzzy goals for requirements-driven adaptation. In *Proc. of the 18th IEEE International Requirements Engineering Conference (RE 2010)* (pp. 125–134). IEEE.

Bencomo, N., Whittle, J., Sawyer, P., Finkelstein, A. & Letier, E. (2010a). Requirements Reflection: Requirements as Runtime Entities. In *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10)* (Vol. *2*, pp. 199–202). Cape Town, South Africa: ACM.

Bencomo, N., Whittle, J., Sawyer, P., Finkelstein, A. & Letier, E. (2010b). Requirements reflection: requirements as runtime entities. In *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2* (pp. 199–202). ACM.

Bencomo, N., Letier, E., Finkelstein, A., Whittle, J. & Welsh, K. (Eds.) (2011). *Proceedings of the 2nd International Workshop on Requirements@Run.Time*. IEEE.

Benevides, A.B., Guizzardi, G., Braga, B.F.B. & Almeida, J.P.A. (2010). Validating Modal Aspects of OntoUML Conceptual Models Using Automatically Generated Visual World Structures. *Journal of Universal Computer Science*, 16(20), 2904–2933.

Borgida, A., Dalpiaz, F., Horkoff, J. & Mylopoulos, J. (2013). Requirements models for design-and runtime: A position paper. In *Proc. of the 5th International Workshop on Modeling in Software Engineering* (pp. 62–68). IEEE Press.

Bourque, P., Fairley, R.E., et al. (2014). *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press.

Brank, J., Grobelnik, M. & Mladenic, D. (2005). A survey of ontology evaluation techniques. In *Proc. of the 2005 Conference on Data Mining and Data Warehouses* (pp. 166–170). Ljubljana, Slovenia.

Bringuente, A.C.O., Falbo, R.A. & Guizzardi, G. (2011). Using a Foundational Ontology for Reengineering a Software Process Ontology. *Journal of Information and Data Management*, 2(3), 511–526.

Cheng, B.H.C. & Atlee, J.M. (2007). Research Directions in Requirements Engineering. In *Future of Software Engineering (FOSE '07)* (pp. 285–303). IEEE.

Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P. & Magee, J. (Eds.) (2009). *Software Engineering for Self-Adaptive Systems. Lecture Notes in Computer Science* (Vol. *5525*). Springer.

Dalpiaz, F., Borgida, A., Horkoff, J. & Mylopoulos, J. (2013). Runtime goal models. In *Proc. of the IEEE 7th International Conference on Research Challenges in Information Science* (pp. 1–11). Paris, France: IEEE.

de Alencar Silva, P. & Weigand, H. (2011). Enterprise monitoring ontology. *Conceptual Modeling–ER 2011*, 132–146.

de Lemos, R., Giese, H., Müller, H.A. & Shaw, M. (Eds.) (2013). *Software Engineering for Self-Adaptive Systems II. Lecture Notes in Computer Science* (Vol. *7475*). Springer.

de Souza, É.F., Falbo, R.d.A. & Vijaykumar, N.L. (2017). ROoST: Reference Ontology on Software Testing. *Applied Ontology*, 1–32.

Duarte, B.B., Souza, V.E.S., Leal, A.L.d.C., Guizzardi, G., Falbo, R.d.A. & Guizzardi, R.S.S. (2016). Towards an ontology of requirements at runtime. In *Proc. of the 9th International Conference on Formal Ontology in Information Systems (FOIS 2016)* (Vol. *283*, p.255). IOS Press.

Falbo, R.A. (2014). SABiO: Systematic Approach for Building Ontologies. In G. Guizzardi, O. Pastor, Y. Wand, S. de Cesare, F. Gailly, M. Lycett and C. Partridge (Eds.), *Proc. of the Proceedings of the 1st Joint Workshop ONTO.COM / ODISE on Ontologies in Conceptual Modeling and Information Systems Engineering*. Rio de Janeiro, RJ, Brasil: CEUR.

Feather, M.S., Fickas, S., van Lamsweerde, A. & Ponsard, C. (1998). Reconciling system requirements and runtime behavior. In *Proc. of the 9th International Workshop on Software Specification and Design* (pp. 50–59). IEEE.

Gillain, J., Faulkner, S., Jureta, I.J. & Snoeck, M. (2013). Using goals and customizable services to improve adaptability of process-based service compositions. In *Proc. of the 7th IEEE International Conference on Research Challenges in Information Science (RCIS)* (pp. 1–9).

Grüninger, M. & Fox, M. (1995). Methodology for the Design and Evaluation of Ontologies. In *IJCAI'95 Workshop on Basic Ontological Issues in Knowledge Sharing*.

Guizzardi, G. (2005). Ontological Foundations for Structural Conceptual Models. PhD Thesis, University of Twente, The Netherlands.

Guizzardi, G. (2007). On ontology, ontologies, conceptualizations, modeling languages, and (meta) models. *Frontiers in artificial intelligence and applications*, 155, 18.

Guizzardi, G., de Almeida Falbo, R. & Guizzardi, R.S. (2008). Grounding Software Domain Ontologies in the Unified Foundational Ontology (UFO): The case of the ODE Software Process Ontology. In *Proc. of the 11th Iberoamerican Conference on Software Engineering (CIbSE)* (pp. 127–140).

Guizzardi, G., Wagner, G., Almeida Falbo, R., Guizzardi, R.S.S. & Almeida, J.P.A. (2013). Towards Ontological Foundations for the Conceptual Modeling of Events. In *Proc. of the 32th International Conference on Conceptual Modeling* (pp. 327–341). Springer.

Guizzardi, G., Wagner, G., Almeida, J.P.A. & Guizzardi, R.S.S. (2015). Towards Ontological Foundation for Conceptual Modeling: The Unified Foundational Ontology (UFO) Story. *Applied Ontology*, 10(3–4), 259–271.

Guizzardi, R.S.S., Li, F.-L., Borgida, A., Guizzardi, G., Horkoff, J. & Mylopoulos, J. (2014). An Ontological Interpretation of Non-Functional Requirements. In P. Garbacz and O. Kutz (Eds.), *Proc. of the 8th International Conference on Formal Ontology in Information Systems* (Vol. *267*, pp. 344–357). Rio de Janeiro, RJ, Brasil: IOS Press.

Huebscher, M.C. & McCann, J.A. (2008). A survey of Autonomic Computing—Degrees, Models, and Applications. *ACM Computing Surveys*, 40(3), 1–28.

Irmak, N. (2013). Software is an abstract artifact. *Grazer Philosophische Studien*, 86(1), 55–72.

Jureta, I., Mylopoulos, J. & Faulkner, S. (2008). Revisiting the Core Ontology and Problem in Requirements Engineering. In *Proc. of the 16th IEEE International Requirements Engineering Conference* (pp. 71–80). IEEE.

Jureta, I.J., Mylopoulos, J. & Faulkner, S. (2009). A core ontology for requirements. *Applied Ontology*, 4(3-4), 169–244.

Kim, J., Lee, J. & Lee, B. (2007). Runtime Service Discovery and Reconfiguration Using OWL-S Based Semantic Web Service. In *7th IEEE International Conference on Computer and Information Technology (CIT 2007)* (pp. 891–896).

Kitchenham, B.A. & Charters, S. (2007). Guidelines for performing Systematic Literature Reviews in Software Engineering. Technical report, Keele University, UK. `https://www.cs.auckland.ac.nz/{\texttildelow}mria007/Sulayman/Systematic{\_}reviews{\_}5{\_}8.pdf`.

Kitchenham, B.A., Budgen, D. & Brereton, O.P. (2010). The Value of Mapping Studies: A Participantobserver Case Study. In *Proc. of the 14th International Conference on Evaluation and Assessment in Software Engineering. EASE'10* (pp. 25–33). Swinton, UK, UK: British Computer Society.

Kotonya, G. & Sommerville, I. (1998). *Requirements engineering: processes and techniques*. Wiley Publishing.

Liu, W. & Feng, Z. (2012). Uncertainty Modeling of Self-adaptive Software Requirement. *International Journal of Advancements in Computing Technology*, *4*(11).

Masolo, C., Borgo, S., Gangemi, A., Guarino, N., Oltramari, A. & Schneider, L. (2003). Dolce: a descriptive ontology for linguistic and cognitive engineering. *WonderWeb Project, Deliverable D17 v2, 1*.

Moltmann, F. (2007). Events, tropes, and truthmaking. *Philosophical Studies*, 134(3), 363–403.

Negri, P.P., Souza, V.E.S., de Castro Leal, A.L., de Almeida Falbo, R. & Guizzardi, G. (2017). Towards an Ontology of Goal-Oriented Requirements. In *Proc. of the 20th Workshop on Requirements Engineering (WER) at the 20th Ibero-American Conference on Software Engineering (CIbSE)*.

Oyama, K., Jaygarl, H., Xia, J., Chang, C.K., Takeuchi, A. & Fujimoto, H. (2008). A Human-Machine Dimensional Inference Ontology that Weaves Human Intentions and Requirements of Context Awareness Systems. In *Proc. of the 32nd Annual IEEE International Computer Software and Applications Conference* (pp. 287–294).

Pasquale, L., Baresi, L. & Nuseibeh, B. (2011). Towards adaptive systems through requirements@runtime. In *Proc. of the 6th Workshop on Models@run.time*.

Qureshi, N.A. & Perini, A. (2010). Requirements Engineering for Adaptive Service Based Applications. In *Proc. of the 18th IEEE International Requirements Engineering Conference* (pp. 108–111). Sydney, Australia: IEEE.

Qureshi, N.A., Jureta, I.J. & Perini, A. (2011). Requirements engineering for self-adaptive systems: Core ontology and problem statement. In *International Conference on Advanced Information Systems Engineering* (pp. 33–47). Springer.

Robertson, S. & Robertson, J. (2012). *Mastering the requirements process: Getting requirements right*. Addison-wesley.

Robinson, W.N. & Purao, S. (2011). Monitoring Service Systems from a Language-Action Perspective. *IEEE Transactions on Services Computing*, 4(1), 17–30.

Robinson, W.N. (2006). A requirements monitoring framework for enterprise systems. *Requirements Engineering*, 11(1), 17–41.

Ruy, F.B., Falbo, R.d.A., Barcellos, M.P., Costa, S.D. & Guizzardi, G. (2016). SEON: A Software Engineering Ontology Network. In *Knowledge Engineering and Knowledge Management: 20th International Conference, EKAW 2016, Bologna, Italy, November 19-23, 2016, Proceedings 20* (pp. 527–542). Springer.

SEI/CMU (2010). CMMI® for Development, Version 1.3, Improving processes for developing better products and services. *no. CMU/SEI-2010-TR-033. Software Engineering Institute*.

Souza, V.E.S. (2012). Requirements-based Software System Adaptation. PhD Thesis, University of Trento, Italy.

Souza, V.E.S., Lapouchnian, A., Robinson, W.N. & Mylopoulos, J. (2013a). Awareness Requirements. In R. Lemos, H. Giese, H.A. Müller and M. Shaw (Eds.), *Software Engineering for Self-Adaptive Systems II. Lecture Notes in Computer Science* (Vol. *7475*, pp. 133–161). Springer.

Souza, V.E.S., Lapouchnian, A., Angelopoulos, K. & Mylopoulos, J. (2013b). Requirements-driven software evolution. *Computer Science - Research and Development*, 28(4), 311–329.

Suárez-Figueroa, M.C., Gómez-Pérez, A., Motta, E. & Gangemi, A. (2012). *Ontology engineering in a networked world*. Springer Science & Business Media.

Van Lamsweerde, A., Dardenne, A., Delcourt, B., Dubisy, F., et al. (1991). The KAOS project: Knowledge acquisition in automated specification of software. In *Proc. of the AAAI Spring Symposium Series*.

Wang, X., Guarino, N., Guizzardi, G. & Mylopoulos, J. (2014). Towards an Ontology of Software: a Requirements Engineering Perspective. In P. Garbacz and O. Kutz (Eds.), *Proc. of the 8th International Conference on Formal Ontology in Information Systems* (Vol. *267*, pp. 317–329). Rio de Janeiro, RJ, Brasil: IOS Press.

Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H. & Bruel, J.-M. (2010). RELAX: a language to address uncertainty in self-adaptive systems requirement. *Requirements Engineering*, 15(2), 177–196.

Wohlin, C., et al. (2005). *Engineering and managing software requirements*. Springer Science & Business Media.

Zave, P. & Jackson, M. (1997). Four Dark Corners of Requirements Engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1), 1–30.