Ontological Foundations for Software Requirements with a Focus on 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 software execution 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 intend to clarify the main notions involved in RRT, establishing an explicit common conceptualization regarding this domain. To do so, we need first to understand what software execution is, as well as what requirements are. Thus, we present three related domain ontologies: the Software Ontology (SwO), an ontology about software nature and execution, the Reference Software Requirements Ontology (RSRO), which addresses what requirements are and types of requirements, and the Runtime Requirements Ontology (RRO), which extends SwO and RSRO to represent the nature and context of RRT. For developing these ontologies, we follow SABiO, a well-established Ontology Engineering method. Moreover, all the three ontologies are grounded in the Unified Foundational Ontology (UFO) and are integrated into the Software Engineering Ontology Network (SEON). Finally, as prescribed by SABiO, the ontologies are evaluated using ontology verification and validation techniques.

Keywords: Software Requirements, Runtime, Ontology, UFO, 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

---

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

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 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, it does not focus on maximizing certain computational properties. 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.

However, before capturing the conceptualization underlying the use of RRT, we need to understand two important notions: the first one is what does it mean to run software; the second one is what requirements really are. Thus, we decided to tackle this problem with a divide-and-conquer approach and develop three ontologies. The first, called Software Ontology (SwO), covers the part of the domain about software nature and how it is executed at runtime. The second, called Reference Software Requirements Ontologies (RSRO), focuses on requirements in general, their main types, and how they are documented. The third and most specific one, is called Runtime Requirements Ontology (RRO). RRO extends RSRO and SwO, focusing on the use of requirements during software execution.

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 re-engineering RRT approaches. Furthermore, they 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 inter-operate. All three ontologies are integrated into the Software Engineering Ontology Network (SEON) (Ruy et al., 2016).

SwO, RSRO and RRO were developed following the process defined by SABiO (Falbo, 2014), an ontology engineering method, and using UFO (Guizzardi, 2005) as foundational ontology. As recommended by SABiO, SwO, RSRO and RRO were evaluated by verification and validation activities. Verification was accomplished by checking if the Competency Questions initially raised were answered by the respective ontology. Validation was done by instantiating the ontologies with data extracted from real world projects using three frameworks that propose the use of requirements at runtime, namely FLAGS (Baresi et al., 2010), ReqMon (Robinson, 2006) and *Zanshin* (Souza, 2012). Finally. it is important to mention that the ontologies were built in this modular way (instead of one big ontology), so that they can be easily found and reused later.

This paper is an extended version of (Duarte et al., 2016). For this version, we present as main contribution three ontologies — SwO, RSRO and RRO — and integrate them 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 ontologies using other two RRT frameworks, improving validation. The verification process was also improved, through the definition of test cases.

The rest of the paper is structured as follows. Section 2 summarizes the general concepts of the RRT domain. In particular, it presents the results of systematic mapping of the literature we performed in the area. Section 3 describes the methodology used to build SwO, RSRO and RRO. Section 4 introduces the ontological foundations for the proposed ontologies. Sections 5, 6 and 7 present SwO, RSRO and RRO, the main contributions of this paper. Section 8 discusses how the ontologies were evaluated. Section 9 compares related work. Finally, Section 10 concludes the paper.

2. Background

This section presents, in Section 2.1, the basic concepts about the use of requirements at runtime that are responsible to provide the theoretical background for the development of this work. Section 2.2 presents the results of the systematic mapping that were performed to capture the consensual knowledge about the requirement at runtime domain. It also presents three frameworks that propose the use of requirements at runtime: *Zanshin*, ReqMon and FLAGS. These frameworks are used to instantiate SwO, RSRO and RRO, as a form of validation in Section 8. 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 a 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. *A Systematic Mapping of the Literature*

To identify proposals that use RRT, 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 1.

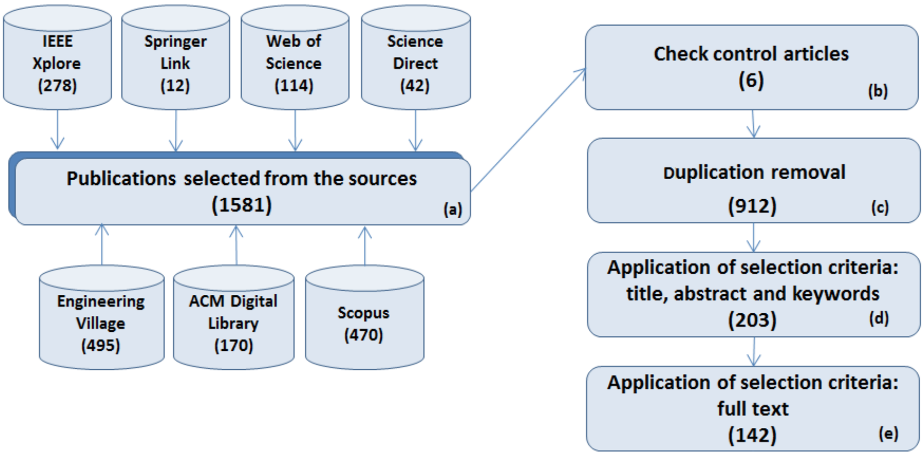


Fig. 1. Steps of the Systematic Mapping process on RRT.

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 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. These 142 publications were read, the research questions were answered and the information gathered with the systematic mapping were used as the primary source of knowledge to build RRO.

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 later 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.

2.3. *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 adaptivity 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 2 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.

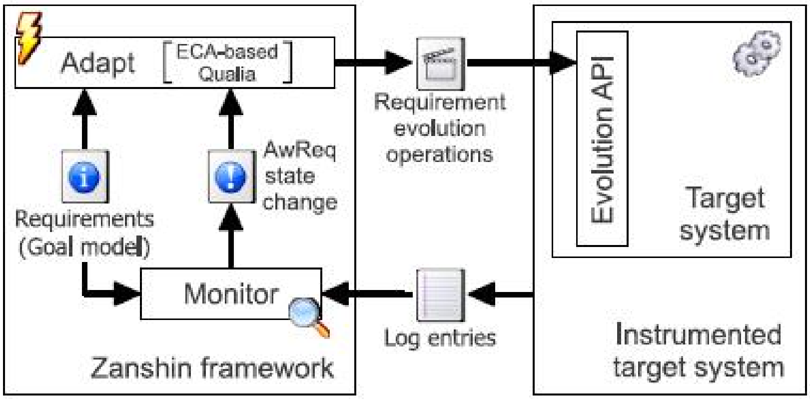


Fig. 2. *Zanshin*'s architecture representation (Souza, 2012)

Figure 3 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 their 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 3) 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. The rest of the model and the other elements of the modeling language are better described in (Souza, 2012).

2.4. *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.

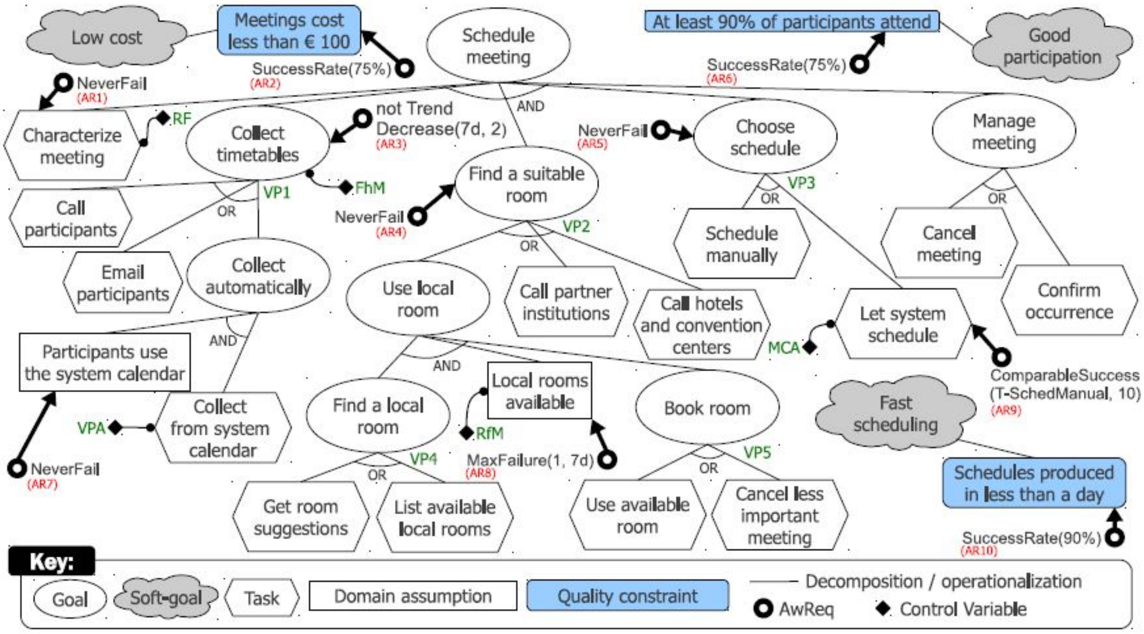


Fig. 3. Requirements model of a Meeting Scheduler, created with the *Zanshin* approach (Souza, 2012)

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.

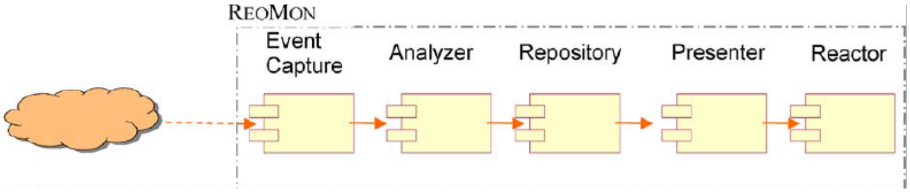
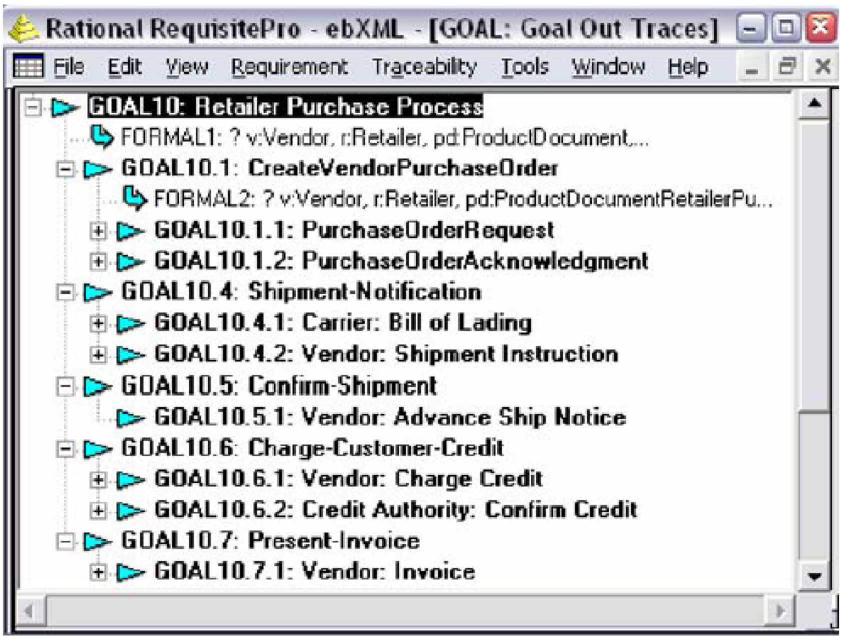


Fig. 4. ReqMon framework architecture (Robinson, 2006)

Figure 5 presents the Retailer Purchase goal tree, from an e-commerce application that is used in (Robinson, 2006) to illustrate the use of ReqMon. Since ReqMon uses a KAOS-based language to represent its requirements, all goals have a formal definition. The language, in order to account for runtime variability, makes use of temporal operators. At runtime, goals (the root-goal Retailer Purchase and its subgoals) can be monitored by a tool called ReqMon Designer Model.

Like *Zanshin*, ReqMon characterizes requirements as goals, also understanding them as complex entities that exist 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 former is focused on adaptations/evolution of a software system through changes in its requirements, while the latter 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.



GOAL10: Retailer Purchase Process
  FORMAL1: ? v:Vendor, r:Retailer, pd:ProductDocument,...
  GOAL10.1: CreateVendorPurchaseOrder
    FORMAL2: ? v:Vendor, r:Retailer, pd:ProductDocumentRetailerPu...
    GOAL10.1.1: PurchaseOrderRequest
    GOAL10.1.2: PurchaseOrderAcknowledgment
  GOAL10.4: Shipment-Notification
    GOAL10.4.1: Carrier: Bill of Lading
    GOAL10.4.2: Vendor: Shipment Instruction
  GOAL10.5: Confirm-Shipment
    GOAL10.5.1: Vendor: Advance Ship Notice
  GOAL10.6: Charge-Customer-Credit
    GOAL10.6.1: Vendor: Charge Credit
    GOAL10.6.2: Credit Authority: Confirm Credit
  GOAL10.7: Present-Invoice
    GOAL10.7.1: Vendor: Invoice

Fig. 5. ReqMon representation of Retailer Purchase goal tree (Robinson, 2006)

2.5. *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 6 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 7 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 at 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

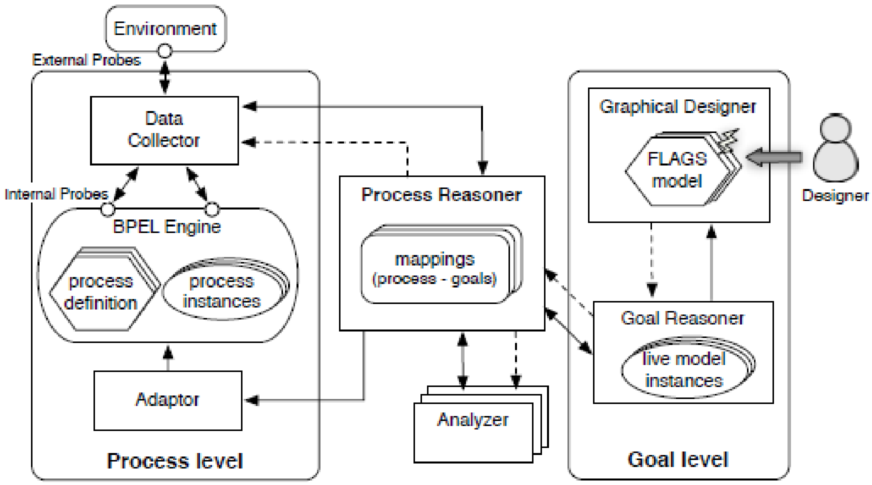


Fig. 6. FLAGS Runtime infrastructure (Pasquale et al., 2011)

needs to change. Other examples and a better description of the FLAGS modeling language can be found in (Baresi et al., 2010).

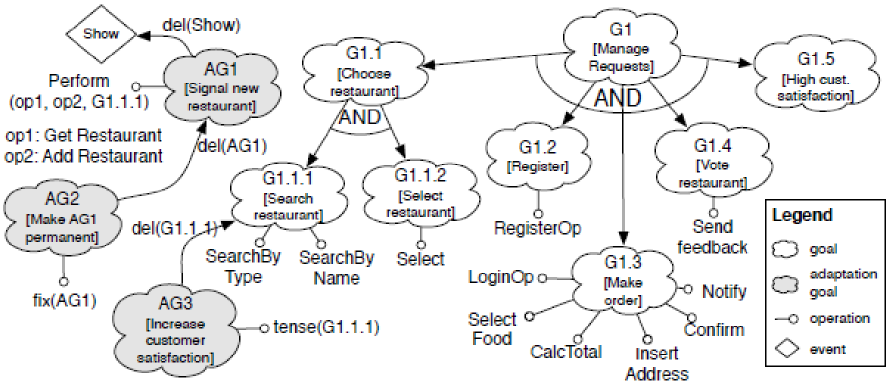


Fig. 7. FLAGS model for a Click&Eat web application (Pasquale et al., 2011)

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 7.

3. Methodology

To develop the ontologies presented in this paper, 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: Software Process Ontology – SPO (Bringuente et al., 2011), Software Measurement Ontology – RSMO (Barcellos et al., 2010) and the Reference Ontology on Software Testing – RooST (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 accompanied 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.

In the Purpose Identification and Requirements Elicitation phase, we started defining the purpose, scope and intended uses for each ontology. Next, we identified Competency Questions (CQs) as functional requirements for the domain ontologies. CQs are questions that the ontology should be able to answer (Grüninger and Fox, 1995). Analogously with the traditional Requirements Engineering (RE), besides representing functional requirements, CQs help to refine the scope of the ontology. They are also used in the ontology verification process to check if the ontology elements (concepts, relations, properties and axioms) are able and sufficient to answer the CQs (Falbo, 2014). Moreover, it is important to highlight that CQs for SwO, RSRO and RRO were refined during each ontology development process. CQs were raised through several meetings with domain experts and through the analysis of international standards (for RSRO) and results of the systematic mapping presented in Section 2 (for SwO and RRO), in a bottom-up strategy, starting with simpler questions and proceeding to find more complex ones.

As non-functional requirements for the three ontologies, we defined that they would: be grounded on a well-known foundational ontology (NFR1); be based on consensual work from the literature (NFR2); and to be integrated to the Software Engineering Ontology Network (SEON) (Ruy et al., 2016), reusing existing networked ontologies, or other relevant ontologies already published in the literature (NFR3).

For NFR1, we grounded our ontologies in the Unified Foundational Ontology (UFO) (Guizzardi, 2005; Guizzardi et al., 2008). UFO provides a set of entities and relations types, that are useful for Conceptual Modeling. Also, UFO is the foundational ontology that is suggested by SABiO. In Section 4, we present the fragment of UFO used to provide the ontological foundations for SwO, RSRO and RRO.

To satisfy NFR2, we followed different approaches for each ontology. In the case of SwO, we took Wang et al.'s Ontology of Software Artifacts (Wang et al., 2014) as basis, as well as the results of the systematic mapping we performed in the RRT domain. These results were useful, since they reveal very relevant notions related to software being executed at runtime. It is important to say that, although we were inspired by Wang et al.'s work, besides encompassing what software is and its artifactual nature — aspects addressed in (Wang et al., 2014) — SwO focuses also on software execution.

In the case of RSRO, we studied relevant standards, such as CMMI (SEI/CMU, 2010) and SWEBOK (Bourque et al., 2014), and selected publications, including classic Requirements Engineering papers (Kotonya and Sommerville, 1998; Wohlin et al., 2005; Robertson and Robertson, 2012). We chose these references because they are widely accepted by the RE community, including the software industry, and, in some sense, represent common aspects of the RE domain. Moreover, we explored the work of Guizzardi et al. (2014), who did an ontological interpretation of non-functional requirements based on definitions provided by UFO.

In the case of RRO, it is important to highlight that the RRT domain its fairly new (according to our research, around 16 years), in particular when compared with traditional Software Engineering. Thus, there is still a lack of standards and capability models that focus on the use of requirements at runtime. Based on that, we decided to investigate this domain by means of a systematic mapping of the literature (briefly reported in Section 2.2), and thus to gather consensual information about the RRT domain.

Regarding NFR3, to build both SwO and RSRO, we reused parts of SEON's core ontology on software process (SPO), in particular its sub-ontology about software artifacts, as well as the ontologies developed by Wang et al. (2014) regarding software artifacts (named here OSA), and by Guizzardi et al. (2014) concerning requirements (named here NFRO). This approach required us to integrate these ontologies to SEON's Software Artifact ontology. Finally, SwO and RSRO were reused to build RRO. In this way, all the resulting ontologies were integrated to SEON. This reuse and integration process is further discussed in Section 4.

Given the requirements and the results of the systematic mapping, we started capturing the concepts and relations of our ontologies, and building their conceptual models. This task was highly iterative and involved weekly meetings in which the primary aspects of the ontologies were discussed and defined. These meetings were attended by several specialists in the areas of Requirements Engineering and Ontology Engineering (expanding the initial group of four engineers involved in the systematic mapping), 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. The resulting reference models and primary products of this work are presented in Sections 5, 6 and 7.

Finally, 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 the three ontologies, we applied two evaluation approaches, namely: assessment by humans, and data-driven evaluation (Brank et al., 2005). For evaluating the reference ontologies as a whole (intended 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 the ontologies 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 discussed above, for building SwO and RSRO, we reused Wang et al.'s Ontology of Software Artifacts – OSA (Wang et al., 2014), and the ontological interpretation of non-functional requirements-related notions – NFRO (Guizzardi et al., 2014), as well as SEON's Artifact sub-ontology. Moreover, all three developed ontologies were grounded in UFO. In this section, we briefly discuss these ontological foundations.

4.1. UFO

The Unified Foundational Ontology – UFO (Guizzardi, 2005; Guizzardi et al., 2008) is a well-established foundational ontology that has been developed based on a number of theories from Formal Ontology, Philosophical Logics, Philosophy of Language, Linguistics and Cognitive Psychology. 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 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).

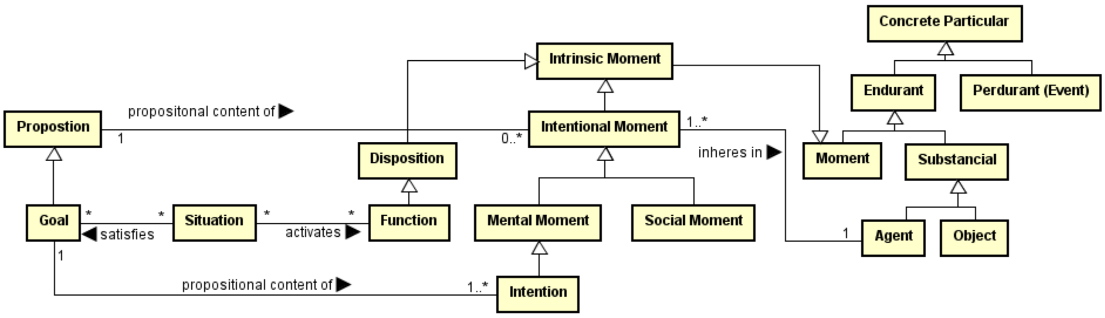


Fig. 8. Fragment of UFO showing Goals, Agents and Intentions.

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 et al., 2013, 2016).

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 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 are realizable through the occurrence of an Event. This occurrence brings about a Situation.

4.2. NFR Ontology

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. As Figure 9 shows, 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.

Moreover, the NFR Ontology is the only ontology we are aware of that combines the following characteristics: it is grounded on a foundational ontology (in fact, the same foundational ontology we use); provides a precise criteria for differentiating between FRs and NFRs; deals with vagueness in NFRs. These characteristics equip the ontology to support the process of requirements engineering by helping to clarify the elicitation and representation of software requirements as well as their satisfaction conditions (including the case of vague NFRs). By reusing this ontology, our proposal in this paper inherits these benefits.

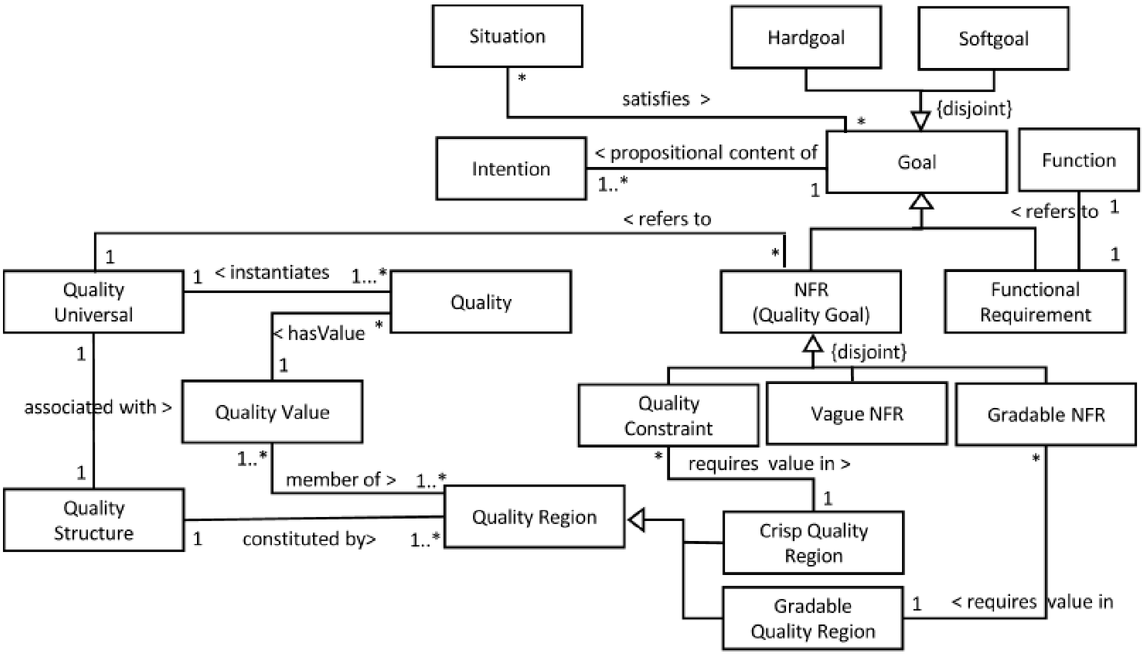


Fig. 9. Non-functional Requirements and related concepts (Guizzardi et al., 2014).

4.3. *Ontology of Software Artifacts*

In Wang et al.'s Ontology of Software Artifacts – OSA (Wang et al., 2014), software is a special kind of entity that is capable of changing while maintaining its numerical identity. These changes (i.e., a simple bug fix 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 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 10, 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: 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 completely mappable to UFO. Thus, we could reuse it straightforwardly. Furthermore, OSA is the

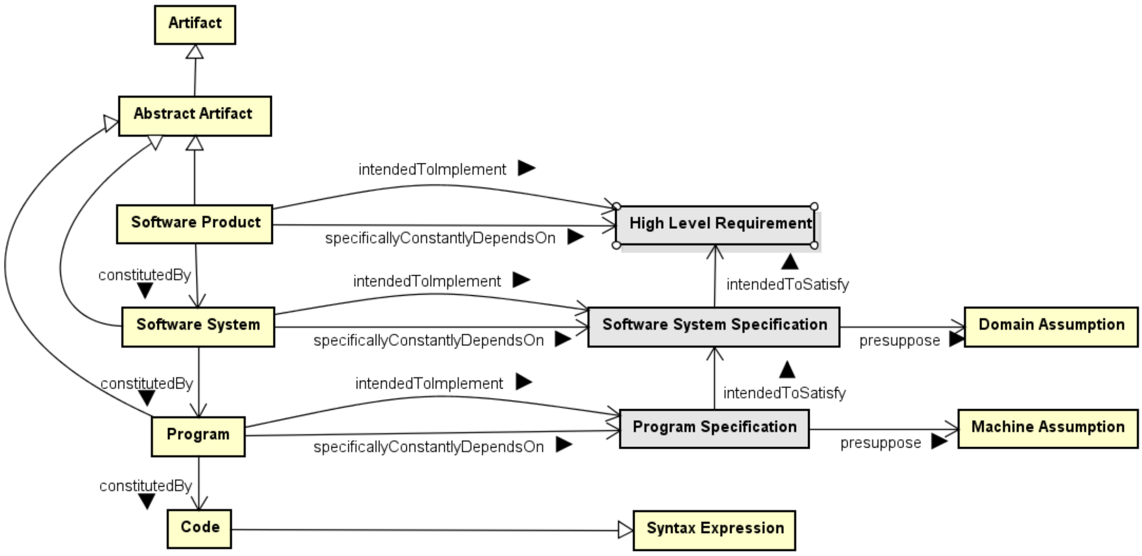


Fig. 10. Fragment of the Ontology of Software Artifacts (Wang et al., 2014).

only ontology we are aware of that combines the following characteristics: it is grounded in a foundational ontology (whose portion used by OSA is mappable to the one we use); it explicitly accounts for the different identity conditions and dependence relations of the multiple artifacts involved in a software ecosystem; and is built over the background of an established requirements engineering framework (the model of Zave and Jackson (1997)).

The use of OSA can bring several benefits to the work proposed here. Firstly, because it can potentially allow us (in a future extension to this ontology) to differentiate among the requirements that can be related to different software-associated artifacts (e.g., licensed software products requirements versus program requirements). Secondly, as discussed by Wang et al. (2014), by properly distinguishing all these kinds of software-associated artifacts, OSA can support a next-generation of more advanced software configuration management processes and tools. By reusing the OSA ontology, the models proposed here inherit these finer-grained distinctions made by the original ontology. For instance, as a by-product of this process, this could afford improving the management of several potential configuration items that appear in the proposed ontologies (e.g., different types of requirements specification) by associating them with different software-associated artifacts.

4.4. SEON

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 to: (i) take advantage of well-founded ontologies (all its ontologies are ultimately grounded in UFO); (ii) provide ontology reusability and productivity, supported by core ontologies organized as Ontology Pattern Languages (Falbo et al., 2016); and (iii) solve ontology integration problems by providing integration mechanisms (Ruy et al., 2016).

In its current version, SEON has a core ontology of software processes (SPO), as well as domain ontologies for the main technical Software Engineering sub-domains, namely requirements, design, coding and testing, and for some management sub-domains, namely software measurement, project management, configuration management, and quality assurance. Figure 11 presents an overview of SEON (its complete specification is available at `https://nemo.inf.ufes.br/projects/seon/`). Each circle represents an ontology. The circle in the center is the core ontology, 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 (in terms of number of relationships between concepts in different ontologies).

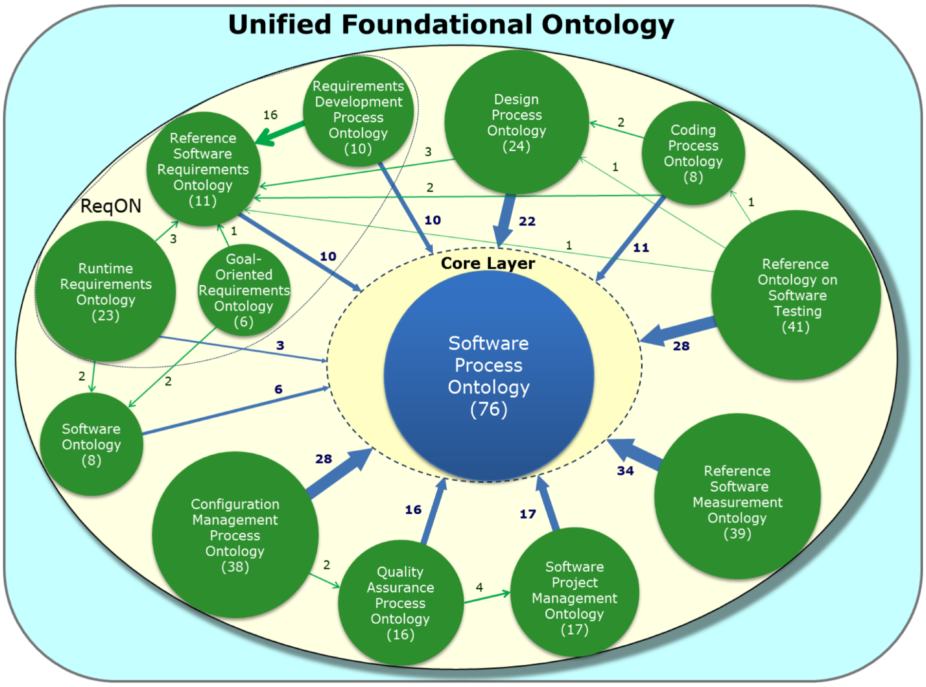


Fig. 11. SEON: The Network view (Ruy et al., 2016).

Concerning requirements, ReqON is SEON's ontology subnetwork devoted to requirements. The Reference Software Requirements Ontology (RSRO), presented in Section 6, is the main 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 7, addresses the use of requirements at runtime.

RSRO and SwO extend SEON's Software Process Ontology (SPO), more precisely, SPO's Artifacts sub-ontology. As Figure 12 shows, Artifacts are objects intentionally made to serve a given purpose in the context of a software project or organization. Stakeholders may be responsible for them. Artifacts can be simple or composite, depending on their mereological structure. A Composite Artifact is an Artifact that is composed by two or more Artifacts. On the other hand, a Simple Artifact is an atomic one, an Artifact that cannot be decomposed in others. In other words, if an Artifact is composite, then it must have at least two disjoint (non-overlapping) parts, which can be simple, composite or any combination of both.

SPO also classifies artifacts according to their natures. A Software Product is defined as one or more computer Programs together with any accompanying auxiliary items, such as documentation, delivered under a single name, ready for use (e.g., Microsoft Word, Eclipse IDE). A Software Item is a piece of software, produced during the software process, not considered a complete Software Product, but an intermediary result (e.g., a component). A Document, in turn, is any written or pictorial, uniquely identified information related to the software development, usually presented in a predefined format (e.g.,

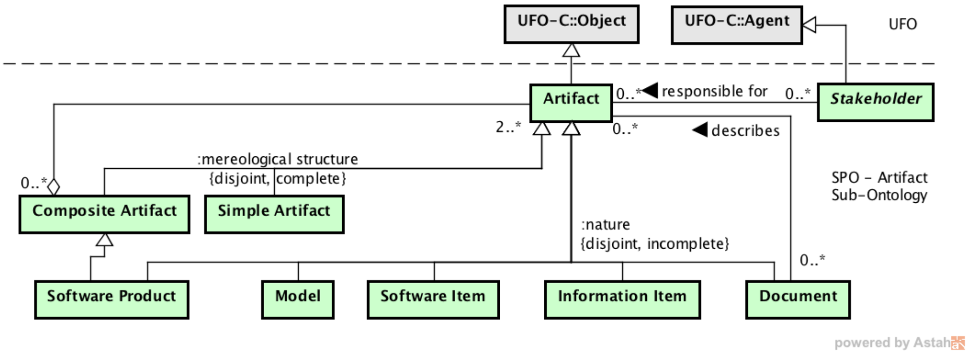


Fig. 12. SEON's view regarding Software and Related Artifacts.

a requirements document, a report). An Information Item is a piece of relevant information for human use, produced or used by an activity (e.g, a component description, a bug report). A Model is a representation (abstraction) of a process or system from a particular perspective (e.g., a use case model, a class model).

In Section 5, we present SwO, the SEON's Software Ontology, which is strongly based on OSA. In fact, SwO incorporates parts of OSA to SEON.

5. Software Ontology (SwO)

The Software Ontology (SwO) aims to capture the complex artifactual nature of software. Software, is a complex entity constituted of different types of artifacts, such as software systems, programs and code. SwO is a domain reference ontology that aims to explain this complex software nature and how it is materialized inside a computer, being executed and producing results that go beyond the limits of the machine, directly affecting the real world. Since it is a fairly general ontology, closer to the core level than RSRO and RRO, SwO is widely reusable for developing more specific domain ontologies, as it is the case of RRO, but also to describe other types of software, such as Software Service and Multimedia Systems. SwO aims to answer the following CQs:

- **CQ1:** What is a software product?
- **CQ2:** What is a software system?
- **CQ3:** What is a program?
- **CQ4:** When is a program functionally in conformance with its specification?
- **CQ5:** What is a running program?
- **CQ6:** Where does a running program execute?
- **CQ7:** What can be observed from a running program execution?

As Figure 13 shows, SwO extends part of the Software Artifact sub-ontology of SPO, while incorporating concepts from OSA. In this figure, concepts from SwO are shown in white (those reused from OSA are prefixed by OSA), concepts from SPO are shown in green, and concepts from UFO are shown in grey.

As in OSA, a Software Product is constituted of Software Systems. However, in SPO, it is made explicit that, as a Composite Artifact, Software Product includes also other artifacts, such as Documents, Models and Software Items. Thus, a Software Product is constituted of one or more Software Systems, together with any accompanying auxiliary items, such as documentation, delivered under a single name, ready for use (e.g., Astah modeling tool, Microsoft Word, Eclipse IDE).

Like in OSA, a Software System (a subtype of Software Item in SwO) is constituted of Programs, and intends to implement a System Specification (a subtype of Document in SwO). A Program, in turn, is also defined as a Software Item, a piece of software, produced during the software process, not considered a complete Software System, but which aims at producing a certain result through execution on a computer, in a particular way, given by the Program Specification, which is a Document that

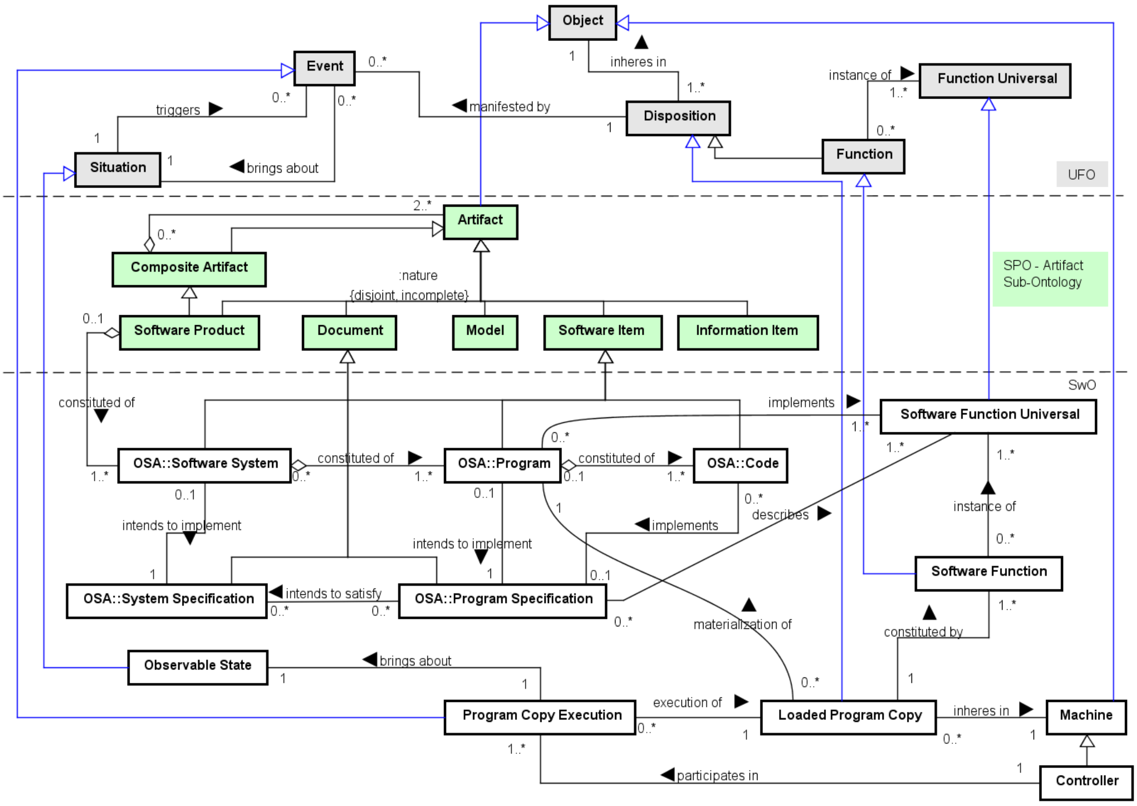


Fig. 13. Conceptual model fragment of SwO.

belongs to the solution domain, describing structural and functional information about the Program with enough detail that would allow implementation and maintenance. A Program is constituted by Code (a subtype of Software Item) written for a specific machine environment (e.g., Microsoft Excel for MacOS). A portion of Code can implement a Program Specification, which, in turn, may intend to satisfy a System Specification.

Since a Program Specification describes a set of Software Function Universals, we can introduce another important notion: the satisfaction of the specification. Axiom A1 captures this notion in SwO: 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.

$$\begin{aligned}&\forall p\colon Program, ps\colon ProgramSpecification \quad functionallyConformant(p, ps)\\ &\iff (\forall sfu\colon SoftwareFunctionUniversal \quad describes(ps, sfu)\\ &\leftrightarrow implements(p, sfu))\end{aligned} \tag{A1}$$

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; Guizzardi et al., 2016).

In SwO, a 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 A2: 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.

$$\begin{aligned}&\forall lpc\colon LoadedProgramCopy, p\colon Program \quad materializationOf(lpc, p)\\ &\quad\rightarrow (\forall sfu\colon SoftwareFunctionUniversal \quad implements(p, sfu)\\ &\quad\rightarrow \exists sf\colon SoftwareFunction \quad instanceOf(sf, sfu) \wedge constitutedBy(lpc, sf))\end{aligned} \tag{A2}$$

As in Irmak (2013), a Program Copy Execution is an Event representing the execution of the Loaded Program Copy running in a Machine. Thus, 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 by Axiom A3.

$$\begin{aligned}&\forall pce\colon ProgramCopyExecution, lpc\colon LoadedProgramCopy, m\colon Machine\\ &\quad executionOf(pce, lpc) \wedge inheresIn(lpc, m) \rightarrow participatesIn(m, pce) \wedge Controller(m))\end{aligned} \tag{A3}$$

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. 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).

As mentioned earlier in Section 4, SwO is strongly based on OSA. It incorporates several concepts presented in OSA, such as Software System, System Specification, Program, Program Specification and Code, to SEON. However, there are differences that need to be explained.

The main difference is that in SEON (and thus SPO and SwO), we assume a software process view on software artifacts. This means that software artifacts are intentionally produced by activities of the software process, and that these artifacts must be made explicit. This explains why we consider System and Program Specification as Documents. In SEON, for them to be considered as Artifacts, these specifications should be documented, as recommended by software quality models and standards. In OSA, on the other hand, specifications may exist only in the developer's mind. Moreover, in OSA, Code is considered just a Syntax Expression, which is not an Artifact. However, in SwO, Code is considered an Artifact (more precisely, a Software Item), since it is also intentionally produced by activities of the software development process.

Another difference is that OSA does not handle complex artifacts being constituted of artifacts of the same time. For instance, a Program can only be constituted by Code, it cannot be constituted by other programs. This is also true for the other artifacts that are presented in OSA, namely Software System (that is constituted by one or many Programs) and Software Product (that is constituted by one or many Software Systems). Thus, OSA does not care about concepts such as sub-system or sub-program. SwO, on the other hand, admits the existence of sub-systems and sub-programs, since artifacts can be simple or composite.

6. 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. These notions basically regard the what, how and who w.r.t. requirements, i.e. what a requirement actually is and how they may be classified, who is interested in which requirement, and how requirements and their context (i.e. assumptions) are documented. Moreover, just like SwO, RSRO was developed to be reused and extended by other ontologies, representing the many facets of Requirements Engineering, such as Goal-Oriented Requirements Engineering approaches (Negri et al., 2017) or the use of requirements at runtime. To be as general as possible, the scope of RSRO is very narrow, and it aims to answer only the following competency questions:

- **CQ8:** What is a requirement?
- **CQ9:** What are the main types of requirements?
- **CQ10:** How are requirements documented?
- **CQ11:** Who are the main stakeholders involved with requirements?
- **CQ12:** What is an assumption in the context of RE?

Figure 14 shows RSRO's conceptual model. In this figure, concepts from RSRO are shown in blue, while concepts from other ontologies follow the color convention introduced in Section 5. 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.

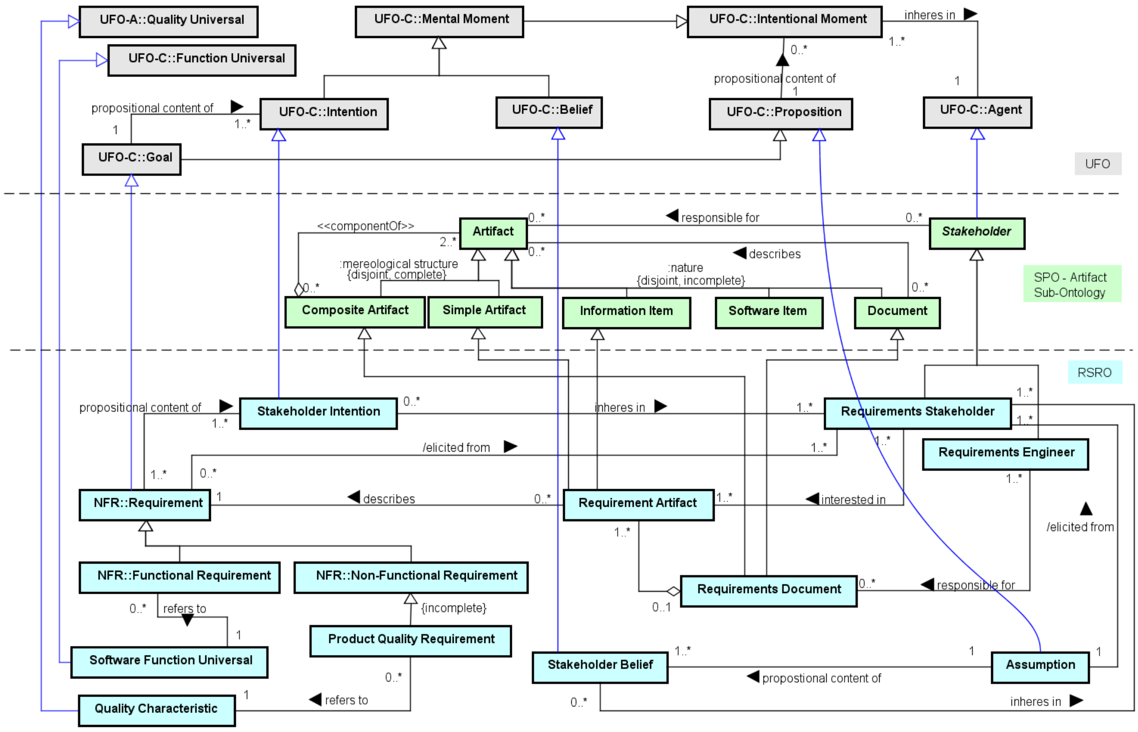


Fig. 14. Conceptual Model of RSRO.

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 elicited from her. Axiom A4 describes the derived relation[1] *elicited from*:

$$\begin{aligned}&\forall r\colon Requirement, ri\colon RequirementIntention, rs\colon RequirementStakeholder\\&\quad propositionalContentOf(r, ri) \land inheresIn(ri, rs) \rightarrow elicitedFrom(r, rs)\end{aligned} \tag{A4}$$

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 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 A5 holds: if a Requirement *r* is elicited from a Requirements Stakeholder *rs*, and *r* is described in the Requirements Artifact *ra*, then *rs* is interested in *ra*.

$$\begin{aligned}&\forall r\colon Requirement, rs\colon RequirementStakeholder, ra\colon RequirementArtifact,\\&\quad elicitedFrom(r, rs) \land describes(ra, r) \rightarrow interestedIn(rs, ra)\end{aligned} \tag{A5}$$

It is important to say that, although Axiom A5 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 whom the corresponding requirement has been elicited.

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 an 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 system 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 A6 and A7 hold:

$$\begin{aligned}&\forall fr\colon FunctionalRequirement, ra\colon RequirementArtifact,\\&\quad describes(ra, fr) \rightarrow FunctionalRequirementArtifact(ra)\end{aligned} \tag{A6}$$

$$\begin{aligned}&\forall nfr\colon NonFunctionalRequirement, ra\colon RequirementArtifact,\\&\quad describes(ra, nfr) \rightarrow NonFunctionalRequirementArtifact(ra)\end{aligned} \tag{A7}$$

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 certain configuration of properties, i.e., a situation type, exists in the environment. Representing such situations is sometimes necessary because they need to be considered in a given solution to a specific problem. For a deeper discussion about assumptions, see (Negri et al., 2017).

---

[1] Derived relations, i.e., relations deduced by derivation rules, are represented in UML diagrams preceded by a "/"

7. 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:

- **CQ13:** What is the relation between a program and its requirements?
- **CQ14:** What is a requirement at runtime?
- **CQ15:** How are requirements used at runtime?
- **CQ16:** Which program is the target of a program that uses runtime requirements?
- **CQ17:** When is a program that uses RRT considered internal or external to a system?

Figure 15 shows RRO's conceptual model. In this figure, concepts introduced in RRO are shown in yellow, while concepts from the other ontologies follow the previously used color convention.

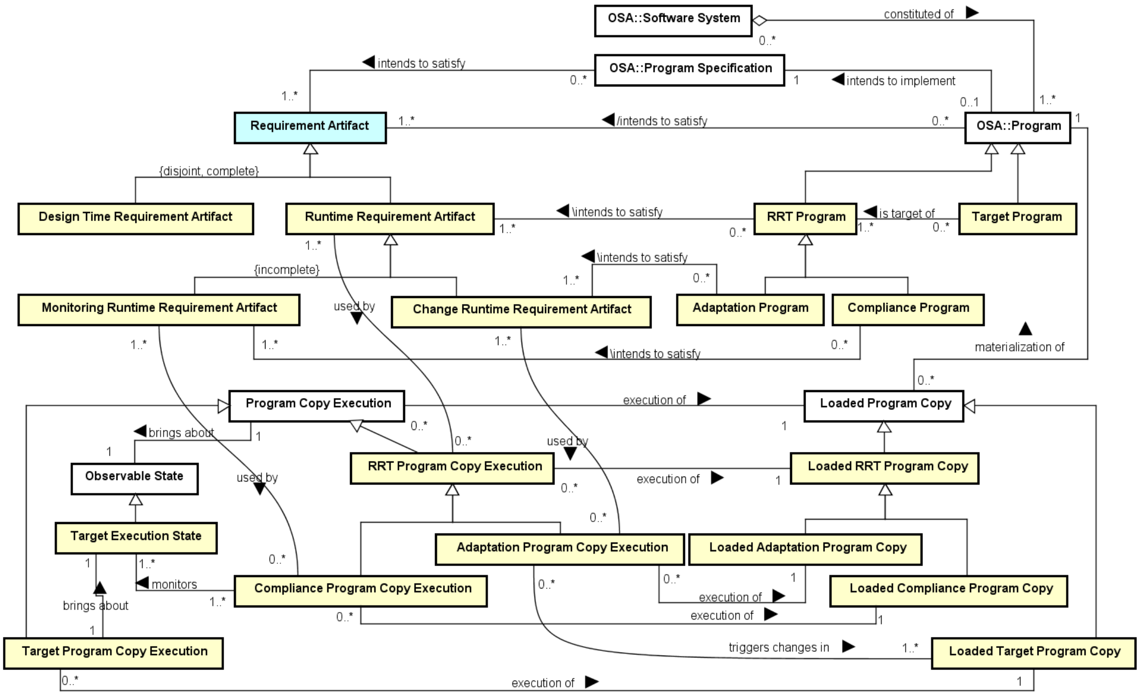


Fig. 15. Conceptual Model of RRO.

An important distinction of RRO 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 at 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.

For properly linking requirements to its use at runtime, we need to relate Program (from SwO) to Requirement Artifact (from RSRO). A Program Specification intends to satisfy some Requirement Artifacts. Thus, a Program that intends to implement a Program Specification also intends to satisfy these Requirement Artifacts, and thus, Axiom A8 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.

$$\begin{aligned}&\forall p\colon Program, ps\colon ProgramSpecification, ra\colon RequirementArtifact\\&\quad intendsToSatisfy(ps, ra) \land intendsToImplement(p, ps) \rightarrow intendsToSatisfy(p, ra)\end{aligned} \tag{A8}$$

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 a Execution State that is 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. In other words, in the RRT domain (cf. Section 2), requirements are used not only to monitor (which can be done by observing the *events* of execution, 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 Target Program Copy as the materialization of a Target Program, inhering in a Machine, e.g., a copy of Microsoft Excel loaded in primary memory by MacOS on my MacBook, which will, at some point, be executed again with its new/changed properties, that were changed based on the adaptation guideline that is described in a Change Runtime Requirement Artifact. This new execution with new properties will produce a new Execution State, that may (or may not) be in conformance with the requirements, that were not being fulfilled before de adaptation.

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 many of these 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:

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

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

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.

8. Evaluation

To evaluate SwO,RSRO and RRO, we performed Ontology Verification & Validation (V&V) activities. Considering the guidelines proposed by SABiO (Falbo, 2014), all three ontologies 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 SwO, RSRO and 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. In what follows, each one of these evaluation steps are described in more details.

8.1. Verification by Experts

For verifying SwO, 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 SwO, showing which elements of the ontology (concepts, relations, properties and axioms) answer each one of the Competency Questions (CQs) of SwO. Analogously, Table 2 and Table 3 present the verification process for RSRO and RRO, respectively. These tables can also be used as a traceability tool, supporting ontology change management.

Table 1

SwO Verification against its Competency Questions.

CQs	**Description, Concepts and *Relations***	**Axioms**
CQ1	**What is a software product?** Software Product is a Composite Artifact *constituted by* Software Systems and other Artifacts.	–
CQ2	**What is a software system?** Software System is *constituted by* Programs and *intends to implement* a System Specification.	–
CQ3	**What is a program?** Program is *subtype of* Software Item, *constituted by* Code and *intends to implement* a Program Specification.	–
CQ4	**When is a program functionally in conformance with its specification?** Program *implements* Software Function Universals and *intends to implement* Program Specification that *describes* Software Function Universals	A1
CQ5	**What is a running program?** Loaded Program Copy is a *materialization of* Program that is *constituted by* Software Functions. Program *implements* Software Function Universals, while Loaded Program Copy is *constituted by* Software Function. Program Copy Execution is an *execution of* Loaded Program Copy.	A2
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.	A3
CQ7	**What can be observed from a running program execution?** Program Copy Execution *brings about* an Observable State.	–

8.2. Validation

Concerning ontology validation, SABiO suggests that the ontology should be capable of properly representing 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 4 illustrates the results of the instantiation with the Zanshin Meeting Scheduler example, Table 5 presents the instantiation for the e-commerce application, and Table 6 presents the instantiation using the 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. Also, it is important to mention that these tables present concepts from all three ontologies. Thus, SwO, RSRO and RRO are being validated at the same time. Since RRO extends and reuses many of SwO's and RSRO's concepts, we believe that a separated validation would bring no advantage.

The successful instantiation of SwO, RSRO and 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.

Table 2
RSRO Verification against its Competency Questions.

CQs	Description, **Concepts** and ***Relations***	Axioms
CQ8	**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.	A4
CQ9	**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. Product Quality Requirement is a *subtype of* Non-Functional Requirement that *refers to* a Quality Characteristic.	–
CQ10	**How are requirements documented?** Requirements Document is composed of Requirements Artifacts that *describe* Requirements. Functional and Non-Functional Requirement are *subtypes of* Requirement.	A6, A7
CQ11	**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.	A5
CQ12	**What is an assumption in the context of RE?** Assumption is a Proposition that is the *propositional content of* an Stakeholder Belief that, in turn, *inheres in* a Requirements Stakeholder	–

Table 3
RRO Verification against its Competency Questions.

CQs	Description, **Concepts** and ***Relations***	Axioms
CQ13	**What is the relation between a program and its requirements?** Program is *intends to implement* a Program Specification, which, in turn, *intends to satisfy* a Requirement Artifact.	–
CQ14	**What is a requirement at runtime?** Runtime Requirement Artifact is a *subtype of* Requirement Artifact that are *intended to be satisfied by* a RRT Program. Monitoring Runtime Requirement Artifact is a *subtype of* Runtime Requirement Artifact that *is intended to be satisfied* by a Compliance Program. Change Runtime Requirement Artifact is a *subtype of* Runtime Requirement Artifact that *is intended to be satisfied* by an Adaptation Program.	–
CQ15	**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.	–
CQ16	**Which program is the target of a program that uses runtime requirements?** Target Program is a *subtype of* Program. Target Program *is target of* RRT Program.	–
CQ17	**When is a program that uses RRT considered internal or external to a system?** An 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

8.3. Test Case Designs

Considering the instances of concepts used to validate SwO, RSRO and RRO, 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 7 shows examples of test cases for CQ16 that we designed to promote a better understanding of the method.

Table 4

Results of Swo, RSRO and 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.
Functional Requirement	Meeting Scheduler should send e-mails for all meeting participants 2 hours before the meeting.
Product Quality Requirement	Meetings should be able to be scheduled in 2 screens.
Quality Characteristic	Usability (refers to the Product Quality Requirement above).
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.
Requirement Stakeholder	An Area Manager of a Company
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.

It is important to say that, if any of the three ontologies proposed in this paper 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 16 was chosen as an example to demonstrate test cases, which should be implemented for all the CQs of an ontology. This test case was designed and presented in this work in order to show that other forms of evaluation are applicable when the operational version of the ontology is being used. This particular CQ was chosen because it has multiple, although fairly simple, answers (one for each RRT framework presented).

9. 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 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 com-

Table 5
Results of SwO, RSRO and 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.
Functional Requirement	Shop cart screen should have a discount coupon field.
Product Quality Requirement	E-commerce website should be up 99.5% of the time and should support up to 500 simultaneous users.
Quality Characteristic	Reliability (refers to the Product Quality Requirement above).
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.
Requirement Stakeholder	An E-commerce Website Owner
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

pleted" (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 (Gillain et al., 2013; Qureshi et al., 2011), including, e.g., *Zanshin* (Souza, 2012). 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

Table 6

Results of SwO, RSRO and 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.
Functional Requirement	Restaurant Owners should be able to manage special/limited offers in the system.
Product Quality Requirement	Click&Eat mobile App should have versions for iOS and Android. Also, it should be accessible through most used Internet Browsers (Chrome, Safari, Firefox, Edge and Opera)
Quality Characteristic	Portability (refers to the Product Quality Requirement above)
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.
Requirement Stakeholder	A Restaurant Owner
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 7

Example test cases for RRO competency question CQ16 - *Which program is the target of a program that uses runtime requirements?*

Test Case Id	**Example**	**Input**	**Expected Output**
T11.1	Meeting Scheduler (Table 4)	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 5)	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 6)	Adaptor component (RRT Program)	The Click&Eat application for a specific machine (e.g: an iPhone App) (Target Program).

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.

10. Conclusions

The main contribution of this paper is the definition of RSRO, SwO and RRO. The first is a domain reference ontology about the use of requirements and the artifacts that describe them. The second is an ontology about software artifacts and their execution of software, at runtime. 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. SwO, RSRO and RRO are integrated into the Software Engineering Ontology Network (SEON) and have 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 SwO, 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.

Falbo, R.A., Barcellos, M.P., Ruy, F.B., Guizzardi, G. & Guizzardi, R.S.S. (2016). Ontology pattern languages. In A. Gangemi, P. Hitzler, K. Janowicz, A. Krisnadhi and V. Presutti (Eds.), *Ontology Engineering with Ontology Design Patterns: Foundations and Applications*. IOS Press.

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., Guarino, N. & Almeida, J.P.A. (2016). Ontological Considerations about the Representation of Events and Endurants in Business Models. In *International Conference on Business Process Management* (pp. 20–36). Springer.

Guizzardi, G., Wagner, G., Falbo, R.d.A., 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`.

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.