SCODA - Framework for Software Capability Representation and Inspection

Hsu Myat Win[1][0009−0001−0422−4496], Sebastian Rodriguez[1][0000−0002−0514−9221], John Thangarajah[1][0000−0002−7699−6444], and Andrew Warhurst[2]

[1] Royal Melbourne Institute of Technology, Melbourne, Australia
{hsu.myat.win, sebastian.rodriguez, john.thangarajah}@rmit.edu.au
[2] Defence Science and Technology Group, Australia
andrew.warhurst1@defence.gov.au

Abstract. Software composition remains a significant challenge in today's technology landscape, especially when dynamically integrating and collaborating among different systems. Before software composition, analyzing software capabilities (SC) in a system is a crucial and challenging step. In this work, we introduce an ontology-based framework for SC representation and inspection. Our novelty lies in our model called the Software Capabilities for Open and Dynamic Architectures(SCODA), which represents software capability in the system by providing essential data for effective software inspection. Unlike previous approaches that primarily focused on software quality or reusability, SCODA emphasizes a structured representation of capabilities by capturing dynamic factors such as preconditions and effects. We used the Multi-Agent Programming Contest (MAC) to develop the model and the Drone Courier System (DCS) to illustrate our approach, demonstrating its utility in real-world scenarios. We have implemented a tool called QueryCap designed for inspecting software capabilities from the dataset (i.e., a list of SCs found in the system) for a given task.

Keywords: Software capability · Ontology.

1 Introduction

Software composition presents one of the most challenging issues in today's rapidly evolving technological landscape, particularly when integrating multiple dynamic software systems. Before engaging in software composition, it is crucial to inspect the functionalities of existing software. This step is also challenging because misrepresenting software functional capabilities may lead to deviations from the task's goal or provide inaccurate information. For example, one of the faults in integrating a Web API like Adyen's payment system is incomplete or insufficient information on how to implement specific API features or functionalities, which can hinder proper integration [21]. Under ISO 9000:2015(en), capability refers to "the ability of an object to realize an output that will fulfil the requirements for that output". In the area of system engineering, capability refers

to "the ability of a system to execute a particular course of action or achieve a desired effect, under a specified set of conditions " [22]. In software engineering, the capability maturity model (CMM) has introduced processes as capabilities, defining how to assess specific qualities of software engineering processes, and offering an understanding of the current state of software systems [13]. We follow the definition from system engineering perspective because in our project, we specifically focus on functional capability in the software rather than the software engineering process or business requirement.

Prior research has primarily focused on assessing and evaluating software capabilities within specific requirements, catering solely to organizational needs [8]. Similarly, some work typically focuses on software quality to determine if it meets business requirements [9]. However, this approach often neglects the exploration of inherent abilities within the existing software. On the other hand, some researchers have introduced feature extractions from systems to compose software artifacts written in different languages [10, 11]. Additionally, software product lines have aimed to identify reusable functionalities from previous software versions [1, 12]. Since they focus on software composition, there is still a need to address the investigation of software abilities across different systems or software before composition. Similarly, some researchers explore assessing the reusability of software components through software capability profiling [2–4]. However, there remains a gap in addressing effects that can impact the features of the entity and influence the expected output. For instance, available charge in a drone's battery could affect its ability to complete its task, yet existing studies often overlook such critical factors.

In this project, we introduce an ontology-based framework for representing software capability, which we have termed the SCODA model. The SCODA model includes six main concepts: entity, action, effect, condition, feature, and resource. We chose the standard ontology because it provides us with a consistent framework for representing and organizing knowledge, promoting reusability across different applications and projects. It also enables seamless integration with other existing ontologies, facilitating data interoperability and reducing the effort required to combine different data sources. We used the Multi-Agent Programming Contest (MAC)[3] as our initial study for developing the model. MAC is an annual event that allows participants to showcase their innovations in the field of multi-agent systems and artificial intelligence (AI). This competition involves developing multi-agent systems to solve cooperative tasks in a dynamically changing environment. Agents are autonomous entities that collaborate to solve tasks. An agent possesses the ability to learn and make autonomous decisions, leveraging interactions with neighboring agents or the environment to acquire new knowledge and execute actions to accomplish their designated tasks within the system [20]. We chose MAC for analyzing software capability because (1)in MAC, agents (e.g., a drone or a truck) work dynamically to achieve specific goals, aligning with SCODA's emphasis on open and dynamic architecture, and (2) our research group has participated in this competition in the past,

[3] https://multiagentcontest.org/2018/

giving us expertise within that domain. For illustration, we developed a Drone Courier System (DCS) as a second case study due to its simplicity for the paper. Specifically, we created instances of the SCODA model for the DCS. We then developed a mechanism to inspect and assess the software capabilities within the dataset for a given task (e.g., do we have a software capability that can deliver a package from a warehouse to a customer?). We choose SPARQL Protocol and RDF Query Language (SPARQL) for developing software capability inspection mechanism because it allows for powerful querying and manipulation of data within Resource Description Framework (RDF) graphs, enabling complex data retrieval and integration in ontology-based systems. Overall, our aim is to provide insights into the software capabilities of existing software for inspection and our contributions are as follows:

- developing a model for software capability representation;
- providing a query language to inspect the software capability; and
- presenting a case study that illustrates the use of our model.

2 Background

In this section, we provide background information on the relevant technologies for the project: ontology engineering, utilized for modeling, and SPARQL, employed for SC inspection. **Ontology engineering** involves creating knowledge maps (ontologies) to efficiently organize information. An ontology represents structured knowledge within a specific domain, encompassing concepts, and relationships [16]. Ontologies serve as specifications for sharing and reusing knowledge across applications [15]. **Web Ontology Language (OWL)** is a standard endorsed by the W3C for constructing OWL knowledge models [14]. It is a semantic web language designed to model intricate knowledge about entities, groups, and their relationships. OWL has found extensive use in modeling concepts such as access control policies [18] and privacy in medical data [17]. Knowledge expressed in OWL can be utilized by computer programs to verify consistency or make implicit knowledge explicit. Hence, our approach is designed based on ontology engineering principles. The ontology is developed using information gathered from human sources (such as domain experts), structured sources (like databases), or unstructured sources (such as books). This information is then assigned to the ontology in the form of concepts, relationships, and definitions. The proposed ontology, referred to as SCODA, was constructed using Protégé 5.0. Protégé is a freely available ontology editor and information management framework developed by the Biomedical Informatics Research Center at Stanford University. **SPARQL** is a standardized query language and protocol for querying RDF data. It allows users to retrieve and manipulate data in RDF format, and supports the principles of the Semantic Web by enabling the integration and querying of data with rich semantics. It also allows applications to understand and reason about the relationships between entities, making it valuable for knowledge representation and discovery.

Some studies emphasize examining software capabilities and resource awareness on available hardware capacity such as CPU and memory [5], neglecting other required resource types (e.g., battery power). Proposals for defining capability include preconditions, effects, input/output, hardware, and parameters during domain planning [6, 7]. However, these often focus on post-conditions rather than dynamic factors influencing process outcomes. In our approach, we consider both effects and preconditions as integral parts of software capability, as they significantly influence software functionality. For example, when a drone moves from one location to another, its battery decreases (we call this an effect), which could lead to the drone not reaching its destination. Therefore, having enough battery should be a precondition for a drone to perform certain tasks and provide specific output. Solely observing the output of software functions is insufficient for defining software capability. Instead, analyzing the preconditions and effects that influence the outcome is crucial. Therefore, our approach aims to address the shortcomings of previous research by considering dynamic factors that influence software capability outcomes, thus providing a more robust model for representing software capabilities (SC).

3 Approach

As shown in Figure 1, our ontology-based approach begins with domain knowledge elicitation followed by software capability (SC) representation modeling (i.e., SCODA model). We also developed a querying mechanism for SC inspection. The objective of the domain knowledge elicitation step is to gather and extract relevant information within the domain (e.g., Multi-Agent City contest (MAC)). This process involves comprehending the context, and identifying their functionalities and essential elements. We begin by reviewing documents and conducting simulations to verify functionalities. For instance, in the context of MAC, we validated that a drone can perform functions such as loading, moving, and charging. Subsequently, we analyze the necessary elements and conditions required to support these functionalities. E.g., for the moving function, the drone needs a battery and knows the destination.

3.1 SCODA model

Inspired by Jabardi and Hadi [19], who utilized ontological engineering to detect and classify fake accounts on Twitter, our ontology development comprises four steps: (1) defining the domain and scope of the ontology (specifically focusing on software capability), (2) establishing classes and their hierarchical structure, (3) specifying class properties, and (4) assigning values or assertions to these properties (referred to as "slot facts"). Specifically, we explored candidate components to represent Software Capability (SC), considering fundamental elements in systems or software. We defined a component as an asset representing a complete aspect of software capability. We then examined which components could influence the expected output of a given task and their characteristics. We identified

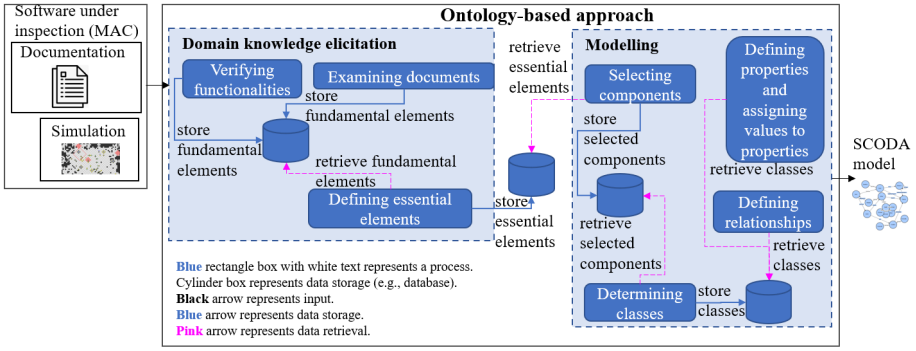

Fig. 1: The framework of the proposed method.

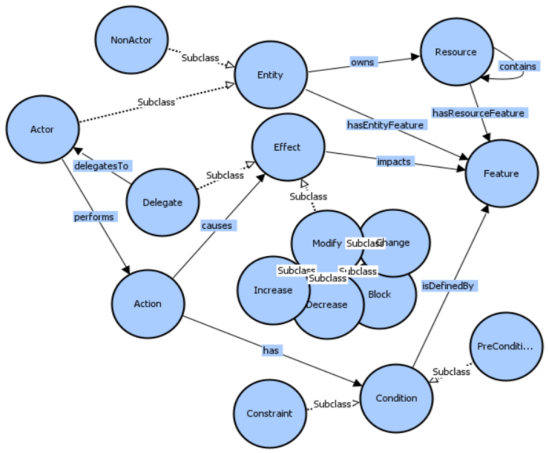

Fig. 2: SCODA model.

the essential components required to represent SC, seeking a balance between generality and specificity. Additionally, we refined the definition and naming of each SC component, clarifying terms like *resource*. We built our model using Protégé, where classes represent components in SC. Subsequently, we investigated connections among these components (i.e., relationships between classes) and brainstormed class properties while assigning values to these properties.

Figure 2 illustrates the SCODA model, comprising six key concepts:

1. **Entity**: It is a complete and functional part of a system, capable of existing independently as either a software component, hardware component, or a combination of both. For example, consider a drone that needs to go to a warehouse. In this context, both the drone and the warehouse would be entities; however, the drone can perform the action of going, while the warehouse simply provides the context to complete the environment. There-

fore, we created two subclasses under entity, which we called **Actor** and **Non-actor**. Specifically, an actor can perform actions or trigger software functions, whereas a non-actor does not initiate actions but serves as a functional part of the system that provides context or environment for actions.

2. **Action**: A process or operation executed by an actor that results in changes to the system or the environment. An action typically involves manipulating resources, affecting conditions, and changing properties. For example, a drone's movement action could change its location.
3. **Resource**: A resource represents a physical or informational element/component that an entity possesses, which can be used, accessed, or depended on by actors to perform tasks. It does not function independently but supports the functioning of the system, being utilized, consumed, or accessed during system operations. For instance, a drone's battery.
4. **Condition**: It refers to any requirement or circumstance that influences an action. For example, before going to a warehouse, a drone must have enough battery. In this context, we introduced **Precondition** as a subclass of condition. Besides precondition, we may need some limitations, such as "Drone must reach a warehouse within 1 minute". Therefore, we introduced **Constraint** as a subclass as well. Specifically, a precondition refers to the condition that must be satisfied for an action to occur, while a constraint denotes any limitation or restriction that governs the action.
5. **Effect**: A measurable or observable change or impact when an action is triggered. It describes the impact of actions on entities and resources. For example, upon going to a warehouse, a drone's battery would experience a negative impact. Conversely, upon charging, a drone's battery would experience a positive impact. Therefore, we introduced the **Modify** concept as a subclass of **Effect**. Specifically, the **Modify** effect can alter the feature of an entity or resource by *increasing*, *decreasing*, *changing*, or *blocking* it. In addition to altering features, an effect can delegate a task to an actor to perform. For example, in MAC, an initiator agent responsible for evaluating and allocating all types of jobs can delegate the selected job to an agent who can perform the action such as "building a well". Thus, we introduced the **Delegate** concept as a subclass of **Effect**.
6. **Feature**: An attribute or property of an entity or a resource. For instance, the location of an agent.

Relationships between concepts. After defining each concept in our model, we now explore how these concepts relate to each other, revealing the connections and interactions among them. We have provided relationships between concepts in Figure 1. Dotted lines represent subclass relationships, while solid lines represent relationships between the concepts. Following are the relationships between the concepts -

performsAction represents a relationship between an actor and an action. Specifically, an actor can perform an action.

causalEffect represents the relationship between an action and an effect on resources or entities. It helps us understand how a specific action leads to observable changes or effects on resources or entities.
impactsFeature represents the relationship to illustrate how an effect impacts and changes the specific feature of a resource or entity. It helps us to understand the direct consequences or changes impacted by a particular effect on the relevant features.
hasResourceFeature represents the relationship between a resource and its associated attributes, represented by features.
hasEntityFeature represents the relationship between an entity and its associated attributes, represented by features.
ownsResource represents a relationship between an entity and a resource. Specifically, an entity can own a resource.
contains represents a relationship between a resource and its own resource. Specifically, a resource can contain resources.
hasCondition represents a relationship between an action and conditions. Before triggering an action, conditions need to be satisfied.
isDefinedBy represents a relationship between a condition and features. Specifically, the condition is defined by features.
delegatesTo represents a relationship between a delegate effect and an actor.

Insights. We derived the concept of an entity from MAC, where multiple agents (i.e., drones, trucks, or cars) can perform actions, while warehouses and charging stations support the context of MAC. We then applied this theory to generalize real-world systems and arrived at the definition above. Some researchers have suggested that a resource is a type of entity [20]. However, in our model, we distinguish resources from entities because resources have distinct features, such as supporting functions without being functional parts of a system (its features determine the expected output), and they exist dependently on entities. Our analysis revealed that rather than direct impacts (i.e., a direct relationship) from operations (referred to as *Actions*) to resources, we need a conceptual representation (an abstract representation) between them, which we term *Effect*. This possesses characteristics distinct from both actions and resources and encompasses more than just a relationship between them. Consequently, we introduced *Effect* as a concept in representing software capability and defined it as such if it can alter the current state of a resource and it is triggered by an action. Since resources can have varied states, we introduced the *Feature* concept (e.g., battery can have not only *current_ charge* but also *current_ temperature*) and an effect could impact these features rather than the resource itself. Specifically, we define the state as a feature if it can be changed, increased, decreased, or blocked by an effect. Finally, our study revealed that certain actions need to trigger the main action, leading us to introduce the concept of the *Delegate* effect.

3.2 SC inspection mechanism

The objective of the SC inspection mechanism is to facilitate the analysis and understanding of software capabilities. We used SPARQL-based query because,

Algorithm 1: Inspecting SC

```
Input: Feature
Procedure InspectingSC (Feature)
    effectsTypes ← QueryEffectsTypes(Feature);
    DisplayListofEffectsTypes(effectsTypes);
    selectedEffectType ← GetUserSelection(effectsTypes);
    suggestedSC ← RetrieveSC(selectedEffectType);
    DisplaySCDetails(suggestedSC);
Procedure QueryEffectsTypes(Feature)
    /* Return effect types based on the provided feature. */
Procedure DisplayListofEffectsTypes(effectsTypes)
    /* Display the effect types list (effectsTypes) to select. */
Procedure GetUserSelection(effectsTypes)
    /* Return the selected effect type chosen by the user */
Procedure RetrieveSC(selectedEffectType)
    /* Return the SC details based on the provided effectType */
Procedure DisplaySCDetails(suggestedSC)
    /* Display suggested SC with corresponding actor, action,
       effect, feature, and entity. */
Main:;
Feature ← Input("Select the feature: ");
InspectingSC(Feature);
```

ideally, we aimed for it to be as close to natural language as possible. This would require Natural Language Processing (NLP); which we aim for future work. As shown in Algorithm 1, we begin with an input parameter *Feature*, which represents the specific feature of the entity or resource. Initially, the procedure *InspectingSC* queries the ontology using *QueryEffectsTypes(Feature)* to retrieve a list of effect types that impact the specified feature. These effect types are then displayed to the user through *DisplayListofEffectsTypes(effectsTypes)*, allowing for user selection via *GetUserSelection(effectsTypes)*. Once the user selects an effect type, *RetrieveSC(selectedEffectType)* fetches details of the corresponding Software Capability (SC). Finally, *DisplaySCDetails(suggestedSC)* presents the suggested SC along with its associated actor, action, effect, feature, and entity details to the user.

Implementation. We implemented in a tool named QueryCap, designed for querying software capabilities from the dataset (i.e., a list of SCODA models of a system). We use RDF for dataset creation because RDF is a standard data model in semantic web technologies, enabling structured description and exchange of data on the web. QueryCap is a standalone Java program and requires JDK 17.0.9, JavaFX SDK 21.0.1, and Apache Jena 4.9.0. It is a Windows application and comprises three components: the User Interface (UI), Data Extraction, and

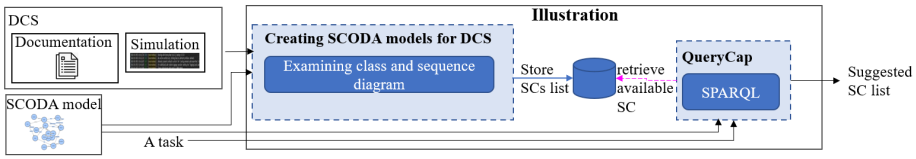

Fig. 3: The process for illustrating our model.

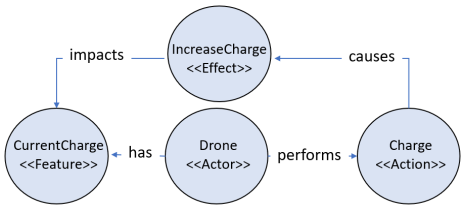

Fig. 4: Software capability that can increase the Drone's battery.

SPARQL scripts. When passing parameters across the class, data is stored in the object model.

4 Case studies

To illustrate our approach, we present a much simpler case study (i.e., Drone Courier System (DCS)) because Multi-Agent Programming Contest(MAC), which we used for model development, requires a lot of domain knowledge to understand. Therefore, for simplicity in illustration purposes, we developed DCS example as a second case study to present our work here. DCS case study involves delivering packages to customers, where the vehicle scheduler coordinates with the warehouse to prepare the package, and a drone is deployed to transport and deliver the package to the customer's address.

The process of our illustration is shown in Figure 3. First, we created SCODA models using our ontology-based model, DCS's documentation and DCS's simulation. Next, we stored the dataset (i.e., list of SCs) on the local machine, and then the SC inspection mechanism implemented as QueryCap extracts the relevant software capabilities based on the given task.

Creating SCODA models. Using SCODA model and DCS's UML class diagram and sequence diagram, we created SCODA models for DCS. We take a class as an actor if it can trigger actions/operations while we define it as a resource if it can be consumed/changed and we define it as a non-actor if neither of that. We identified attributes of each class as features if they possess the distance property (e.g., drone's *currentLoad*). These attributes are further classified based on whether they can be changed, increased, decreased, or blocked, thus determining their effects and types. We classified operations of a class as actions if they impact any of the defined features. Because of page limitation,

```
SELECT ?feature
WHERE {
  ?entity hasEntityFeature ?feature .
  FILTER(str(?entity) = str(Drone))
}
```

(a) SPARQL

	feature
1	BatteryCapacity
2	DroneStorageCapacity
3	CurrentCharge
4	DroneLocation
5	CurrentLoad

(b) Result

Fig. 5: Finding features for Drone.

we illustrate SC which can charge Drone's battery. As shown in Figure 4, when *Drone* triggers *Charge* action, it causes a positive effect on the battery's current charge.

4.1 Inspecting software capability

We use SPARQL to extract SC from the dataset. We provided a sample query in Figure 5. In the **WHERE** clause, **?entity hasEntityFeature ?feature .** is a triple pattern to retrieve all features (**?feature**) belonging to entities (**?entity**). **FILTER(str(?entity) = str(Drone))** is a filter clause to check if the **?entity** is **Drone**. Specifically, this query retrieves all features of Drone, using RDF triples and a filter condition in SPARQL.

Based on algorithm 1, to illustrate our approach, we investigate SCs that can alter a drone's battery. The query begins by selecting types of effects that impact a drone's battery charge. Specifically, in the **WHERE** clause, the query defines conditions **?effect ?impactsObj ?feature .** and **?entity ?hasObj ?feature .** to select all effects that impact the feature of entity, filtered with conditions **str(?feature)=str(CurrentCharge)** and **str(?entity)=str(Drone)** to verify if the feature is *CurrentCharge* of *Drone*. This query specifically retrieves all effects (i.e., An effect that can increase the battery and An effect that can decrease battery) that impact the Drone's battery. We assume the user chooses "An effect which can increase battery", our inspection continues to extract actions which cause that effect and actors who perform those actions. Specifically, in the **WHERE** clause, the query defines conditions **?action ?causes ?effect .** and **?actor ?performs ?action .** to extract all actions and corresponding actors. In the filter clause, we have **str(?effect)= str(Increase)** to verify if the effect is *Increase*. This query specifically retrieves all actions that can increase Drone's battery and actors that can perform those actions. The result of that query is shown in Figure 6.

5 Discussion and conclusion

In this project, we developed a model for representing software capabilities which we termed the SCODA model - an ontology-based framework to represent and inspect software capabilities. We used Multi-Agent Programming Contest (MAC)

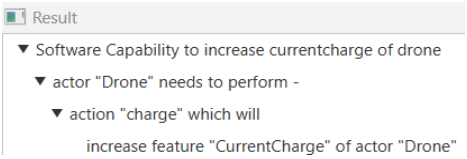

Fig. 6: SC which can able to increase Drone's battery.

for developing the model and the Drone Courier System (DCS) to illustrate our approach. We also developed an inspection mechanism for software capabilities with the help of SCODA model. For illustration purposes, we implemented a tool named QueryCap[4] for querying capabilities from the dataset and our demonstration shows that our approach can able to extract the relevant software capabilities for a given task. These insights enable us to make informed decisions when dynamically integrating and collaborating among different systems.

Although this approach provides a robust framework for conceptualizing software capabilities, it lacks real-time simulation capabilities and instance-level modeling. Without dynamic analysis and runtime implementation, potential runtime issues such as concurrency problems remain unaddressed, providing scope for future work. However, our current approach lays the foundation for software capability inspection using SCODA models and, in the future, we aim to integrate dynamic analysis into our models to address runtime issues such as concurrency problems.

6 Acknowledgments

This research is supported by the Commonwealth of Australia as represented by the Defence Science and Technology Group.

References

1. Debbiche, J., Lignell, O., Krüger, J., Berger, T.: Migrating Java-based apo-games into a composition-based software product line. In: Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A, pp. 98–102 (2019)
2. Belfadel, A., Laval, J., Cherifi, C.B., Moalla, N.: Toward service orchestration through software capability profile. In: Enterprise Interoperability VIII: Smart Services and Business Impact of Enterprise Interoperability, pp. 385–395 (2019). Springer
3. Belfadel, A., Laval, J., Bonner Cherifi, C., Moalla, N.: Semantic software capability profile based on enterprise architecture for software reuse. In: Reuse in Emerging Software Engineering Practices: 19th International Conference on Software and Systems Reuse, ICSR 2020, Hammamet, Tunisia, December 2–4, 2020, Proceedings 19, pp. 3–18 (2020). Springer

[4] https://github.com/hmteams/scoda

4. Belfadel, A., Amdouni, E., Laval, J., Cherifi, C.B., Moalla, N.: Towards software reuse through an enterprise architecture-based software capability profile. Enterprise Information Systems **16**(1), pp. 29–70 (2022). Taylor & Francis
5. Rolia, J., Cherkasova, L., Arlitt, M., Andrzejak, A.: A capacity management service for resource pools. In: Proceedings of the 5th International Workshop on Software and Performance, pp. 229–237 (2005)
6. Hendler, J., Wu, D., Sirin, E., Nau, D., Parsia, B.: Automatic web services composition using Shop2. In: Proceedings of The Second International Semantic Web Conference (ISWC) (2003)
7. Buehler, J., Pagnucco, M.: A framework for task planning in heterogeneous multi robot systems based on robot capabilities. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 28, no. 1 (2014)
8. Paulk, M.C., Curtis, B., Chrissis, M.B., Weber, C.V.: Capability maturity model, version 1.1. IEEE Software **10**(4), 18–27 (1993). IEEE
9. Bollinger, T., McGowan, C.: A critical look at software capability evaluations: An update. IEEE Software **26**(5), 80–83 (2009). IEEE
10. Apel, S., Lengauer, C.: Superimposition: A language-independent approach to software composition. In: International Conference on Software Composition, pp. 20–35 (2008). Springer
11. Apel, S., Kästner, C., Lengauer, C.: Language-independent and automated software composition: The FeatureHouse experience. IEEE Transactions on Software Engineering **39**(1), 63–79 (2011). IEEE
12. Linsbauer, L., Fischer, S., Michelon, G.K., Assunção, W.K.G., Grünbacher, P., Lopez-Herrejon, R.E., Egyed, A.: Systematic software reuse with automated extraction and composition for clone-and-own. In: Handbook of Re-Engineering Software Intensive Systems into Software Product Lines, pp. 379–404 (2022). Springer
13. Chrissis, M.B., Konrad, M., Shrum, S.: CMMI for development: guidelines for process integration and product improvement. Pearson Education (2011)
14. W3C. Retrieved July 9, 2024, from https://www.w3.org/OWL/
15. Neches, R., Fikes, R.E., Finin, T., Gruber, T., Patil, R., Senator, T., Swartout, W.R.: Enabling technology for knowledge sharing. *AI Magazine* **12**(3), 36–36 (1991).
16. Beimel, D., Peleg, M.: Using OWL and SWRL to represent and reason with situation-based access control policies. *Data & Knowledge Engineering* **70**(6), 596–615 (2011). Elsevier
17. Rahmouni, H.B., Solomonides, T., Casassa Mont, M., Shiu, S.: Modelling and enforcing privacy for medical data disclosure across Europe. *Medical Informatics in a United and Healthy Europe*, 695–699 (2009). IOS Press
18. Kayes, A.S.M., Rahayu, W., Dillon, T., Chang, E.: Accessing data from multiple sources through context-aware access control. *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, 551–559 (2018). IEEE
19. Jabardi, M., Hadi, A.S.: Twitter fake account detection and classification using ontological engineering and semantic web rule language. *Karbala International Journal of Modern Science* **6**(4), 8 (2020).
20. Dorri, A., Kanhere, S.S., Jurdak, R., Gauravaram, P.: Multi-agent systems: A survey. IEEE Access 6, 28573–28593 (2018)
21. Aué, Joop, Maurício Aniche, Maikel Lobbezoo, and Arie van Deursen. An exploratory study on faults in web API integration in a large-scale payment company. In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pp. 13–22, 2018.

22. SEBoK Editorial Board. 2024. The Guide to the Systems Engineering Body of Knowledge (SEBoK), v. 2.10, N. Hutchison (Editor in Chief). Hoboken, NJ: The Trustees of the Stevens Institute of Technology. Accessed 15 July 2024. www.sebokwiki.org. BKCASE is managed and maintained by the Stevens Institute of Technology Systems Engineering Research Center, the International Council on Systems Engineering, and the Institute of Electrical and Electronics Engineers Systems Council.