From a Scrum Reference Ontology to the Integration of Applications for Data-Driven Software Development

Paulo Sérgio Santos Júnior[1,2], Monalessa Perini Barcellos[1],
Ricardo de Almeida Falbo[1], João Paulo A. Almeida[1]

[1] Ontology and Conceptual Modeling Research Group (NEMO), Department of Computer Science, Federal University of Espírito Santo, Vitória – ES, Brazil
[2] Department of Computer Science, Federal Institute of Espírito Santo, Serra – ES, Brazil
paulo.junior@ifes.edu.br; {monalessa, falbo, jpalmeida}@inf.ufes.br

Abstract

Context: Organizations often use different applications to support the Scrum process, including project management tools, source repository and quality assessment tools. These applications store useful data for decision-making. However, data items often remain spread in different applications, each of which adopt different data and behavioral models, posing a barrier for integrated data usage. As a consequence, data-driven decisions in agile development are uncommon, missing valuable opportunities for informed decision making. Objective: Considering the need to address semantic issues to properly integrate applications that support the agile development process, we aim to provide a common and comprehensive conceptualization about Scrum in the software development context and apply this conceptualization to support application integration. Method: We have developed the Scrum Reference Ontology (SRO) and used it to semantically integrate Azure DevOps and Clockify. Results: SRO served as a reference model to build software artifacts in a semantic integration architecture that enables applications to automatically share, exchange and combine data and services. The integrated solution was used in the software development unit of a Brazilian government agency. Results demonstrate that the integrated solution contributed to improving estimates, provided data that helped allocate teams, manage team productivity and project performance, and enabled to identify and fix problems in the Scrum process execution. Conclusions: SRO can serve as an interlingua for application integration in the context of Scrum-process support. By capturing the conceptualization underlying Scrum, the reference ontology can address semantic conflicts and thereby support the development of integrated data-driven solutions for decision making.

Keywords: Ontology, Scrum, Semantic Interoperability, Application Integration.

1 Introduction

Agile methods have been increasingly adopted in software development because they enable organizations to deliver valuable products to clients in short iterative cycles [1][2]. There are several agile methods, practices and frameworks (e.g., Scrum, XP, Kanban, Safe, LeSS). Scrum has been used by many software organizations and is the most popular agile method in this domain [1].

Organizations use different applications[1] to support different aspects of the agile software process [3]. For example, project management and time-tracking applications are used to support project management, while integrated development environments, version control tools and code quality tools are used to support coding. The intensive use of applications in software development creates opportunities involving the various kinds of data they store, enabling data-driven decision making [4]. For example, data regarding code quality (e.g., number of defects, number of smells, etc.) and rework (effort spent fixing errors made during the development process) could provide useful information to support decisions about testing and coding strategies.

According to Svensson et al. [5], despite the vast amount of data stored in applications, decisions related to software development are commonly based on subjective aspects, such as previous experiences of the

[1] In this paper, the terms application, tool and system are used as synonymous.

managers and stakeholders, intuitions or a combination of these. One of the reasons organizations fail to leverage data stored in applications is the difficulty to access, integrate, analyze and view data handled by heterogeneous applications. In general, each application implements its own data and behavioral models and focuses on specific aspects of the software process, with little concern for sharing and integration aspects, leading thus to several conflicts [6]. Particularly in the agile development context, the challenge is to use data to support decision making in such a way that does not represent a bottleneck to the process agility. There is a need to convert data stored in the applications into useful information, and to present it to the team within their development environment, in an effective and proactive way [7]. Hence, to extract and integrate stored data automatically and to present integrated data in meaningful dashboards, without requiring extra effort from the team, is a good strategy. Using data to support the development process and decision making has been recognized as an important issue in agile development to achieve the project and organization goals. In the last years, the evolution of agile development has been approached in the context of Continuous Software Engineering (e.g.,[8], [9] and [10]), where the use of development and customer-related data to support daily activities and decision making is considered of key relevance.

One source of difficulty for data integration is semantic heterogeneity, which can result in conflicts whenever the same information item is given divergent interpretations, a situation which may not even be detected [11]. Neglecting these "semantic conflicts" can lead to integrated solutions that fail in achieving their purposes (e.g., if applications are not properly integrated, they can provide wrong information for decision-making) [12]. To reduce these conflicts, integration should address semantic issues [3][6]. For addressing these problems, ontologies can be used to establish a common conceptualization about the domain in order to support communication and applications integration. They can be used as an interlingua to map the concepts used by different applications, enabling data and services understanding [6]. In fact, in the last decade, ontologies have become the predominant way to deal with semantic aspects in semantic integration initiatives [13].

In this paper, we introduce the *Scrum Reference Ontology* (SRO), which provides a common and comprehensive conceptualization about Scrum in the software development context and can be used to support application integration. Although there are in the literature some ontologies addressing Scrum (e.g.,[14][15][16]), they cover only general concepts about agile development, not detailing the Scrum process, the various roles played in the process and the artifacts manipulated.

SRO was built as a networked ontology in the Software Engineering Ontology Network (SEON) [17]. SEON is an ontology network that (i) describes various subdomains of the Software Engineering domain (e.g., Software Requirements, Software Process, Software Measurement); (ii) offers mechanisms to facilitate building and integrating new SE subdomain ontologies to the network; and (iii) promotes integration by keeping a consistent semantics for concepts and relations along the whole network [17]. Since existing ontologies in SEON describe general concepts in Software Engineering, SRO can focus on specifics of Scrum, while reusing notions such as "software project", "project team" and "requirement".

In order to demonstrate the applicability of SRO in a realistic setting, we employed SRO to integrate applications to support the Scrum software development process and provide information for decision making in a project at the software development unit of Prodest[2], a Brazilian government agency responsible for IT (Information Technology) solutions in the state of Espírito Santo. In this demonstration, the following applications were chosen: Azure DevOps[3] and Clockify[4]. Azure DevOps supports project management in agile software projects. Clockify, in turn, supports time-tracking. These applications have been used in projects in at Prodest in a complementary way. While Azure DevOps has been used to aid in project management in general (e.g., to create a new project, to record scope by means of user stories, to define tasks and to allocate them to a team), Clockify has been used to enable detailed control of tasks duration, schedule, effort and cost. Thus, general data about the project (e.g., concerning sprints, user stories and related tasks) is handled by Azure DevOps, while detailed data about how the tasks were performed over time (e.g., concerning time entries, effort and cost related to each time entry) is handled by Clockify. Since the applications were not integrated, redundant data needed to be manually entered in both of them (e.g., the same task was created in both

[2] *https://www.prodest.gov.br*
[3] *https://azure.microsoft.com/pt-br/services/devops/*
[4] *https://clockify.me*

applications), and when integrated data was needed (e.g., data about how tasks that implemented user stories of a given sprint were performed), human intervention was needed to retrieve data from both applications and integrate them by using spreadsheets. We used SRO to build a solution that integrates Azure DevOps and Clockify, enabling automatic sharing and exchange of data between the applications and providing consolidate data useful for decision making. The successful use of SRO in this integration initiative provided initial evidence that the ontology is suitable for integration purposes.

This paper presents SRO and its use to integrate the two applications. With SRO, we intend to contribute to researchers and practitioners by providing a comprehensive and well-founded conceptualization about Scrum, which can be used for communication and to support application integration, among other purposes. With the integration solution produced by using SRO, we demonstrate SRO's use in a real-world context and provide an example of how it can be used to support application integration. The paper is organized as follows: Section 2 presents the theorical background for the paper, addressing aspects related to Scrum, application integration and ontologies; Section 3 describes the research method adopted in this work; Section 4 presents SRO; Section 5 describes the integrated solution produced using SRO to integrate Azure DevOps and Clockify and its use in a project at Prodest; Section 6 discusses related work and; finally, Section 7 presents our final considerations and outlines future work.

2 Background

2.1 Scrum

Scrum was created with the assumption that software development is too complex and unpredictable to be fully planned at the beginning of a project. Therefore, it employs an iterative, incremental approach to optimize predictability and control risk [2]. A *Scrum team* is a flexible, adaptive and small team (usually up to 7 people). Scrum teams are self-organized, cross-functional and capable of delivering products iteratively and incrementally, maximizing opportunity for continued feedback. A Scrum team is composed of a *product owner*, which is the role played by a person acting on behalf of the client and responsible for maximizing the value of the developed product, and a *development team*, which is responsible for developing the product. The development team, in turn, is composed of *developers* and a *Scrum master*. The *Scrum master* is a facilitator who ensures that the development team is provided with an adequate environment to complete the project successfully [2].

The Scrum process starts with an initial planning to establish the product requirements and record them ordered in the *product backlog* [18]. The product backlog is never complete, and it can constantly change [2]. The project is developed through incremental time-boxed cycles (usually lasting one month or less) called *sprints.* For each sprint, there is a *sprint planning meeting*, when the team selects from the product backlog the items to be addressed in the sprint and plans the work to be done. The planning result is recorded in the *sprint backlog*. A sprint produces a visible, usable, deliverable product that implements one or more user interactions with the system. The key idea behind a sprint is to deliver valuable functionality. Each product increment builds on previous increments. The goal is to complete the tasks defined in the sprint backlog by the sprint's delivery date and deliver an increment of a *done* product. An increment is said *done* if it is in conformance to established acceptance criteria and, thus, it can be delivered to the client.

As a time-boxed event, the end date for a sprint does not change. The team can reduce functionality to be delivered at the end of the sprint, but the delivery date cannot change [18]. During the sprint the team holds *daily stand-up meetings* aiming at optimizing the probability of the development team meeting the sprint goal. Before delivering the increment produced during a sprint, the team performs a *sprint review meeting* to inspect the increment and adapt the product backlog if needed. At the end of the sprint, there is a *sprint retrospective meeting*, when the team evaluates and reflects about itself and the project, regarding to people, relationships, processes and tools. As a result, a plan for improvement can be created. The meetings that occur during a sprint are known as *ceremonies* [2].

2.2 Integration and Interoperability

Integration can be defined as the act to incorporate components into a complete set, conferring it some expected properties. The components are combined in a way to form a new system constituting a whole and

creating synergy [19]. *Interoperability*, in turn, is the ability of application components to exchange or share data and service [20]. Interoperability provides two or more business entities with the ability of exchanging or sharing information and of using functionality of one another in a distributed and heterogeneous environment. It preserves component systems as they are [21]. In this paper, the term integration is adopted in broader sense, covering both integration and interoperability.

For integrating applications and properly support software-related processes, it is necessary to create a coherent information system architecture in which the various software-related processes, data storages and applications are integrated so that they appear seamless from the point of view of the individual user [21].

Integration is a complex task [22]. Organizations have used an increasing number of applications to support software processes [3]. In general, these applications are based on different models, computing languages, platforms and operating systems, which leads to integration challenges.

Integration can address different layers, namely [19]: data, service and process. *Data integration* deals with moving or federating data between multiple data storages. Integration at this layer assumes bypassing the application logic and manipulating data directly in the data store (e.g., a database, through its native interface). *Message or service integration* addresses messages exchange between the integrated applications. Any tier of an application, such as GUI, application logic or database, can originate or consume the message. *Process integration* views enterprises as a set of interrelated processes and it is responsible for handling message flows, implementing rules and defining the overall process execution. It constitutes the most complex integration approach.

Integration can also consider different levels [19]. Integration at *syntactical* level addresses the way the data model and operation signatures are written down. Integration at *semantic* level, in turn, deals with the intended meaning of the concepts in a data schema or operation signature.

2.3 Ontology

Semantic integration requires us to contrast and harmonize the conceptualizations underlying applications to be integrated. In this setting, ontologies have an important role to play, as they are formal and explicit specifications of a (shared) conceptualization [23]. According to Scherp et al.[24] ontologies can be organized in a three-layered architecture that discriminates between foundational ontologies, core ontologies, and domain ontologies. *Foundational ontologies* aim at modeling the very basic and general concepts and relations that make up the world (e.g., objects, events, participation and part hood). They are generic across any area and are highly reusable in different modeling scenarios. *Core ontologies* provide a refinement to foundational ontologies by adding detailed concepts and relations in a specific area (such as service, process, organizational structure) that still spans across various domains. Finally, *domain ontologies* make the best possible description of knowledge that is specific for a particular domain in reality, such as a domain-specific medical ontology describing the anatomy of the human body. Domain ontologies can make use of or be based on foundational ontologies and core ontologies, by specializing their concepts.

Another important distinction sets apart ontologies as conceptual models, called *reference ontologies*, from ontologies as computational artifacts, called *operational ontologies* [25]. A reference ontology is constructed with the goal of making the best possible description of the domain in reality, representing a model of consensus within a community, regardless of its computational properties. Once users have agreed on a common conceptualization, operational versions (machine-readable ontologies) of a reference ontology can be implemented. Differently from reference ontologies, operational ontologies are designed with the focus on guaranteeing desirable computational properties [26]. Given the focus of this work on semantic integration, SRO is proposed as a reference ontology.

For a complex domain, representing its knowledge as a single ontology results in a large and monolithic ontology that is hard to manipulate, use, and maintain [27]. On the other hand, representing each subdomain in isolation is a costly task that leads to a very fragmented solution that is again hard to handle [17]. In such cases, building an ontology network is a suitable approach [27]. 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 [27]. Anchoring SRO in an ontology network facilitates its development through reuse and supports its integration with a number of other SE domains, as discussed in the sequel.

2.4 *Software Engineering Ontology Network – SEON*

The *Software Engineering Ontology Network*[5] (SEON) [17] is an ontology network that describes various subdomains of the Software Engineering domain. SEON organize its ontologies according to the layers defined by Scherp et al. [24] as discussed in the previous section (foundational, core and domain layers). While we omit the foundational layer in this paper for brevity, it has been instrumental for the design of SEON (we refer the reader to [17] for further details).

SRO reuses concepts from three SEON ontologies: the *Enterprise Ontology* (EO), a core ontology that addresses core aspects regarding organizations, such as organizational roles, project and teams; the *Software Process Ontology* (SPO), a core ontology that provides a conceptualization on the software process domain, addressing aspects related to processes, activities, resources, people, artifacts and procedures; and the *Reference Software Requirements Ontology* (RSRO), a domain ontology that deals with concepts related to software requirements. Figure 1 shows the fragment of these ontologies relevant to this paper as a UML class diagram. Concepts represented in yellow are belong to EO, those in green are to SPO and in pink to RSRO. In the model description, SEON concepts are underlined and examples (instances) are shown in *italics*.

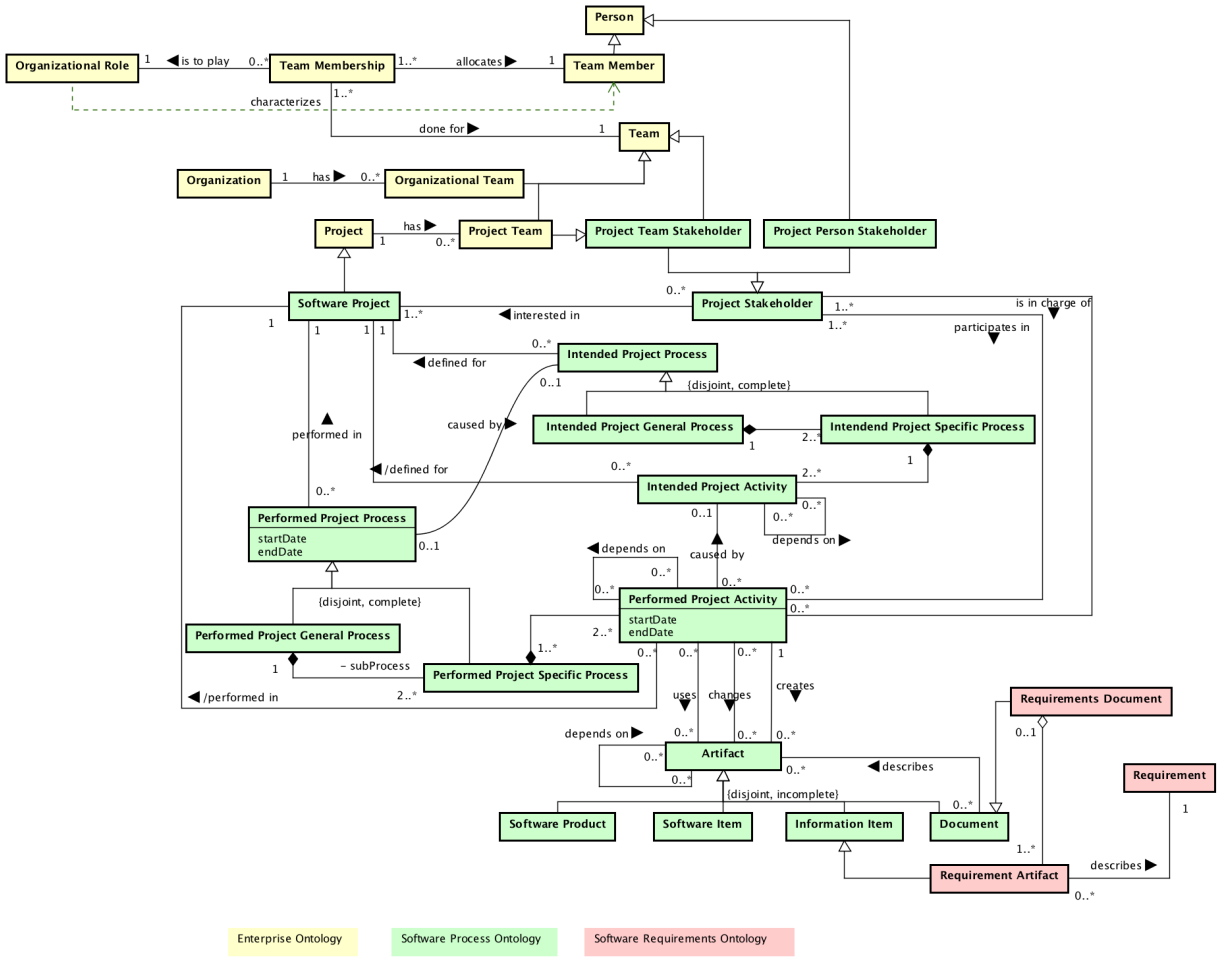

Figure 1 - SEON fragment

An Organizational Role is a social role recognized by the Organization, assigned to agents when they are hired, included in a team, allocated or participating in activities. For example, *project manager*, *designer* and *programmer* are examples of Organization Roles existing in the *Prodest* (Organization).

A Team Member is a Person that plays an Organizational Role in a particular Team. A Team can be related to a Project (Project Team), e.g., *the development team of a particular project at Prodest,* or to an Organization (Organizational Team), e.g., *the marketing team of Prodest*. The allocation of a Team member to play an

[5] SEON specification is available at *http://nemo.inf.ufes.br/en/projects/seon/*.

Organizational Role in a Team is made through the social relation Team Membership. For example, a team membership can allocate *John* as a team member to play the *programmer* organizational role in the *development team of a particular project at Prodest*.

In the software domain, a Software Project is a Project related to software development or maintenance. An agent interested in a particular Software Project is a Project Stakeholder. It can be a Project Person Stakeholder (e.g., the *project manager*) or a Project Team Stakeholder (e.g., the *project development team*).

An important distinction in SPO is between Intended and Performed Project Process. The former refers to a process intended to be performed in the project, i.e., the process planned for that project. The latter refers to the process as actually executed in the project. Therefore, an intended process is understood as an intention to execute certain types of actions; in its turn, a performed process is understood as a complex action (an "occurrence") which may not correspond to the original intention.

An Intended Project Process can be a General Intended Project Process, which refers to the whole process defined for a project, or a Specific Intended Project Process, which is defined with a specific purpose for a project. The *Project Management* and the *Requirements Engineering* processes defined for a *particular project at Prodest* are examples of Specific Intended Project Process. The whole process comprising the *Project Management*, *Requirements Engineering*, *Design*, *Implementation*, and *Test* specific processes defined to that project is an example of General Intended Project Process. A Specific Intended Project Process is composed of a set of Intended Project Activities that support the achievement of the process purpose. For example, *Requirements Elicitation* and *Requirements Documentation* could be intended activities of the *Requirements Engineering intended process*.

Analogous to Intended Project Process, a Performed Project Process can be a General Performed Project Process or a Specific Performed Project Process, which is composed of Performed Project Activities. Performed processes and activities can be caused by intended processes and activities. For example, the intended activity *Requirements Elicitation* defined to a particular project could cause the execution of the *Requirements Elicitation activity* in that project, in the sense that the intention to perform an activity can result in performing the activity.

A Project Stakeholder can participate in or be in charge of a Performed Project Activity. The relation in charge of indicates that the Project Stakeholder was responsible for performing the Performed Project Activity. On the other hand, the relation participates in means that the Project Stakeholder contributed with the execution of the Performed Project Activity. For example, in a particular project, the *system analyst* can have been in charge of *Requirements Elicitation*, while the *client* can have participated in that activity.

Performed activities create, use or change Artifacts such as Software Products, Software Items, Information Items and Documents. Software Product refers to one or more computer programs together with any accompanying auxiliary items, such as documentation, delivered under a single name and ready for use (e.g., *Eclipse IDE, MSWord*). Software Item is a piece of software considered an intermediary result of the software process (e.g., *a program*, *a script*, *a database schema*). Information Item refers to relevant information for human use in the software process context (e.g., *a bug reported*, *a documented requirement*). Document is any written or pictorial, uniquely identified, information related to the software process, usually presented in a predefined format (e.g., *a Design Specification*). *Documents* can describe other Artifacts (e.g., a *Design Specification* describes a *software architecture*). A Document composed of Requirements Artifacts that describe Requirements is said a Requirements Document (e.g., a *Requirements Specification*). Requirements are goals to be achieved, representing a condition or capacity needed for the user (e.g., *create the register of products in the web site*).

3 Methodological Approach

The work addressed in this paper follows the Design Science Research (DSR) paradigm, which concerns extending "human and organizational capabilities by creating new and innovative artifacts" [28] [29]. We used this research approach because the object of study is an artifact—specifically, a reference ontology that captures the conceptualization underlying Scrum—and its evaluation was possible in a real organizational environment. More specifically, the reference ontology is put to use to support a new semantic interoperability solution.

According to Hevner [29], Design Science Research is an iterative process including three cycles. A Design Science Research project begins with the *Relevance Cycle*, which involves defining the problem to be addressed, the requirements and the criteria for evaluating the results [29]. The problem addressed by this work involves the need for integrating applications considering semantic issues to provide useful information to support decision making in agile development context. Such problem was identified from the literature and also observed by one

of us when working as Scrum master and consultant in projects adopting Scrum. Considering the identified problem, we decided to develop a Scrum reference ontology to provide a conceptualization about Scrum in the software development context and serve as a basis to solve semantic integration problems. As requirements to develop the ontology we defined: (R1) the ontology must provide a common and comprehensive conceptualization about the domain; (R2) the ontology must be able to represent real world situations; and (R3) the ontology must be able to aid semantic integration solutions. R1 and R2 were defined because it is expected that a reference ontology satisfies these criteria [26]. R3 was established considering that we intend to use SRO to support application integration.

The *Design Cycle* involves developing and evaluating artifacts or theories to solve the identified problem [29]. Therefore, in this cycle, we developed and evaluated the *Scrum Reference Ontology* (SRO). To evaluate SRO concerning R1 and R2, we performed verification and validation activities, as suggested in [26]. As for R3, we used SRO in the development of a semantic integration solution for Azure DevOps and Clockify and applied the integrated solution in a software development project at Prodest. The project lasted 24 months and the project team was composed of 14 team members.

Finally, the *Rigor Cycle* refers to using and generating knowledge [29]. In this work, the main contribution is SRO, which provides a conceptualization about Scrum that can be used as a reference model for other semantic interoperability initiatives, as well as for communication purposes. The integration solution produced to integrate Azure DevOps and Clockify is also a contribution of this work.

4 Scrum Reference Ontology - SRO

The Scrum Reference Ontology (SRO) consolidates reference literature on the topic, using as main sources [2][30][31][32][33]. SRO is organized into five subontologies: (i) the *Scrum Process* subontology addresses the events that occur in a project that adopts Scrum, such as the Scrum ceremonies; (ii) the *Scrum Stakeholders* subontology concerns the teams, agents and roles involved in a Scrum project; (iii) the *Scrum Stakeholders Participation* subontology deals with the participation of stakeholders in the events of a Scrum project; (iv) the *Product and Sprint Backlog* subontology addresses aspects related to the requirements established in a Scrum project and activities planned to materialize them; and finally (v) the *Scrum Deliverables* subontology focuses on the results produced during a Scrum project. Figure 2 shows an overview of SRO. In the figure, each package inside the SRO package represents a subontology of SRO. Dependency relationship indicates that one (sub)ontology reuses concepts from another.

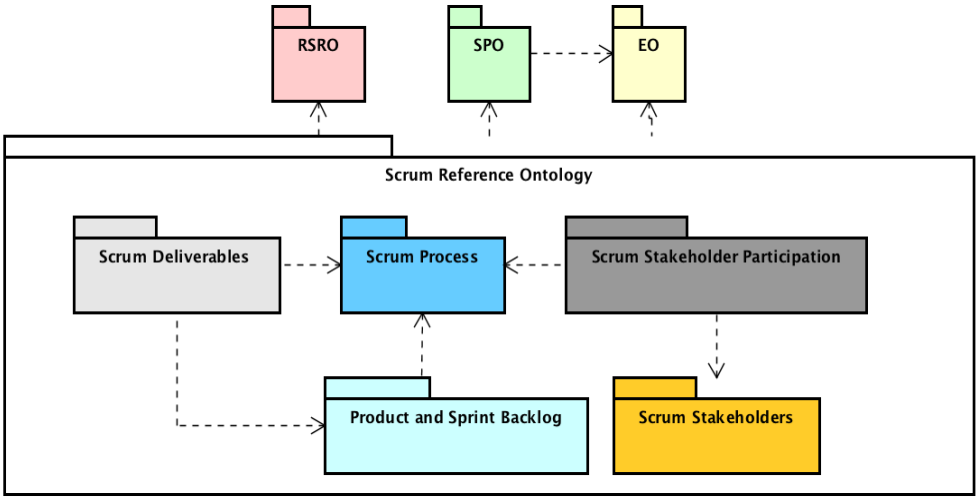

Figure 2 – SRO Architecture

SRO was developed following SABiO [26], a systematic approach that guides the development of reference ontologies. As proposed in SABiO, SRO functional requirements were established by means of competency questions, which are questions that the ontology must be able to answer and are used as a basis to build the ontology conceptual model. Next, we describe each SRO subontology. We use the following

conventions: SRO concepts are written in **bold**, SEON concepts are underlined and examples (instances) are shown in *italics*. The colors used in Figure 2 are used to indicate the (sub)ontology to which a concept belongs.

4.1 Scrum Process Subontology

This subontology aims to answer the following competency questions:

CQ01. Which processes and activities make up a Scrum process?
CQ02. In a Scrum project, on which other activities/processes did a certain activity/process depend?
CQ03. How many sprints were performed in a Scrum project?
CQ04. What ceremonies were performed in a sprint?
CQ05. What development tasks were performed in a sprint?
CQ06. When did a Scrum project start?
CQ07. When did a Scrum project end?
CQ08. When did a Scrum process start?
CQ09. When did a Scrum process end?
CQ10. When did a Scrum project activity start?
CQ11. When did a Scrum project activity end?

CQ01 and CQ02 regard process and activities involved in a project that adopts Scrum. They aim to provide knowledge about the Scrum process structure and the order in which its subprocesses and activities occur. CQ03 to CQ05 concern information about a particular project or sprint. For example, from CQ04 a Scrum master can identify if a ceremony was not performed in a sprint and thus can verify the reasons and act accordingly [35][36]. Moreover, by answering these CQs for many projects it is possible to get consolidated information, such as the average number of sprints performed in projects of a certain organization. CQ06 to CQ11 refer to temporal aspects [35][36]. They provide information about processes and activities duration and allow for project performance analysis. Figure 3 shows the Scrum Process subontology conceptual model. In the figures depicting SRO conceptual models, a dashed line separates SEON and SRO concepts.

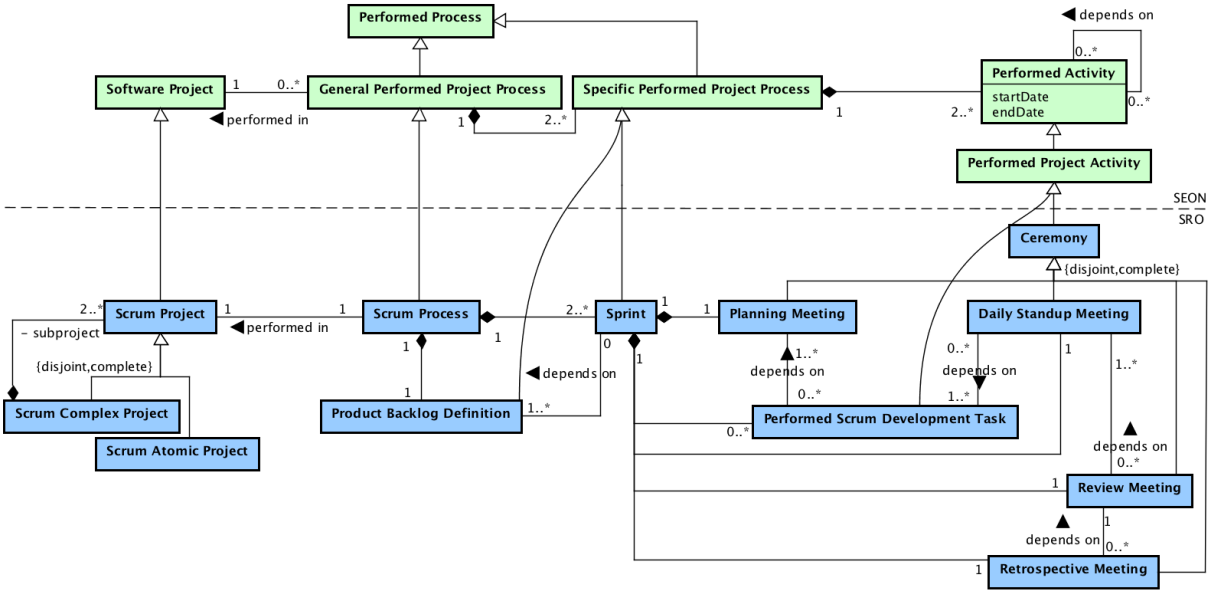

Figure 3 - Scrum Process Subontology

A **Scrum Project** is a Software Project that adopts Scrum in its process (**Scrum Process**). A **Scrum Process** is a General Performed Project Process (i.e., it is an overall process performed in a project) composed of two types of Specific Performed Project Process: **Product Backlog Definition**, which aims at defining and prioritizing the functionalities to be produced in the **Scrum Project**, and two or more **Sprints**, which occur after the **Product Backlog Definition** and aims at developing the product.

A **Sprint** is composed of two types of Performed Project Activities: **Ceremonies** and **Scrum Performed Development Tasks**. The ceremonies that compose a Sprint are **Planning Meeting**, **Daily Standup Meeting**, **Review Meeting** and **Retrospective Meeting**. Dependency relations (depends on relation between Performed

Project Activities) establish the order in which these ceremonies occur. For example, **Performed Scrum Development Task** depends on **Planning Meeting**, because a **Performed Scrum Development Task** refers to the execution of a task planned in a **Planning Meeting**. **Daily Standup Meeting**, in turn, depends on **Performed Scrum Development Task**, because a **Daily Standup Meeting** occurs after the execution of the development tasks discussed in that meeting.

4.2 Scrum Stakeholders Subontology

This subontology aims to answer the competency questions CQ12 to CQ16:

CQ12. What roles are involved in a Scrum project?
CQ13. What teams are involved in a Scrum project?
CQ14. Which roles are involved in a team in a Scrum project?
CQ15. Who are the members of a team in a Scrum project?
CQ16. Which role is played by a team member in a Scrum project?

CQ12 to QC14 provide knowledge about roles and teams involved in Scrum projects. CQ15 and CQ16, in turn, allow identifying who is allocated to a certain team and the role he/she plays in that team [33]. This information is important to identify the team members of Scrum projects and avoid overallocation to the same person. Moreover, when performing new allocations, it is possible to look at allocation historical data to verify people profile and allocate them to play roles accordingly. The conceptual model of the Scrum Stakeholders subontology is shown in Figure 4.

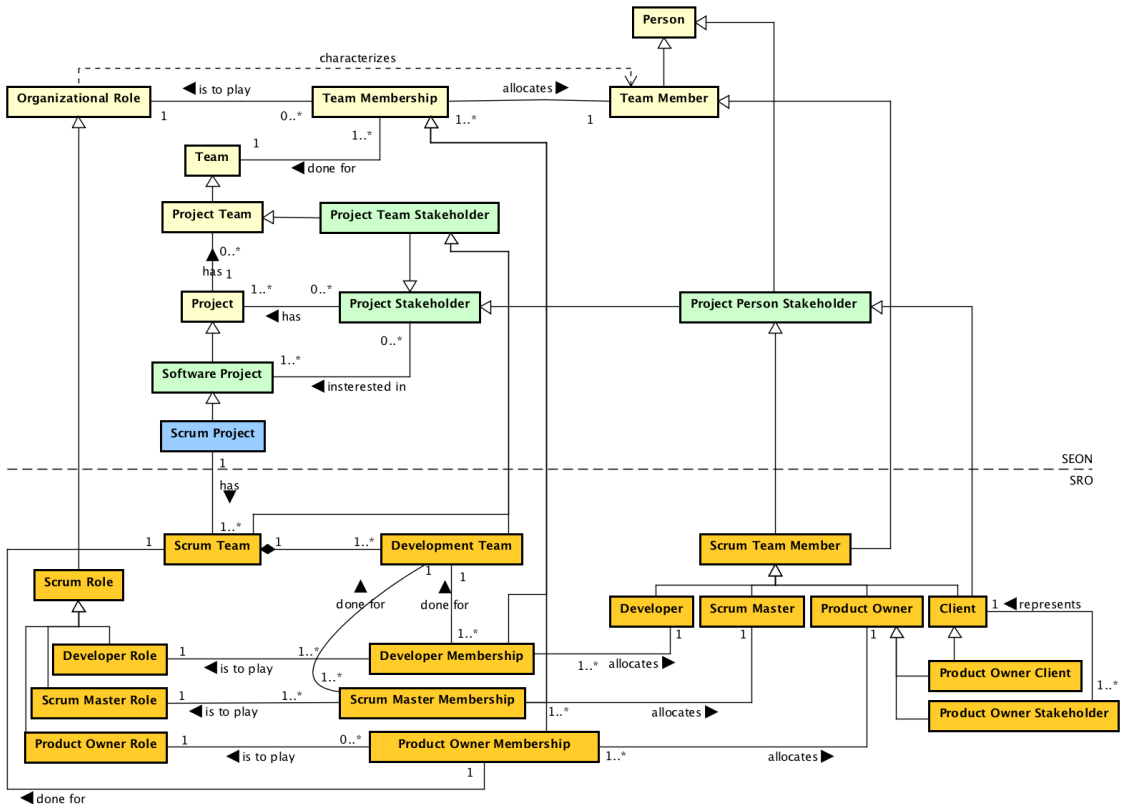

Figure 4 – Scrum Stakeholders Subontology

A **Scrum Team Member** is a Project Person Stakeholder interested in a **Scrum Project**. A **Scrum Team Member** is allocated to a **Scrum Team** (a Project Team Stakeholder interested in a **Scrum Project**) to play the Organizational Role of **Product Owner Role**, **Scrum Master Role** or **Developer Role**. As explained in the Enterprise Ontology context (see Section 2), Team Memberships allocate Team Members to Teams. Thus, **Product Owner Membership** allocates a Project Person Stakeholder to play the **Product Owner Role** in a **Scrum Team**. This Project Person Stakeholder is called **Product Owner**. For example, if *John* is allocated to

play the **Product Owner Role** in the **Scrum Team** *st* of the **Scrum Project** *sp*, it means that *John* is the **Product Owner** in *st.* Analogously, **Scrum Master Membership** and **Developer Membership** allocate, respectively, a **Scum Master** and a **Developer** to a **Development Team.** The **Development Team** is part of a **Scrum Team** and is responsible for developing the product and intermediary results.

A **Product Owner** can be a **Product Owner Client** or a **Product Owner Project Stakeholder**. The former occurs when the **Client** himself is a **Scrum Team Member** and plays the **Product Owner Role**. The latter occurs when another person represents the **Client**'s interests by playing the **Product Owner Role** in the **Scrum Project**.

4.3 Scrum Stakeholders Participation Subontology

This subontology answers questions about the participation of stakeholders in a Scrum project. It provides information about the stakeholders involved in processes and activities of a Scrum project (CQ17 to CQ22 below). This information helps analyze the participation of stakeholders in a project and verify team members performance (e.g., by identifying the development tasks a team member performed and the tasks duration)[2][32][33]. The subontology also provides knowledge about the roles involved in processes and activities of a Scrum project.

CQ17. Which stakeholders are in charge of the ceremonies of a Scrum project?
CQ18. Which stakeholders participate in the ceremonies of a Scrum project?
CQ19. Which stakeholders are in charge of development tasks of a Scrum project?
CQ20. Which stakeholders participate in of development tasks of a Scrum project?
CQ21. Which stakeholders are in charge of processes of a Scrum project?
CQ22. Which stakeholders participate in of processes of a Scrum project?

Figure 5 shows the subontology conceptual model.

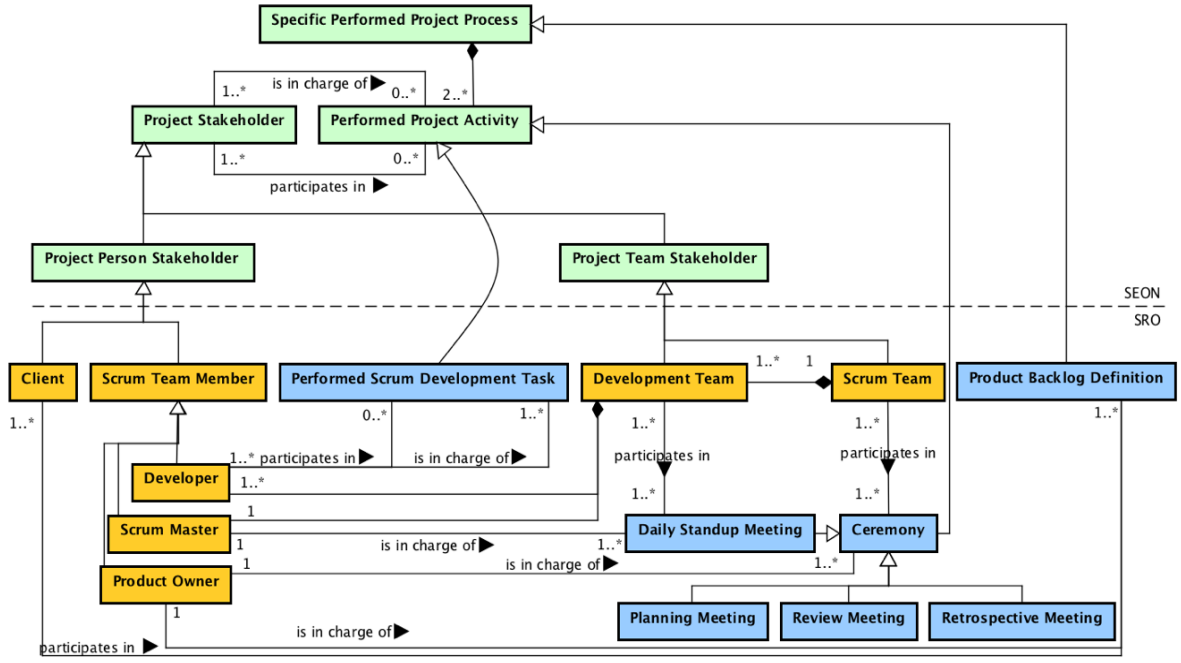

Figure 5 - Scrum Stakeholders Participation Subontology

To address the involvement of stakeholders in processes and activities of a **Scrum project**, the subontology focuses on the is in charge of and participates in relations defined between stakeholders (Project Stakeholders) and activities (Performed Project Activity) in SPO. As discussed in Section 2, the former states that one or more stakeholders are responsible for performing one or more activities. The latter establishes that stakeholders contribute to the execution of one or more activities. The equivalent relationships between stakeholders and process are derived from the relationships between stakeholders and performed activities and whole-part relationship between activities and process.

A **Product Owner** is responsible for the **Product Backlog Definition, Planning Meeting**, **Review Meeting** and **Retrospective Meeting**, while a **Scrum Master** and a **Development Team** participate in these process and ceremonies. The **Client** also participates in the **Product Backlog Definition**.

A **Scrum Master** is responsible for the **Daily Standup Meeting**, in which **Developers** participate. Finally, **Developers** are responsible for or participate in **Performed Scrum Development Tasks**.

The model depicted in Figure 6 represents the stakeholders (**Developer**, **Scrum Master**, **Product Owner**) involved in Scrum activities and subprocess. The respective Organizational Roles played by these stakeholders (**Developer Role**, **Scrum Master Role**, **Product Owner Role**) can be seen in Figure 5 and represent the roles involved in Scrum activities and subprocess (e.g., the **Scrum Team Member** in charge of a **Performed Scrum Development Task** is a **Developer**, i.e., it plays the **Developer Role**).

4.4 Product and Sprint Backlog Subontology

This subontology focuses on aspects related to the requirements established in a Scrum project. It aims to answer the following competency questions:

CQ23. What user stories were defined in the product backlog of a Scrum project?
CQ24. What is the priority of a user story in the product backlog of a Scrum project?
CQ25. How is a user story broken down into others?
CQ26. What acceptance criteria were established to a user story?
CQ27. Which user stories were selected to a sprint backlog?
CQ28. What development tasks were planned to materialize a user story?
CQ29. What development tasks were performed to materialize a user story?
CQ30. What development tasks were planned to a sprint?
CQ31. What development tasks were performed in a sprint?

CQ23 to CQ25 regard the user stories defined to a Scrum project and recorded in its product backlog. This information is necessary to know the project scope and the priority to address its requirements (i.e., the user stories). Moreover, it is possible to identify user stories decomposed into others to ease development tasks and project management [30]. CQ26 provides information about criteria that must be considered to evaluate the results produced when materializing a user story. This information is important to quality assurance. CQ28 and CQ30 provide information about tasks planning, while QC29 and QC31 concern tasks execution. By answering these questions, it is possible to track planned and performed tasks and evaluate the adherence between them. CQ27 allows identifying the user stories to be materialized during a sprint. By relating this information to information provided by CQ31 and CQ29 it is possible to verify if the user stories selected to be addressed during a sprint were materialized in that sprint. Figure 6 shows the conceptual model of the Product and Sprint Backlog subontology.

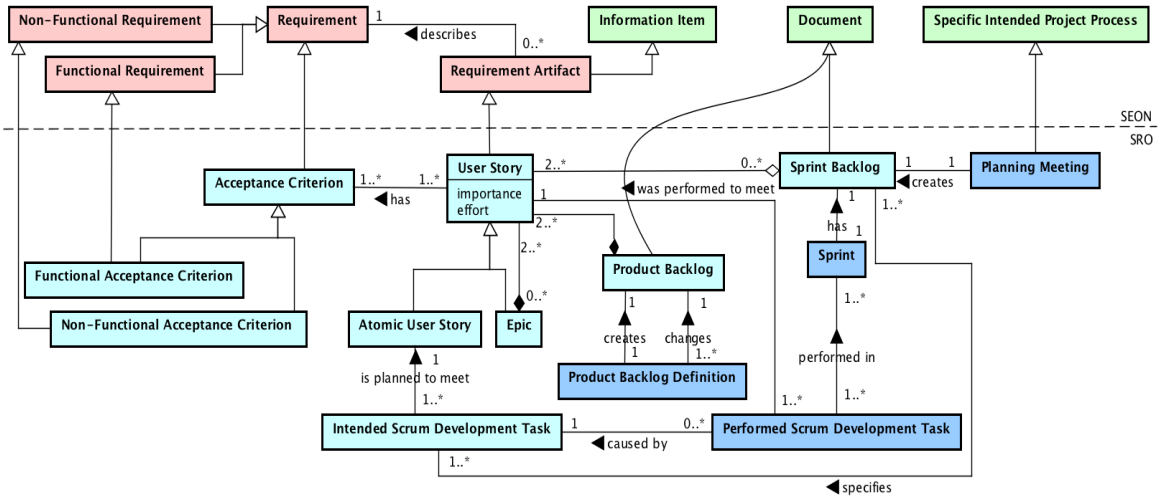

Figure 6 - Product and Sprint Backlog Subontology

The **Product Backlog** is created during the **Product Backlog Definition**. The **Product Backlog** is a Document that contains the requirements of the product to be developed in the **Scrum Project**. These requirements are described by means of **User Stories**. Therefore, a **User Story** is a Requirement Artifact that describes Requirements in a **Scrum Project**. For example, *(US1) I, as a Traveler, want to pay my travel ticket.* A **User Story** can be an **Atomic User Story**, when it is not decomposed in others (e.g., (US1.1) *I, as a Traveler, want to pay my travel ticket with my credit card;* (US1.2) *I, as a Traveler, want to pay my travel ticket with bank slip*), or an **Epic**, when it is composed of other **Use Stories** (e.g., US1, which is composed of US1.1 and US1.2).

A **User Story** has two main properties [31]: **Importance** and **Effort**. **Importance** defines how valuable for the organization the **User Story** is. Usually, the **Product Owner** set a number for it. The higher the number, more valuable is the **User Story** for the organization. **Effort** defines how difficult is for the **Development Team** to implement the **User Story**. The higher the effort, more difficult it is to materialize the **User Story**.

Each **User Story** has **Acceptance Criteria,** which are Requirements used to verify if the **User Story** was developed correctly and meets the client needs. An **Acceptance Criterion** can be a **Functional Acceptance Criterion** (i.e., a Functional Requirement used to verify if the functionality addressed in the **User Story** was developed correctly) or a **Non-Functional Acceptance Criterion** (i.e., a Non-Functional Requirement establishing a quality criterion related to product characteristics or capabilities, such as usability and portability). (AC1) *The credit card must be valid* and *(AC2) The payment authentication is done in less than 10ms* are, respectively, examples of **Functional** and **Non-Functional Acceptance Criterion** related to US1.1.

During the **Sprint Planning Meeting** of a **Sprint**, the **User Stories** to be addressed in that **Sprint** are selected from the **Product Backlog**. For each selected **User Story**, **Intended Scrum Development Tasks** are planned. They describe the tasks needed to materialize each **User Story**. The selected **User Stories** and the respective **Intended Scrum Development Tasks** are thus recorded in the **Sprint Backlog,** a Document that describes the **Sprint** planning.

When the **Sprint** planning is executed, **Intended Scrum Development Tasks** causes **Performed Scrum Development Tasks**, which are the tasks actually performed to materialize the **User Stories**. That is, the tasks planned to produce the **User Stories** lead to the execution of tasks with that purpose.

Intended Scrum Development Tasks defined in the **Sprint Backlog** but not executed in the respective **Sprint** (i.e., without a respective **Performed Scrum Development Task**) can be latter associated to the **Sprint Backlog** of next **Sprints**. Hence, an **Intended Scrum Development Task** may be related to several **Sprint Backlogs** and, consequently, to several **Sprints**.

4.5 Scrum Deliverables Subontology

This subontology aims to answer the following competency questions:

CQ32. Which types of deliverables are produced in a Scrum project?
CQ33. What deliverables were produced in a Sprint?
CQ34. Which deliverables were produced in a Scrum project?
CQ35. Which user stories did a deliverable materialize?
CQ36. Which deliverables were accepted in a Sprint?
CQ37. Which development tasks produced accepted deliverables?

CQ32 to CQ35 provide information about deliverables produced during Scrum projects. This information associated to information from CQ27 to CQ31 allows verifying project performance and progress in terms of selected user stories, tasks performed to implement the selected user stories and deliverables that materialized the user stories [2][30]. Moreover, by answering CQ31, CQ33, CQ36 and CQ37 it is possible to evaluate work quality. For example, by relating information from CQ33 and CQ36 it is possible to verify if all the deliverables produced in a sprint were accepted or if there is a need for reworking to fix not accepted deliverables [2][30]. Furthermore, from CQ31 and CQ37 it is possible to identify how much work has been spent on producing deliverables that end up "not accepted". All this information is useful to verify work quality and team productivity. The conceptual model of the Scrum Deliverables subontology is shown in Figure 7.

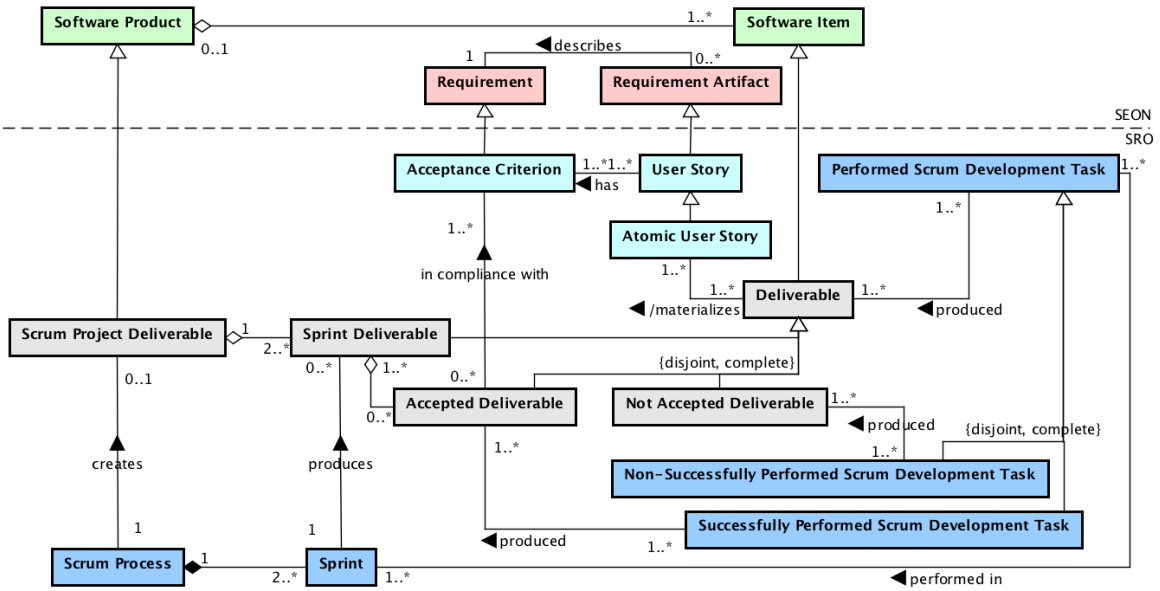

Figure 7 – Scrum Deliverables Subontology

Performed Scrum Development Tasks are performed in a **Sprint** aiming to produce **Deliverables**, which are Software Items that materialize **User Stories** addressed in that **Sprint** (e.g., a *feature to search and select flight*; a *feature to pay the travel ticket using credit card*).

A **Deliverable** is evaluated considering the **Acceptance Criteria** related to the **User Stories** it materializes. When the **Deliverable** is in conformance to the related the **Acceptance Criteria**, it is said an **Accepted Deliverable** and it means that it is "done". Otherwise, it is a **Not Accepted Deliverable**. For instance, *the feature to pay the travel ticket using credit card,* which materializes US1.1 (*I, as a Traveler, want to pay my travel with my credit card*) is an **Accepted Deliverable** if it is in conformance to both the **Acceptance Criteria** defined to US1.1: (AC1) *The credit card must be valid (AC2) The payment authentication is done in less than 10ms*. Otherwise, it is a **Not Accepted Deliverable**.

A **Performed Scrum Development Task** that produced only **Accepted Deliverables** is said a **Successfully Performed Scrum Development Task**. On the other hand, a **Non-Successfully Performed Scrum Development Task** is a **Performed Scrum Development Task** that produced one or more **Not Accepted Deliverables**. **User Stories** related to **Not Accepted Deliverables** can return to the **Product Backlog** to be addressed again in next **Sprints**. For example, suppose that the *feature to pay the travel ticket using credit card,* which materializes US1.1 (*I, as a Traveler, want to pay my travel with my credit card*), was produced by the **Performed Scrum Development Task** *Build page to credit card payment* in the **Sprint** *S1*. Moreover, consider that US1.1 was not in conformance to the **Acceptance Criterion** *(AC2) The payment authentication is done in less than 10ms.* That means that the *feature to pay the travel ticket using credit card* is a **Not Accepted Deliverable** and, thus, *Build page to credit card payment* is a **Non-Successfully Performed Scrum Development Task**. Since the **User Story** US1.1 was related to a **Not Accepted Deliverable,** after *S1* is finished, US1.1 can return to the **Product Backlog** to be addressed in the **Sprint** *S2* (or other) in order to be materialized by an **Accepted Deliverable**.

Accepted Deliverables produced in a **Sprint** are integrated forming a more complex and complete Software Item called **Sprint Deliverable**, which is the **Sprint** result delivered to the client. The set of **Sprint Deliverables** produced in the **Sprints** of a **Scrum Project** forms the **Scrum Project Deliverable**, which is the final deliverable of the **Scrum Project**. **Scrum Project Deliverable** is a Software Product and, as such, it is composed of (i) one or more Software Items (e.g., *programs*) working together for satisfying certain needs, and (ii) other items to support the Software Product use or maintenance, such as documentation.

Some constrains not captured by the SRO conceptual models were defined by means of axioms. As an example, in the context of the Product and Sprint Backlog and Scrum Deliverable subontologies (Figures 7 and 8), Scrum development tasks performed in a given sprint must be related to a user story from the sprint backlog of that sprint. That is, if a **Performed Scrum Development Task** *psdt* is performed in a **Sprint** *s* that has the

Sprint Backlog *sb* and *psdt* is to produce a **Deliverable** *d*, then there is a **User Story** *us* in the **Sprint Backlog** *sb* that is materialized by the **Deliverable** *d*. In First Order Logic:

$$\forall (psdt: Performed\ Scrum\ Development\ Task, s:Sprint, sb:Sprint\ Backlog, d:Deliverable, us:User\ Story)$$
$$isPerformedIn(psdt, s) \wedge has(s, sb) \wedge isToProduce(psdt, d) \rightarrow$$
$$\exists us\ (isPartOf(us,sb) \wedge materializes(d, us))$$

Another example, focusing on the Product & Sprint Backlog subontology (Figure 7), constrains that a task performed to meet a user story is caused by a task planned to do that. That is, if a **Performed Scrum Development Task** *psdt* was performed to meet a **User Story** *us* from the **Sprint Backlog** *sb*, then *psdt* was caused by an **Intended Scrum Development Task** *isdt* specified in *sb* and planned to meet *us*. In First Order Logic:

$$\forall (psdt: Performed\ Scrum\ Development\ Task, us:User\ Story, sb:Sprint\ Backlog, isdt: Intended\ Scrum\ Development\ Task)\ wasPerformedtoMeet(psdt, us) \wedge isPartOf(us,sb) \rightarrow$$
$$\exists isdt\ (specifies(sb,isdt) \wedge isPlannedToMeet(isdt, us) \wedge causedBy(psdt, isdt))$$

4.6 SRO Evaluation

For evaluating SRO, we performed Verification and Validation activities, as suggested in [26]. We used two approaches to ontology evaluation: assessment by human and data-driven approach [34]. In the first, we performed a verification activity by means of expert judgment, in which we checked whether the concepts, relations and axioms defined in SRO are able to answer the competency questions. For each competency question, we identified the elements of SRO which together are able to address the question. Table 1 presents (a fragment of the) results produced during verification. In the second, we aimed to validate the ontology by assessing whether it is suitable for representing real-world situations. For that, we instantiated SRO using data extracted from a real project. Table 2 presents (a fragment of the) instantiations recorded during validation. The instances were extracted from a project developed at Prodest.

Table 1 – SRO Verification (fragment)

Competency Questions	SRO **Concepts**, properties, and *relationships*
CQ04 - What ceremonies were performed in a sprint?	**Planning Meeting**, **Daily Meeting**, **Review Meeting**, **Retrospective Meeting** *subtype of* **Ceremony** **Sprint** *composed by* **Planning Meeting** **Sprint** *composed by* **Daily Meeting** **Sprint** *composed by* **Review Meeting** **Sprint** *composed by* **Retrospective Meeting**
CQ10 - When did a Scrum project activity start?	**Performed Activity**.startDate **Performed Project Activity** *subtype of* **Performed Activity** **Ceremony** *subtype of* **Performed Project Activity** **Performed Scrum Development Task** *subtype of* **Performed Project Activity**
CQ30 - What development tasks were planned to a sprint?	**Sprint** *has* **Sprint Backlog** **Sprint Backlog** *specifies* **Intended Scrum Development Task**
CQ33 - What deliverables were produced in a sprint?	**Performed Scrum Development Task** *performed in* **Sprint** **Performed Scrum Development Task** *produced* **Deliverable** **Successfully Performed Scrum Development Task** *produced* **Accepted Deliverable** **Non-Successfully Performed Scrum Development Task** *produced* **Not Accepted Deliverable**

Table 2 – SRO Validation (fragment)

SRO Concept	Instance
Scrum Project	ESPM Project
Scrum Process	Scrum process defined to the ESPM Project, comprising Product Backlog Definition, Sprints, Ceremonies and Performed Scrum Development Tasks
Product Backlog Definition	Process performed at the beginning of the project (from 01/02/2018 to 11/01/2018) to define the Product Backlog
Sprint	Sprint S40, performed from 10/21/2019 to 12/02/2019
Ceremony/Planning Meeting	Planning meeting performed at the first day of S40, on 10/21/2019
Ceremony/Daily Standup Meeting	First daily meeting performed in S40, on 10/22/2019
Ceremony/Review Meeting	Review meeting performed in S40, on 12/02/2019
Ceremony/Retrospective Meeting	Retrospective meeting performed in S40, on 12/02/2019
Scrum Team Member/Developer	Barney S. and Ted M.[6]
Scrum Team Member/Scrum Master	Marshall E.
Scrum Team Member/Product Owner/Product Owner Stakeholder	Robin S.
Scrum Team Member/Client	Robin S.
Scrum Team	The team composed of the Scrum Team Members cited above.
Developer Membership	Allocation of Barney S. to play the Developer role in the Scrum team.
Scrum Master Membership	Allocation of the Marshall E. to play the Scrum Master role in the Scrum team.
Product Owner Membership	Allocation of the Robin S. to play the Product Owner role in the Scrum team.
User Story/Epic	US65: "I, as a public servant, I want to visualize my payslips"
Atomic User Story	US65.1: "I, as a public servant, I want to visualize my payslips in an application to smartphone". US35.2: "I, as a public servant, I want to visualize my payslip in a web browser".
Product Backlog	Product backlog containing the user stories defined to the ESPM Project. It contains US65, US65.1, US65.2 among others.
Sprint Backlog	Sprint backlog created during the planning meeting of sprint S40. It contains US65, US65.1, US65.2, among others.
Acceptance Criterion/ Functional Acceptance Criterion	AC1 (related to US65.1): The payslip to be shown is determined by the month and year informed by the public servant.
Acceptance Criterion/ Non-Functional Acceptance Criterion	AC2 (related to US65.1): A payslip should appear in 500 milliseconds.
Intended Scrum Development Task	**Task** "Create payslip report functionality in mobile app" planned to implement the user story US65.1 during S40.
Performed Scrum Development Task/Successfully Performed Scrum Development Task	**Task "**Create payslip report functionality in the mobile app", performed during S40 to implement the user story US65.1 and that produced the accepted deliverable below.
Deliverable/Accepted Deliverable	Functionality "Payslip report" in the mobile app, resulting from the implementation of US65.1.
Performed Scrum Development Task/Non-Successfully Performed Scrum Development Task	**Task "**Create payslip report in the web application", performed during S40 to implement the user story US65.2 and that produced the not-accepted deliverable below.
Deliverable/Not Accepted Deliverable	Functionality "Payslip report" in the web application, resulting from the implementation of US65.2, and that was not accepted due to failure.
Sprint Deliverable	The set of accepted functionalities developed during the S40 (which includes "Paylist report" in the mobile app, among others), incorporated to the software version delivered in S39.
Scrum Project Deliverable	SmartCity, the software product resulting from ESPM project.

[6] The names of Individual project participants were omitted in favor of initials of privacy.

5 Using SRO to Support Application Integration

Here, we report on the application of SRO at Prodest's software development unit. The unit consists of around 50 employees, organized in various teams. According to their purposes, teams adopt different methods, tools, processes and development approaches. Software managers involved in projects adopting Scrum reported the need to obtain integrated data to better monitor software projects, time-tracking and product quality. According to them, data was scattered in different applications and was manually extracted in a repetitive, error-prone process.

Projects employing Scrum were supported by Azure DevOps and Clockify which lack off-the-shelf integration. For example, to obtain information about the amount of hours spent on development tasks in a given sprint, managers used to perform the following procedure: (i) export a spreadsheet from each application with data about the project, (ii) select from Azure DevOps spreadsheet tasks related to the desired sprint; (iii) retrieve from Clockify spreadsheet data corresponding to the selected tasks; (iv) for each task, sum hours of all its time entries recorded in Clockify spreadsheet; (v) record data resulting from (ii) and (iv) in a new spreadsheet and sum time spent on all the tasks. This process demands effort and has to be repeated every time integrated data is needed. If any mistake is made over the process, resulting information can be incorrect and lead to poor decision support. Moreover, manual consistency management was required in the two applications. For example, when a development task was created in Azure DevOps, a team member had to create the same task in Clockify, where task execution is controlled in detail. When the task execution was concluded, the member had to change its status from "In Progress" to "Done" in Azure DevOps and Clockify manually. These manual activities naturally create opportunities for data accuracy problems. Ideally, when a task is concluded in Clockify, the number of hours recorded in all its time entries should be summed, automatically recorded in Azure DevOps and the task status should be changed to "Done".

We followed the process illustrated in Figure 8, which is constituted of two macro-activities: *Conceptual Integration* and *Integration Design and Implementation*. SRO is used to assign semantics at the conceptual level. Thus, we show Conceptual Integration in detail.

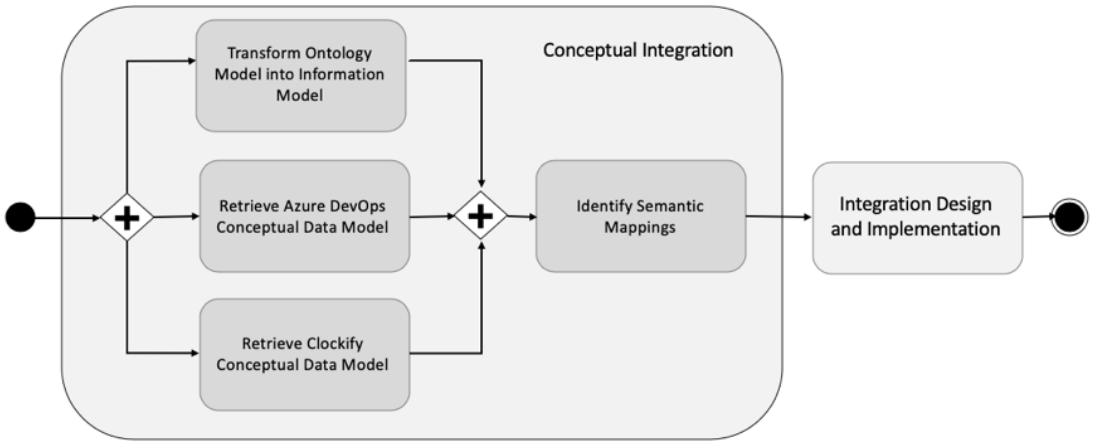

Figure 8 – Followed process to develop the proposed architecture

The first activity, *Conceptual Integration,* uses SRO as a basis to identify semantic mappings that will serve as a basis for data integration. For that, it is necessary to *Transform Ontology Model into Information Model*. An information model concerns what kind of information may be stored and exchanged considering demands of specific agents (the "recorded world"), while an ontology model concerns metaphysical aspects of a domain (i.e., it concerns what is considered to exist in the "real world") [35]. Thus, by turning the ontological model into an information model, the resulting model preserves the conceptualization in a structure more suitable for computing demands. Figure 9 shows a fragment of SRO information model, obtained by applying the information transformations to the source ontology model as proposed in [35] (attributes omitted for brevity). All classes in the SRO information model specialize a general abstract class "Entity". The generalizations were omitted.

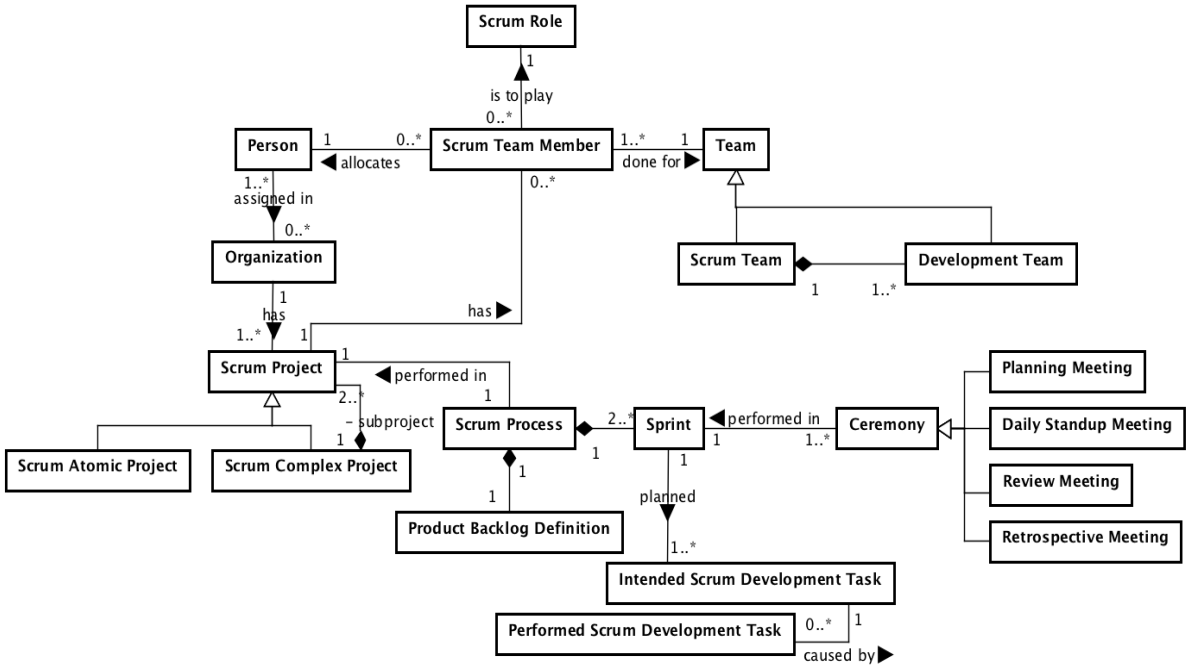

Figure 9 – Fragment of SRO information model

Besides the SRO information model, to produce the integration model, it is necessary to *Retrieve the Conceptual Data Model* of each application. By analyzing Azure DevOps, Clockify and their documentation, we retrieved their conceptual data models. Figure 10(a) shows a fragment of Azure DevOps conceptual model and Figure 10(b) presents a fragment of the Clockify conceptual data model. In the text, concepts from the applications conceptual models are shown in ***bold italics***.

In Figure 10(a), a ***Project*** has one or more ***Teams***, which are composed of one or more ***Team Members***. Each ***Team Member*** is a person that has an ***Identity*** inside Azure DevOps. A ***Project*** has ***Interactions*** that implement ***WorkItems***. A ***WorkItem*** has one or more ***WorkItem Status*****,** which can be "new" (when it is created), "open" (when it is selected in the interaction), "in progress" (when it is being performed), "closed" (when it finished) or "resolved" (when a bug is fixed). A ***WorkItem*** is if a ***WorkItem Type***, which can be a "Task", a "User Story", an "Epic" or a "Bug". In a ***WorkItem*** life cycle, a ***Team Member*** creates it, several ***Team Members*** can be assigned to implement it, and a ***Team Member*** closes it. Finally, a ***WorkItem*** can be related to other ***WorkItems***. This way, it is possible, for example, to link a ***WorkItem*** of type **"**User Story" with another of type "Task", indicating that the latter implements the former. A ***Product Backlog*** is composed of ***WorkItems*** that are implemented by a ***Team***.

In Figure 10(b), a ***Workspace*** is an entity that groups ***Projects*** related to a ***Client***. A ***Project*** has one or more ***Tasks*** performed by ***Users*** connected to the ***Project*** through ***Membership***. When performing a ***Task***, a ***User*** record ***Time*** **Entries**, representing the time spent on the ***Task***.

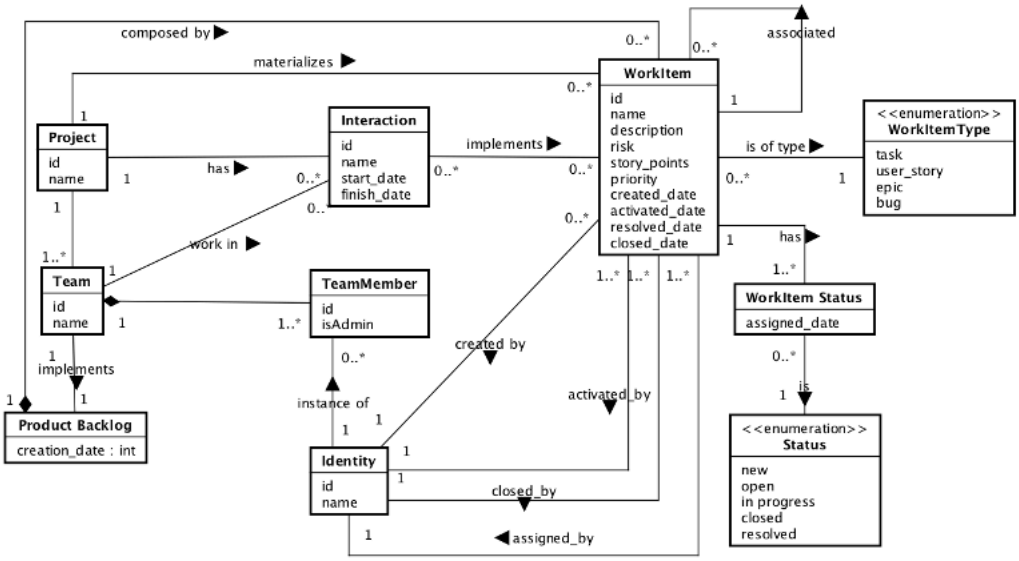

Figure 10 (a). Azure DevOps

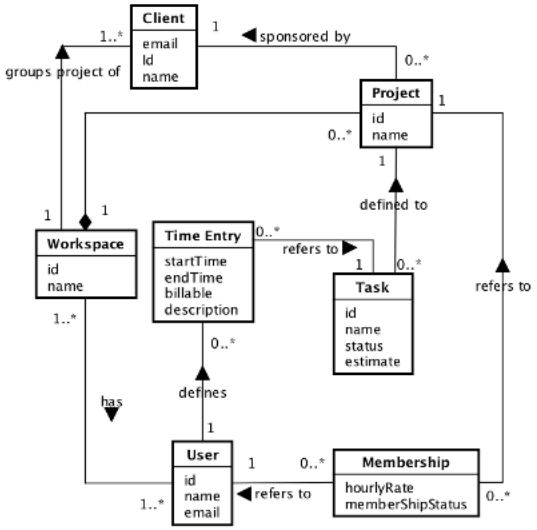

Figure 10 (b). Clockify

Figure 10 – Fragment of the applications conceptual data model.

Once we have obtained the applications' conceptual data models and the ontology information model, we needed to *Identify Semantic Mappings*. We used the SRO information model to establish the mappings, assigning semantics to application elements by relating them to SRO elements. Table 3 presents examples of mappings. Some of them directly map an application element to an SRO element. For example, ***Team*** from Azure DevOps is directly mapped to **Scrum Team** from SRO (row 3 in Table 3), since both have the same meaning. Others involve more elements or rules. For example, ***Project*** from Azure DevOps is mapped to **Atomic Scrum Project** from SRO if the ***Project*** has only one ***Team***, and it is mapped to **Complex Scrum Project** from SRO if the ***Project*** has many ***Teams*** (rows 7 and 8 in Table 3), because in Azure DevOps it is not possible to create projects inside other projects. To solve this limitation, the application allows creating a project with multiple teams, where each team refers to a subproject with its own ***Product Backlog***.

Table 3 – Examples of mappings among concepts.

#	SRO Information Model	Microsoft DevOps	Clockify
1	Person	Identity	User
2	Scrum Team	Team	-
3	Development Team	Team	-
4	Team Member	Identity and Team Member	User, when there is Membership between User and Project
5	Developer	Identity and Team Member when admin is false	-
6	Scrum Master	Identity and Team Member when admin is true	-
7	Scrum Complex Project	Project, when it has many Teams	Workspace, when it has many Projects
8	Scrum Atomic Project	Project, when it has one Team	Workspace, when it has one Project
9	Atomic User Story	Work Item, when WorkItem Type is "User Story"	-

The semantic mappings are important to implement the integration rules to enable services integration. The semantic mappings use SRO as a bridge between the applications and identify which elements of the different applications are equivalent according to SRO conceptualization. By doing that, it is possible to know which data can be integrated and how they must be stored in the SRO Database[7]. Moreover, the mappings support keeping traceability between data stored in the SRO Database and in the applications.

Once we have assigned semantics to applications' elements by means of semantic mappings, we performed *Integration Design and Implementation*. In this activity, we developed software artifacts (database, code libraries, services and dashboard) and combined them into integration processes that coordinate data integration in our architecture. We used SRO information model to build a relational database, using Python[8] and the ORM framework SQLAlchemy[9]. Each entity from the SRO information model was mapped to a class and we implemented CRUD+L (Create, Read, Update, Delete and List) operations for each class. After that, SQLAlchemy created the SRO Database automatically.

To address applications access and data traceability, we included in SRO Database some concepts in addition to the ones from the SRO information model. Figure 11 shows the added concepts (in grey). ***Application*** refers to the applications being integrated (i.e., Azure DevOps and Clockify). In order to access an application, an organization provides a user secret key and URL to access an application. This ***Configuration*** is necessary to allow the architecture services to access the data of applications. ***Application Reference*** provides an identifier for each entity stored in an application database (e.g., the ID of a task saved in Clockify, the ID of the same task saved in Azure DevOps). The identifiers are necessary to enable data integration and traceability. This way, in SRO Database, it is possible to relate each entity stored in SRO to its corresponding in Clockify and Azure DevOps.

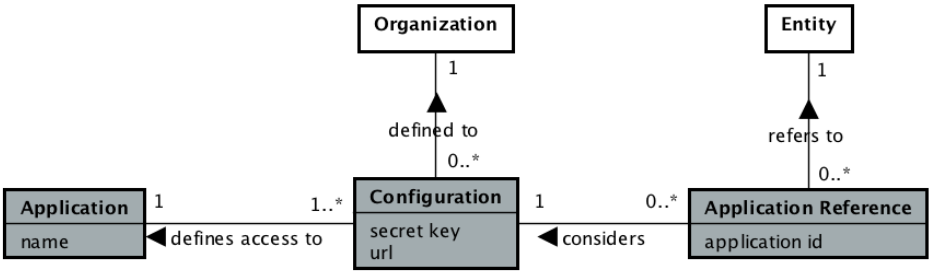

Figure 11 – Concepts added to SRO Database to allow data traceability.

[7] SRO Database is a database built based on SRO and that integrates data from the applications.
[8] *www.python.org*
[9] *www.sqlalchemy.org*

After creating SRO database, we implemented libraries containing (i) functions to enable to receive and send data to applications (e.g., *tfsx*[10] and *clockify*[11]), which receive and send data to Azure DevOps and Clockify; (ii) functions that address the semantic mappings identified in the first activity (e.g., *devops-microsoft-mapping-SRO*[12] and *clockify-mapping-SRO* implement rules addressing the semantic mappings, respectively, between Azure DevOps and SRO and between Clockify and SRO). The libraries produced in this work can be reused by anyone from open-source community.

We also created the income webservice (SRO Online webservice) to receive data from Azure DevOps when data is created or changed in that application. Once SRO database, libraries and income webservice were created, we created integration services responsible for implementing design decisions that enable exchanging and sharing information among the applications and SRO Database. Thus, we organized the integration services in integration processes to handle applications integration.

Finally, we created a dashboard to provide data to support decision making. The dashboard was built using Superset[13] and shows, among others: data regarding some agile metrics (e.g., Work in Progress (WIP) and Lead Time), key process indicators (KPIs) (e.g., number of projects and average amount of hours spent on each project), users stories where story points are missing, total of hours spent on tasks, total of deliverables considered "done. Figure 12 shows a fragment of the dashboard.

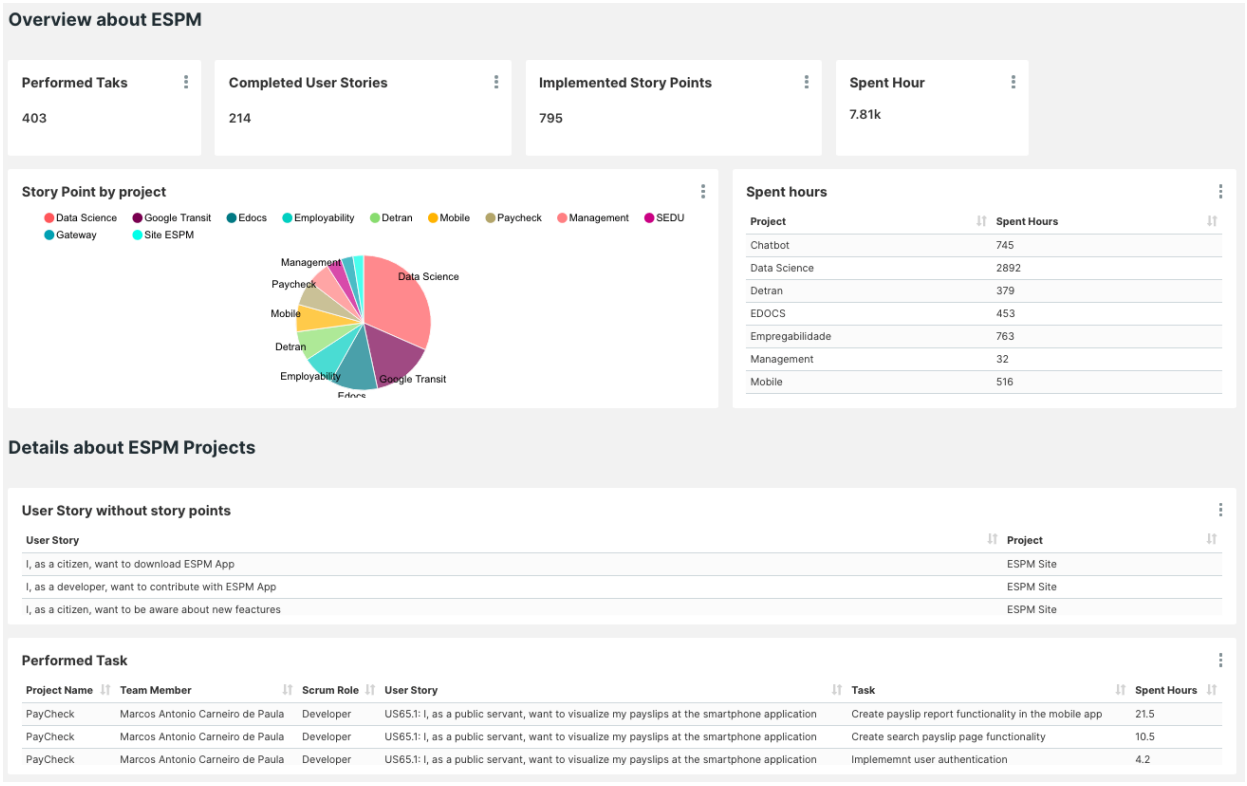

Figure 12 – Fragment of the dashboard showing integrated data.

5.1 Evaluation

Using SRO to address semantics in our integrated solution served as a proof of concept, showing that the use of SRO with integration purposes is feasible and that SRO is useful in this context. SRO helped us properly understand the conceptualization behind the applications and integrate them. Moreover, by using SRO we built

[10] *https://pypi.org/project/tfsx*
[11] *https://pypi.org/project/clockify*
[12] *https://pypi.org/project/devops-microsoft-mapping-sro/*
[13] *https://superset.incubator.apache.org*

SRO Database, which structures knowledge about projects adopting Scrum and can be used to integrate other applications supporting this domain.

To ensure that the integrated solution achieves its purpose, we used it in a real software project. Our goal was to evaluate whether an integrated solution that uses SRO conceptualization to address semantics is able to properly support activities in a Scrum project and provide data useful for decision making. Aligned with this goal, we defined the following research question: *Does the use of the integrated solution produced using SRO improve software development work and decision making?*

The project considered in the evaluation is called ESPM: Smart City[14] and it aims at providing smart public services (e.g., bus schedules and free courses) for the citizens of the state of the Espírito Santo, Brazil. The services are accessed through a mobile application. The project was developed between 2018 and 2020, counted with 14 team members (2 Scrum masters and 12 developers) and used Scrum as software process. The project was organized in subprojects (the ones shown in Figure 12, e.g., PayCheck, Edocs) and the Scrum teams had between 2 and 4 developers. Each sprint lasted two weeks. Each Scrum team had one Scrum master and one product owner. Each Scrum master was responsible for more than one team. The first author of this paper acted as Scrum master in the project from January 2018 to December 2019. He had then 6 years of experience with agile development and Scrum. The other Scrum master, in turn, had 2 years of experience with agile development and Scrum. The developers were junior interns with little experience with agile development.

From August 2019 to December 2019, the first author (playing the Scrum master role) and the other Scrum master used the integrated solution to facilitate project management. During this period, they accessed the dashboard and shared information with the team in biweekly meetings. After that, the first author performed semi-structured interviews with the other Scrum master and 9 developers to answer the research question. The developers were interviewed together, and the interview lasted about 60 minutes. The interview with the Scrum master lasted about 30 minutes. The interviews were recorded, transcribed and validated with each participant. The researcher started the interviews by asking the research question and the participants were free to present their perceptions about the use of the integrated solution. Based on the participants answers, the researcher presented new questions to investigate aspects mentioned by the interviewees. For example, when a participant said that "*it was nice to know my WIP*", the researcher asked him to explain how that information was useful. Next, we present the main results obtained from the first author (as Scrum master) and other team members perceptions.

Based on information provided in the dashboard, it was possible to make decisions along the project and also to create more realistic plans to new projects. For example, decisions about team member allocation and duration estimates became more accurate. Team member allocation was performed considering information about previous allocations provided in the dashboard. Therefore, to allocate team members to tasks, it was possible to look for team members that have worked on projects or tasks with similar characteristics (considering functional and non-functional requirements). Effort and duration estimates were performed based on information about the members productivity. By using historical data available at the dashboard to support estimates, an average deviation of 7.3% in estimates was achieved, which is smaller than the average deviation (24%) in projects that started before August/2019. During the interview, a developer reported that the first sprints of the projects used to have inaccurate time planning and pointed out that the dashboard aided to better estimate duration of each task. Another developer said that data provided in the dashboard allowed to monitor the team's velocity, contributing to observe the team evolution and consider its velocity to plan the sprints more realistically.

It was also noted that using the integrated solution the development team truly understood the meaning of "Done" in a Scrum project, since only deliverables in conformance to all acceptance criteria appeared in the "Done" section of the dashboard. This operationalization of the notion in the integrated tool helped the development team to understand that it is necessary to deliver valuable and accepted software artifacts in each sprint, contributing to product quality and team performance as well as improved project planning. In this context, a developer reported that by properly understanding the meaning of "Done" and knowing the individual and team's WIP, the team members were encouraged and motived to increase their performance and produce better deliverables.

[14] *ESPM Smart City websites: espm.es.gov.br and developers.es.gov.br*

The integrated solution also enabled to identify problems in the execution of the Scrum process in the project. For example, the dashboard provides information about user stories without story points (i.e., estimated effort to implement the user story) defined to them. This may lead the Scrum master to ask the team to complete the information. In the ESPM project, when asking some team members to set the story points, it was realized that, being novices in Scrum, they did not know how to properly estimate using story points. Thus, training in this matter was provided to the team and the problem was solved.

The Scrum master highlighted that the integrated solution helped to address problems earlier (as in the example cited above) and aided the team to change its practices and work processes "on the fly", according to the information provided in the dashboard, promoting self-organization. Moreover, she said that information provided in the dashboard helped her make decisions together with developers, which increased their engagement. Another benefit from using the integrated solution was the automatic synchronization of data between the applications, which decreased manual work, avoided errors due to manual manipulation of the same data in different applications and contributed to the team focus their attention on development activities.

As limitations, developers reported that they depended on the Scrum masters to get information provided in the dashboard, because, due to technical constraints, only the Scrum masters had direct access to the dashboard.

In sum, the results showed us that using SRO helped to address semantics and produce an integrated solution that aided improving software development work and supported decision making. The evaluation has some limitations that should be considered. We highlight the participation of the first author, who acted as Scrum master and, hence, cannot provide an unbiased external view. Moreover, since he is familiar with the SRO conceptualization and the integrated solution, this knowledge may have influenced results. We should also take the developers profile into consideration. Since they were novices in Scrum, their relatively little insight into Scrum practices may have contributed to little criticism. Another limitation is the fact that we used SRO to produce a single integrated solution, involving two applications and we used the solution in only one project. We intend to use SRO to integrate other applications and carry out similar studies in further projects to increase confidence in the generality of the obtained results.

6 Related Work

We discuss here some related efforts, both with respect to ontologies and metamodels that cover similar ground and with respect to integration solutions.

Regarding related ontologies, the studies most closely related to our work were conducted by Parsons [14], Kiv et al. [15] and Lin et al. [36]. Parsons presented a general ontology on agile methods to propose an analytical framework to understand how an overarching agile methodology is constructed. Lin et al., in turn, introduced a Scrum ontology based on concepts from CRIO metamodels [37], modelled using OWL. Kiv et al. proposed an agile method ontology modelled using UML and implemented with OWL to represent knowledge about projects. Differently from SRO, the works by Parsons [14] and Kiv et al. [15] propose general ontologies about agile paradigm, describing methods and goals without a focus on Scrum. Lin et al. [36], in turn, propose a lightweight ontology, which provides a limited conceptualization. Moreover, these ontologies are not connected to other aspects of Software Engineering. SRO describes the conceptualization about Scrum in the Software Engineering context. Thus, the SRO concepts are related to concepts from other Software Engineering sub-domains such as Requirements and Software Process.

Due to the strong connection between the Scrum process and other Software Engineering aspects, SRO was developed as a networked ontology of SEON [17]. This modeling decision allowed us to reuse concepts from other SEON ontologies and achieve a broad understanding about the Scrum process in the Software Engineering context. For example, by connecting SRO to other SEON ontologies it is possible to understand that User Story is a Requirements Artifact and, as such, describes requirements of stakeholders of the project. It is also possible to understand that a Sprint is a process composed of activities performed during a time-box. Understanding the Scrum process in the context of the Software Engineering domain contributes to a better understanding about the conceptualization and to make comparison or integration of information regarding different paradigms. For example, by acknowledging that User Story is a Requirement Artifact, when looking at information about projects developed using different process models, one can understand that, in a project that adopts the Scrum process, the User Story plays the same role than the Requirement Description plays in a project that adopts the Waterfall

process, since both are types of Requirements Artifacts in SEON. This broad analysis is not possible in any of the cited works.

Another difference of SRO when compared to the aforementioned works, is that SRO provides a more precise conceptualization. For example, SRO establishes the meaning of "Done" and what are the impacts of this concept in different aspects of a software process based on Scrum. Finally, being a networked ontology of SEON, SRO is ultimately grounded in UFO (the Unified Foundational Ontology) [38] which results in a well-founded conceptualization that can better represent real-world situations.

In addition to ontologies covering similar ground, the Method Engineering field has produced some metamodels concerned with agile methods. This is the case of Damiani et al. [39], who present Scrum metamodels using MOF[15] (Meta-Object Facility), and Ayed et al. [40], who introduce an approach to model an agile process according to an organization's characteristics, based on Situational Method Engineering (SME) [41] and using SPEM[16] (Systems Process Engineering Metamodel). Different from SRO, the metamodel proposed in [39] focuses only on few concepts related to the ceremonies and backlog, resulting in a limited view of Scrum. In [40], the metamodel concerns agile development in general, and as such, does not address specific aspects of Scrum or other particular agile methods. Because of this, the proposed metamodel is defined at a rather abstract level, hindering its use as semantic grounding for operational data that is handled by the various tools. Moreover, the purpose of the proposed models is not to provide a comprehensive conceptualization able to address semantic issues. Instead, they are intended to support process/method definition and evolution.

Concerning the architecture to application integration proposed in this work, there are in the literature some approaches that, like ours, use ontologies to support semantic integration. For example, in the work reported by Izza [19], operational ontologies are used as solutions for semantically describing, discovering and composing webservices. Different from our work, these work uses operational ontologies and, thus, deals with semantic integration at operational level. In [6], reference ontologies are used as an interlingua to integrate applications at conceptual level. Similarly, in our architecture, the ontology serves as a bridge to connect elements from different applications, assigning semantics to them at the conceptual level. Different from the cited works, in our architecture the ontology is transformed into an information model and it is used as information resource to build software artifacts (database, libraries and services) used to integrate the applications. Moreover, the aforementioned works induce design decisions that guide the developer to build a peer-to-peer connector between applications, while our architecture uses SRO database and integration services to connect the applications.

Furthermore, in our work the ontology model has an important role in the integration architecture. For example, the SRO Database is built systematically from SRO and structures knowledge about projects adopting Scrum. The knowledge is independent of technology and can be used regardless the applications being integrated. Therefore, the organization can change the project management and time-tracking applications without losing data recorded in SRO database. If new applications are used, new semantic mappings can be established, the applications can be integrated by the architecture and new data can be integrated to data previously recorded in SRO Database.

In the context of providing data to support decision making in software development, we can consider some Mining Software Repository (MSR) works as related to ours. MSR aims to analyze and cross-link data present in software repositories (e.g., source control, bug repositories, deployment logs, source code repositories and emails) to uncover actionable information about software and projects [42]. It seeks to transform static record-keeping software repositories into active repositories that could provide information to support decision making in software development [42]. For example, Mattila et. al [43] use data from Jira[17] to guide decisions to decrease deviations between planned and executed process. Malik and Hassan [44], in turn, analyze source code from code repositories to identify and propagate changes when a software artifact is modified in a project. Destefanis et. al [45] explore data from Jira to investigate how social aspects (e.g., being polite) influence developers' productivity on agile software projects. Cubranic et. al [46] and Kim et. al [47] discuss that linking data from different and heterogeneous software repositories (e.g., email, source control repository and chat) could improve data quality and, thus, provide a more complete view of a project.

[15] https://www.omg.org/mof/
[16] https://www.omg.org/spec/SPEM/2.0/Beta1/About-SPEM/
[17] Jira – www.jira.com

Similar to our work, the aforementioned works aim to use and integrate existing data to provide useful information to support decision making. However, differently from our work, the authors were not concerned with semantic aspects explicitly. As we previously discussed, neglecting semantic aspects can lead to conflicts whenever the same information item is given divergent interpretations [11]. Our work proposes the use of a reference ontology (SRO) to assign semantics to applications' information items and structure the repository of the integration solution. In this way, the created repository (SRO database) can be used to integrate not only Microsoft DevOps and Clockify, but other applications addressing similar scope. Thus, once semantics is assigned to applications' information items, it is possible to change a software repository for another (e.g., from Microsoft DevOps to Jira). The cited works, in turn, provide solutions considering the data structure of the used repositories, which makes it difficult to reuse them with different repositories. By using SRO, our work not only supports the integration solution as also helps understand the domain of interest. We argue that ontology-based approaches such as the one employed in this paper could contribute to MSR solutions by providing comprehensive and well-founded conceptual models to combine and interpret data extracted from software repositories as well as support the linking of data from different repositories.

Finally, there have been some works addressing Software Process Improvement (SPI) that also propose integrated data to support decision making and, as such, are also related to ours. For example, Renault et. al [48] use ontologies to integrate MantisBT[18] and Subversion[19], which are applications used to support Issue Management and Software Configuration Management processes, respectively. As in our work, in [48] ontologies are used to assign semantics at the conceptual level. However, differently from our work, the focus is on integration at process level and support of decision-making to improve the software processes, while we focus on integration at data level to support product and process improvement. In the context of agile and continuous development, Kleebaum et. al [49] extract decision knowledge from data recorded in source code, version control and tracking systems and transform the extracted data into decisions to support developers to improve coding activities and results. For the same purpose, Johanssen et. al [50] propose an infrastructure that enables to systematically extract and manage usage and decision knowledge from data recorded in source code repositories. Similar to our work, [49] and [50] extract and integrate data from applications to support decision making. However, as we argue regarding the aforementioned MSR works, they do not address semantic aspects. Moreover, although related to our work due to their integration concern, [48], [49] and [50] do not cover aspects related to Scrum.

7 Conclusion and Future Work

Scrum has been increasingly adopted in software development and organizations often use several applications to support different aspects of the Scrum software process, making it difficult to obtain integrated data to support decision making. One of the main challenges when integrating applications is to deal with semantic conflicts that occur whenever applications adopt different meanings to the same information item. Considering the successful use of ontologies to address semantics in integration initiatives [13] and the lack of a comprehensive ontology about Scrum, in this paper, we presented the Scrum Reference Ontology (SRO), which provides a conceptualization about Scrum in the Software Engineering context and can be used as a reference model in integration efforts.

To the best of our knowledge, SRO is the first reference ontology about Scrum grounded in concepts of a Software Engineering ontology network. This design decision allowed us to reuse general Software Engineering concepts (e.g., project, stakeholder, intended and performed processes and activities) and focus on specifics of Scrum. Moreover, reusing concepts from other SEON [17] ontologies helps understand Scrum concepts in the broader Software Engineering context. For instance, by integrating SRO to SEON, it is made explicit that a User Story is a Requirements Artifact (i.e., the record of a requirement), a Deliverable is a Software Item, and an Acceptance Criterion is a Requirement (i.e., a goal to be met by the produced software item). Furthermore, since SRO is integrated to SEON, it takes advantage of SEON whole conceptualization. Therefore, it is possible to

[18] https://mantisbt.org
[19] https://subversion.apache.org

extract a SEON fragment larger than SRO to address other aspects in the agile context, such as coding and configuration management, which are already addressed in SEON.

We used SRO to address semantics in an architecture to integrate Azure DevOps and Clockify and applied our integrated solution in the software development unit of a Brazilian government agency. SRO was used as interlingua, helping properly integrate the applications and avoid semantic conflicts. The ontology model was turned into an information model, which facilitated the integration of the various applications' elements. The architecture is made up mainly of a database and services, which were built using model-driven development principles (i.e., information present in models were used directly to build software artifacts), allowing to exchange and share data and events among applications, which decreases the complexity to integrate applications and allows to reuse the middleware to integrate other applications.

The proposed architecture can be extended to include other applications. Moreover, it is possible to replicate its development process to create similar architectures involving other applications. The results of using the integrated solution in a real project indicated that it was useful to provide data to support decision making to improve software development work. It helped identifying and solving problems in the execution of the Scrum process and decreasing manual work.

It is important to note that SRO does not constraint the development process agility or flexibility. SRO is used to produce a semantic interoperability solution that integrates data from different applications used by the team and shows integrated data in dashboards. We must emphasize that there is no extra documentation burden imposed by the ontology-based approach. The reference ontology and integration work is done completely outside of the agile product development trajectory. The solution itself is then used during the development process. As a consequence, existing data (i.e., data that is already stored in the applications used by the team) is integrated without manual intervention and presented in dashboards, providing useful information that could otherwise not be easily obtained from the applications. The information can be thus used to support daily activities and decision making, contributing to improve process and product, without extra effort from the team.

Besides supporting application integration, SRO can be used to aid other integration efforts. For example, SRO conceptualization can be useful to harmonize different Scrum standards [51] as well as to annotate and integrate data in semantic web and semantic documentation [52] contexts. Moreover, SRO conceptualization can be used to support knowledge management solutions (i.e., serving as a basis to a knowledge management system) and knowledge workers can use it for communication purposes.

As limitation of this work, we highlight its evaluation, which involved two applications, was applied in a single organization and involved the participation of the first author. Hence, as future work, we intend to use SRO in other integration initiatives and also extend the produced integrated solution to add other applications (e.g., code quality, configuration management). This can be done by exploring other SEON ontologies (e.g., System and Software Ontology, Configuration Management Process Ontology) to provide integrated data covering other aspects relevant in the Scrum context. In the context of another project (ongoing work), we have followed the process shown in Section 5 and explored a SEON fragment larger than SRO to integrate other applications. We also plan to extend SRO conceptualization to address other relevant processes in the agile context, such as continuous integration.

8 Acknowledgements

This work is partly funded by CNPq (grants number 313687/2020-0 and 312123/2017-5) and Fapes (grant number 180/2017).

9 References

[1] B. Julian, J. Noble, C. Anslow, Agile Practices in Practice: Towards a Theory of Agile Adoption and Process Evolution, in: P. Kruchten, S. Fraser, F. Coallier (Eds.), Agil. Process. Softw. Eng. Extrem. Program. XP 2019, Springer International Publishing, Cham, 2019: pp. 3–18. https://doi.org/10.1007/978-3-030-19034-7_1.

[2] J. Schwaber, Ken; Sutherland, The scrum guide-the definitive guide to scrum: The rules of the game, (2013) scrum.org.

[3] V.S. Fonseca, M.P. Barcellos, R.D.A. Falbo, An ontology-based approach for integrating tools supporting the software measurement process, Sci. Comput. Program. 135 (2017) 20–44. https://doi.org/10.1016/j.scico.2016.10.004.

[4] E. Brynjolfsson, L.M. Hitt, H.H. Kim, Strength in Numbers: How Does Data-Driven Decision making Affect Firm Performance?, SSRN Electron. J. 1 (2011). https://doi.org/10.2139/ssrn.1819486.

[5] R.B. Svensson, R. Feldt, R. Torkar, The Unfulfilled Potential of Data-Driven Decision Making in Agile Software Development, in: vol 355. S. Kruchten P., Fraser S., Coallier F. (eds) Agile Processes in Software Engineering and Extreme Programming. XP 2019. Lecture Notes in Business Information Processing (Ed.), Int. Conf. Agil. Softw. Dev., Springer, Cham, 2019: pp. 69–85.

https://doi.org/10.1007/978-3-030-19034-7_5.
[6] R. Calhau, R. Falbo, An Ontology-Based Approach for Semantic Integration, in: 14th IEEE Int. Enterp. Distrib. Object Comput. Conf., 2010: pp. 111–120. https://doi.org/10.1109/EDOC.2010.32.
[7] R. Chatley, Supporting the developer experience with production metrics, in: Proc. - 2019 IEEE/ACM Jt. 4th Int. Work. Rapid Contin. Softw. Eng. 1st Int. Work. Data-Driven Decis. Exp. Evol. RCoSE/DDrEE 2019, IEEE, 2019: pp. 8–11. https://doi.org/10.1109/RCoSE/DDrEE.2019.00009.
[8] H.H. Olsson, H. Alahyari, J. Bosch, Climbing the Stairway to Heaven: A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software, in: 2012 38th Euromicro Conf. Softw. Eng. Adv. Appl., IEEE, 2012: pp. 392–399. https://doi.org/10.1109/SEAA.2012.54.
[9] B. Fitzgerald, K.J. Stol, Continuous software engineering: A roadmap and agenda, J. Syst. Softw. 123 (2017) 176–189. https://doi.org/10.1016/j.jss.2015.06.063.
[10] J. Bosch, Continuous Software Engineering: An Introduction, in: Contin. Softw. Eng., Springer International Publishing, Cham, 2014: pp. 3–13. https://doi.org/10.1007/978-3-319-11283-1_1.
[11] H. Wache, T. Vögele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, S.H. Ubner, Ontology-Based Information Integration: A Survey, Int. J. Artif. Intell. (2002).
[12] S. V Pokraev, Model-driven semantic integration of service-oriented applications, Thesis, 2009. https://research.utwente.nl/en/publications/model-driven-semantic-integration-of-service-oriented-application-2.
[13] J.C. Nardi, R. de Almeida Falbo, J.P.A. Almeida, Foundational Ontologies for Semantic Integration in EAI: A Systematic Literature Review, in: Conf. e-Business, e-Services e-Society, I3E 2013 Collab. Trust. Privacy-Aware e/m-Services, Springer, Berlin, Heidelberg, 2013: pp. 238–249. https://doi.org/10.1007/978-3-642-37437-1_20.
[14] D. Parsons, Agile software development methodology, an ontological analysis, in: Proc. 9th Int. Conf. Appl. Princ. Inf. Sci., Proceedings of 9th International Conference on Applications and Principles of Information Science, 2010: pp. 5–8. https://doi.org/10.13140/2.1.3298.6883.
[15] S. Kiv, S. Heng, M. Kolp, Y. Wautelet, Agile Methods Knowledge Representation for Systematic Practices Adoption, in: Int. Conf. Agil. Softw. Dev., Springer, Cham, 2019: pp. 19–34. https://doi.org/10.1007/978-3-030-19034-7_2.
[16] Y. Lin, V. Hilaire, N. Gaud, A. Koukam, Scrum Conceptualization Using K-CRIO Ontology, in: Int. Symp. Data-Driven Process Discov. Anal., Berlin, Heidelberg, 2012: pp. 189–211. https://doi.org/10.1007/978-3-642-34044-4_11.
[17] F. Borges Ruy, R. de A. Falbo, M.P. Barcellos, S.D. Costa, G. Guizzardi, SEON: A software engineering ontology network, in: Lect. Notes Comput. Sci. (Including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), 2016: pp. 527–542. https://doi.org/10.1007/978-3-319-49004-5_34.
[18] L. Rising, N.S. Janoff, The Scrum Software Development Process for Small Teams, IEEE Softw. 17 (2000) 26–32. https://doi.org/10.1109/52.854065.
[19] S. Izza, Integration of industrial information systems: From syntactic to semantic integration approaches, Enterp. Inf. Syst. 3 (2009) 1–57. https://doi.org/10.1080/17517570802521163.
[20] P. Wegner, Interoperability, ACM Comput. Surv. 28 (1996) 285–287.
[21] F. Vernadat, Interoperable Enterprise Systems: Principles, Concepts, and Methods, Annu. Rev. Control. 31 (2007) 137–145. https://doi.org/10.1016/j.arcontrol.2007.03.004.
[22] M. Themistocleous, Zahir Irani, P.E. Love, Evaluating the integration of supply chain information systems: A case study, Eur. J. Oper. Res. 159 (2004) 393–405. https://doi.org/10.1016/j.ejor.2003.08.023.
[23] R. Studer, V.R. Benjamins, D. Fensel, Knowledge engineering: principles and methods., Data Knowl. Eng. 25 (1998) 161–197. https://doi.org/10.1016/S0169-023X(97)00056-6.
[24] A. Scherp, C. Saathoff, T. Franz, S. Staab, Designing Core Ontologies, Appl. Ontol. 6 (2011) 177–221. https://doi.org/10.3233/AO-2011-0096.
[25] G. Guizzardi, On Ontology, Ontologies, Conceptualizations, Modeling Languages, and (Meta)Models, in: Proc. 2007 Conf. Databases Inf. Syst. IV Sel. Pap. from Seventh Int. Balt. Conf. DB&IS'2006, IOS Press, NLD, 2007: pp. 18–39.
[26] R. de A. Falbo, SABiO: Systematic approach for building ontologies, in: ONTO. COM/ODISE@ FOIS., 2014.
[27] M.C. Suárez-Figueroa, A. Gomez-Perez, E. Motta, A. Gangemi, Introduction: Ontology Engineering in a Networked World, in: Ontol. Eng. a Networked World, Springer, Berlin, Heidelberg, 2012: pp. 1–6. https://doi.org/10.1007/978-3-642-24794-1.
[28] A. Hevner, S. March, J. Park, S. Ram, Design Science in Information Systems Research, MIS Q. 28 (2013) 75–105. https://doi.org/10.2307/25148625.
[29] A. Hevner, A Three Cycle View of Design Science Research, Scand. J. Inf. Syst. 19 (2007) 4.
[30] K. Schwaber, M. Beedle, Agile Software Development With Scrum, Upper Saddle River: Prentice Hall, 2002.
[31] M. Cohn, Succeeding with Agile: Software Development Using Scrum, Pearson Education, 2010.
[32] R. Kenneth, Essential Scrum: A practical guide to the most popular Agile process, Addison-Wesley, 2012.
[33] T. Satpathy, ed., A Guide to the Scrum Body of Knowledge : SBOK Guide, Scrumstudy a brand of VMEdu, Inc, 2013. https://www.scrumstudy.com/sbokguide.
[34] J. Brank, M. Grobelnik, D. Mladenic, A survey of ontology evaluation techniques, in: Proc. Conf. Data Min. Data Warehouses (SiKDD 2005), Citeseer Ljubljana, Slovenia, Citeseer Ljubljana, Slovenia, 2005: pp. 166–170.
[35] R. Carraretto, Separating Ontological and Informational Concerns : A Model-driven Approach, Master Thesis, Universidade Federal do Espírito Santo, 2012. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6406251.
[36] Y. Lin, V. Hilaire, N. Gaud, A. Koukam, Using K-CRIO Ontology, in: Int. Symp. Data-Driven Process Discov. Anal., 2012: pp. 189–211.
[37] M. Cossentino, N. Gaud, S. Galland, V. Hilaire, A. Koukam, A Holonic Metamodel for Agent-Oriented Analysis and Design, in: Int. Conf. Ind. Appl. Holonic Multi-Agent Syst., Springer, Berlin, Heidelberg, 2007: pp. 237–246. https://doi.org/10.1007/978-3-540-74481-8_23.
[38] G. Guizzardi, Ontological Foundations for Structural Conceptual Models, PhD Thesis, University of Twente, 2005. https://research.utwente.nl/en/publications/ontological-foundations-for-structural-conceptual-models.
[39] E. Damiani, A. Colombo, F. Frati, C. Bellettini, A Metamodel for Modeling and Measuring Scrum Development Process, 2007. https://doi.org/10.1007/978-3-540-73101-6_11.
[40] H. Ayed, B. Vanderose, N. Habra, A metamodel-based approach for customizing and assessing agile methods, in: Proc. - 2012 8th Int. Conf. Qual. Inf. Commun. Technol. QUATIC 2012, 2012: pp. 66–74. https://doi.org/10.1109/QUATIC.2012.11.
[41] F. Harmsen, S. Brinkkemper, J. Oei, Situational method engineering for information system project approaches, 1994.
[42] A.E. Hassan, The road ahead for mining software repositories, Proc. 2008 Front. Softw. Maintenance, FoSM 2008. (2008) 48–57. https://doi.org/10.1109/FOSM.2008.4659248.
[43] A.-L.L. Mattila, K. Systä, O. Sievi-Korte, M. Leppänen, T. Mikkonen, Discovering Software Process Deviations Using

Visualizations, in: Lect. Notes Bus. Inf. Process., 2017: pp. 259–266. https://doi.org/10.1007/978-3-319-57633-6_18.
[44] H. Malik, A.E. Hassan, Supporting software evolution using adaptive change propagation heuristics, IEEE Int. Conf. Softw. Maintenance, ICSM. (2008) 177–186. https://doi.org/10.1109/ICSM.2008.4658066.
[45] G. Destefanis, M. Ortu, S. Counsell, S. Swift, M. Marchesi, R. Tonelli, Software development: Do good manners matter?, PeerJ Comput. Sci. 2016 (2016). https://doi.org/10.7717/peerj-cs.73.
[46] D. Čubranić, G.C. Murphy, J. Singer, K.S. Booth, Hipikat: A project memory for software development, IEEE Trans. Softw. Eng. 31 (2005) 446–465. https://doi.org/10.1109/TSE.2005.71.
[47] S. Kim, T. Zimmermann, K. Pan, E.J.J. Whitehead, Automatic Identification of Bug-Introducing Changes, in: 21st IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE 2006), Tokyo, Japan, 2006: pp. 81–90. https://doi.org/10.1109/ASE.2006.23.
[48] L.D.C. Renault, M.P. Barcellos, R. de Almeida Falbo, Using an ontology-based approach for integrating applications to support software processes, in: ACM Int. Conf. Proceeding Ser., 2018: pp. 220–229. https://doi.org/10.1145/3275245.3275269.
[49] A. Kleebaum, J.O. Johanssen, B. Paech, R. Alkadhi, B. Bruegge, Decision knowledge triggers in continuous software engineering, in: Proc. - Int. Conf. Softw. Eng., 2018: pp. 23–26. https://doi.org/10.1145/3194760.3194765.
[50] J.O. Johanssen, A. Kleebaum, B. Paech, B. Bruegge, Continuous software engineering and its support by usage and decision knowledge: An interview study with practitioners, in: J. Softw. Evol. Process, John Wiley and Sons Ltd, 2019. https://doi.org/10.1002/smr.2169.
[51] F. Ruy, E. Souza, R. Falbo, M. Barcellos, Software Testing Processes in ISO Standards: How to Harmonize Them?, in: Proc. 16th Brazilian Symp. Softw. Qual., 2017: pp. 296–310.
[52] E.C. Bastos, M.P. Barcellos, R. de Almeida Falbo, Using semantic documentation to support software project management, J. Data Semant. 7 (2018) 107–132. https://doi.org/10.1007/s13740-018-0089-z.