Modeling and Model Transformation as a Service: Towards an Agile Approach to Model-Driven Development

Adel Vahdati and Raman Ramsin(✉)

Department of Computer Engineering, Sharif University of Technology, Azadi Avenue, Tehran, Iran
vahdati@ce.sharif.edu, ramsin@sharif.edu

Abstract. Scalability has always been a challenge in software development, and agile methods have faced their own ordeal in this regard. The classic solution is to use modeling to manage the complexities of the system while facilitating intra-team and inter-team communication; however, agile methods tend to shy away from modeling to avoid its adverse effect on productivity. Model-driven development (MDD) has shown great potential for automatic code generation, thereby enhancing productivity, but the agile community seems unconvinced that this gain in productivity justifies the extra effort required for modeling. The challenge that the MDD community faces today is to incorporate MDD in agile development methodologies in such a way that agility is tangibly and convincingly preserved. In this paper, we address this challenge by using a service-oriented approach to modeling and model transformation that pays special attention to abiding by agile values and principles.

Keywords: Model-Driven Development · Agile methods · Service-oriented architecture

1 Introduction

In Model-Driven Development (MDD), models play the primary role throughout the process of software development [1]. One of the motivations for using this approach is to automatically create the product from models of the system. In model-driven development, the problem domain is described in terms of models at high levels of abstraction. By executing a chain of model-to-model transformations, the details of the solution domain are gradually added, thus producing refined models of the system. The process culminates in generation of code by using model-to-text transformation.

Agile methods are widely used in the software industry. Although they strive to expedite software development and delivery as much as possible, they also pay special attention to enhancing flexibility in order to respond to change in a timely manner [2]. Agile methodologies are lightweight and tend to shy away from modeling, as executable code is considered the main measure of progress; however, they all incorporate a highly-disciplined and well-defined process [1].

It might seem that agile development and model-driven development are poles apart. Agile methods are lightweight, fast, responsive and adaptable, while model-driven approaches are heavyweight and require early investment in modeling [3]. Agile methods focus more on the process and methodological aspects of software development, while model-driven approaches rely on architectural aspects and separation of concerns [4]. Nevertheless, it has been observed that by combining these two approaches, we can take advantage of the strengths of both and cover some of their weaknesses [3]. The goal of both approaches is to manage complexity: model-driven methods reduce accidental complexity by separating design concerns from implementation details [5]; agile methods manage complexity by creating product increments in short iterations and receiving early and fast feedback [6]. Both approaches also try to accelerate development and enhance response to change: agile methods move in this direction by early and continuous delivery of products in short iterations [4]; MDD achieves this goal by raising the level of automation in code generation [2].

There are several reasons for integrating agile methods and model-driven approaches, including [6]: improving agility and minimizing unnecessary tasks, increasing collaboration, enhancing requirements analysis, reducing risks by receiving early feedback, accelerating response to change, increasing the level of automation, managing complexity and building the models in an iterative-incremental fashion, and better understanding of the problem domain [6]. Some very influential agile methods started as model-phobic frameworks; however, it has since been realized that all of them can make use of modeling in some way [7]. One of the potential solutions is Agile Modeling (AM), which provides a means for adding modeling to agile methods without compromising agility [7]. The most important issue in agile MDD is to determine which agile practices, under what circumstances, and how, should be used in MDD [8]. In AM, models are created just in time and just enough for the specific purpose intended [9]. Despite its merits, AM's applicability to MDD should be further explored.

We propose a new agile approach to MDD by using service-oriented concepts. The purpose of this approach is to facilitate participation and collaboration in the modeling process and to improve scalability in terms of model size and the number of modelers involved in the modeling process. To this aim, we introduce the idea of "multilevel modeling as a service" and "model transformation as a service", and propose a new model-driven architecture. For modeling at different levels of abstraction, we propose the concept-based abstract syntax, which allows the description of the problem domain and the solution domain from structural, functional and behavioral perspectives.

The rest of this paper is structured as follows: Sect. 2 provides an overview of the previous works related to this research; Sect. 3 presents the problems currently afflicting agile software development and MDD, and the potential opportunities that will arise as a result of their integration; Sect. 4 describes different approaches for complexity management in modeling processes based on model decomposition; Sect. 5 describes our proposed approach for modeling and model transformation as a service; and Sect. 6 presents the conclusion and explains the next steps in this research.

2 Related Works

Agile development and MDD are both mature domains, and numerous efforts have been made over the years to integrate them. The works mentioned here are meant to provide a brief overview of the related literature.

Matinnejad [1] has evaluated a number of Agile Model-Driven Development (AMDD) methods as to their agility and MDD support. Essebaa and Chantit [4] have proposed a method for combining MDA and agile methods, and have examined how agile methods can benefit from MDA. Alfraihi and Lano [6, 12] have investigated the motivations and challenges of integrating agile development and MDD; lack of a well-defined process and tool support, and the steep learning curves involved, are recognized as the key challenges. Alfraihi and Lano [12] have also conducted a systematic literature review to examine the practices used in agile MDD. To facilitate sprint management in Scrum, Chantit and Essebaa [10] have combined Model-Driven Engineering (MDE) and Model-Based Testing (MBT) to produce a customized V-development life cycle that is integrated into Scrum. Bernaschina [11] has proposed an agile framework for rapid prototyping of model transformations. Asadi and Ramsin [13] have evaluated several MDA-based methods according to general, MDA-related, and tool-related criteria.

3 Integrating Model-Driven and Agile Development Approaches

A prerequisite for integrating agile and MDD methods into agile MDD processes is to be familiar with the strengths and weaknesses of these two areas. Also, the nature of the problems targeted by agile MDD is another issue that should be considered. In this section, we separately examine the deficiencies of agile development and MDD and investigate the challenges and opportunities facing the integration of these two areas.

3.1 Agile Development Challenges

Agile methods have come a long way as to their support for scalability, but scalability is still a serious problem, especially in large and complex projects involving distributed teams [14]. Over-reliance on face-to-face conversation as the sole means for conveying and information, and avoidance of modeling at all costs, can be detrimental to scalability. This poses a challenge to coordination and communication in distributed teams; lack of trust, common ground, language and knowledge base make it difficult for distributed teams to work together and develop a large and complex system [15].

Another problem with agile methods is their attitude towards architecture [2]. Agile approaches are risk-driven rather than architecture-driven, and even though modern, more mature agile methods such as Disciplined Agile Delivery (DAD) [9] pay special attention to architecture, most methods see the main goal as mitigating the risks rather than providing a reliable high-level structure that addresses the quality attributes. As a result, modeling and refining the architecture is not focused upon sufficiently, which in turn adversely affects scalability: it is difficult to assess the effects of architecturally significant design decisions on quality attributes; software evolution is difficult and tedious because the code is the only available means for learning and knowledge sharing; and there will be a steep learning curve for newcomers to the team.

3.2 MDD Challenges

MDD requires early planning, investment and design [3]. Typically, model-driven methodologies have heavyweight processes that have a negative effect on agility. Modeling large and complex systems in an iterative-incremental fashion, and collaboration among teammates during modeling activities in large distributed teams, are other challenges of MDD. The strongest motivation for using MDD is the continuous evolution of software technologies. In MDD, code can be (semi)automatically generated through a series of model transformations [16], but the main problem with this approach is that we need to prepare and take the initial steps before starting the development process [2]. This early investment can add value when the assets produced during these steps can be reused frequently. Therefore, production of reusable artifacts and responding to changes by demand is one of the challenges of MDD [2].

The model-driven architecture (MDA), which has become quite popular in MDD, is a good example of a multi-layered architecture. The three modeling levels of MDA (Computation-Independent Model or CIM, Platform-Independent Model or PIM, and Platform-Specific Model or PSM) enhance reusability through abstraction [16]. For instance, in PIM models, application specifications are platform-independent and ignore implementation technology issues. Therefore, it is possible to reuse these models for different implementation technologies [16].

Research in the field of MDA has focused more on the PIM and PSM levels, and little work has been done on CIM level models. UML, as a popular modeling language, is not suitable for displaying models at higher levels of abstraction such as CIM. Using domain-specific languages (DSL) can improve the expressiveness of the language for displaying models of a particular application domain [16], but in current MDE practices, the process of building DSLs is done on an ad-hoc basis [17].

3.3 Opportunities and Challenges of Integrating Agile and MDD Approaches

Models, as a common language and basis, facilitate communication and interaction between different teams and improve the scalability of agile methods [14]. Model-phobia in some prominent agile methods poses challenges to maintenance, evolution and change tracking. MDD strives to improve productivity by automatically generating code from models, and to provide sufficient detail to assist the maintenance phase by creating models at various abstraction levels. However, MDD is not inherently agile [18]. Therefore, we need to adhere to agile values and use best practices in agile modeling, along with the lessons learnt from hands-on experience and practical expertise in the field, in order to achieve effective agile modeling of software systems [19].

In Agile MDD, instead of modeling the whole system at once, models evolve continuously according to user demands [19]. Agile modeling is done gradually and in small steps, and instead of creating a large and complex model, several models are created and used in parallel. In modeling, unnecessary details are avoided and the focus is on the required aspects. During the modeling process, users are actively involved and simple tools are used to produce the models [6]. By storing artifacts in shared repositories and applying collaborative modeling techniques, communication and interaction between stakeholders is improved and existing artifacts can be reused [20].

One of the gaps in agile methods is the role of architecture in software solutions [14], which is well covered by the use of MDD. Typically, the technologies that are supposed to support a business change faster than the business itself [21]. MDD facilitates software evolution by separating the problem domain from the solution domain. By establishing a mapping between the problem domain and the solution domain, if the problem domain models change, these modifications are propagated to solution domain models through the model transformations and mapping between the two levels, but if the solution domain changes (by adopting a new technology or platform) we only need to modify the mapping (transformations) and the changes will not be propagated to the problem domain models [2].

Despite the opportunities available, integrating agile and model-driven approaches poses its own issues and problems. Most of the proposed methods lack a systematic and well-defined process, and teams usually proceed on an ad-hoc basis based on their experiences [6, 12]. Lack of appropriate tools to take advantage of Agile MDD and the steep learning curve that developers have to face are other problems hindering the integration of these two areas [6]. For example, CI/CD tools are key enablers of agile methodologies, and version control systems play an important role in this pipeline. Current version control systems manage and track changes and resolve conflicts at the code level. Therefore, they can only identify and resolve conflicts at the syntax level, and semantic conflicts caused by changes in modeling artifacts cannot be detected by these tools [20]. This is an interesting research topic, but it will not be addressed in this paper.

The agile approach prioritizes people and their interactions over processes and tools. However, having the right tools to facilitate the use of agile MDD plays an important role in fast product delivery and response to change. In addition to supporting modeling and testing, these tools should also support change and configuration management [6].

4 Complexity Management in Modeling

Software systems are complex in nature, and many are distributed as well. Different teams can be involved in the system development process, but the members of these teams are not necessarily co-located. The key question is how to manage the modeling complexity of the problem domain and improve collaboration in the modeling process. In MDD, the metamodel is first defined by identifying the domain concepts, which are then instantiated to yield the model elements. Accordingly, the model must conform to the syntactic rules and constraints defined in the metamodel. We use model decomposition for managing complexity in modeling processes, and propose three approaches based on the meta-level to which decomposition is applied. As shown in Fig. 1, each approach has its own benefits and liabilities when used in an agile MDD context.

In the first approach, a metamodel is defined for the entire domain and the problem domain is described in the form of a single model. The second approach, similar to the first approach, uses a single metamodel to define the concepts and rules of domain-specific modeling language, but manages model complexity by domain decomposition, breaking up the problem domain model into multiple partial models. The partial models describe different parts of the problem domain, but their modeling language is the same, and an overall model of the problem domain is obtainable by integrating these partial models. In the third approach, breaking up the problem domain takes place at both metamodel and model levels. Therefore, in order to describe the same aspect in different parts (contexts) of the problem, the context-specific metamodel of each part is first defined, and the problem domain is then described from that perspective with the help of models that conform to context-specific aspect-related metamodels.

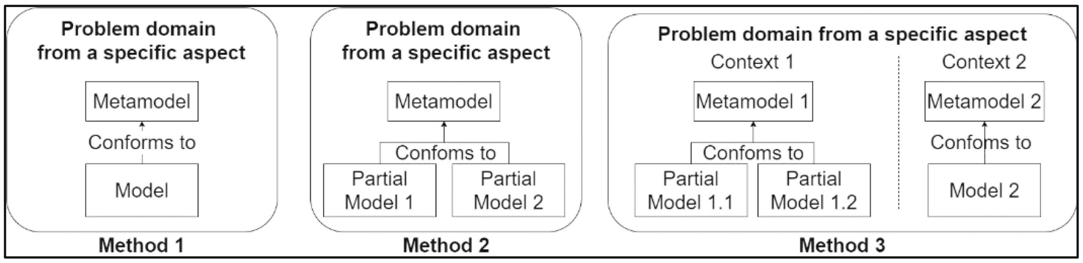


Fig. 1. Three approaches to metamodeling and modeling

4.1 First Approach

This approach is suitable for describing simple problems, but faces serious challenges for large and complex systems. From a scalability point of view, we encounter a large and complex metamodel that contains all the domain concepts and syntactic rules. Validation and maintenance are difficult as it is not possible to get early feedback from the user before defining a heavyweight metamodel. Reusability of the modeling artifacts is also low, as for each aspect, different contexts of the system are described in the form of a single model by using a single modeling language (single metamodel).

This approach lacks agility, and makes iterative-incremental modeling impossible. If the metamodel is modified, these changes should be reflected to a large and complex model, which makes it difficult to keep the model and the metamodel compatible. From the perspective of cooperation and collaborative modeling, this approach also faces various issues. Collaborative modeling requires breaking up the modeling tasks, but this approach lacks a clear strategy for this purpose.

It should be noted that in general, cooperation of team members in the modeling process can be done either synchronously or asynchronously. In synchronous collaboration, all members work on a single shared model, and if a part of the model is modified by a team member, the changes are communicated synchronously to all the members involved in the modeling process. This method usually uses locking mechanisms to maintain consistency. Each modeler must lock the model before making any changes, which interferes with the design process. Locking the elements of a large model and managing and releasing locks is an important problem of this method.

In asynchronous collaboration, each team member has a copy of the remote model and modifies the local version of the model, using version control systems to apply changes to the remote version of the model. Merging the local changes with the remote model is handled automatically in the absence of conflict, otherwise the conflicts must be resolved manually. Pulling all the elements of a large model from the remote repository and storing them locally is not efficient in terms of resource consumption.

4.2 Second Approach

The second approach uses a multitude of models to describe the problem domain. By identifying different areas (subdomains) of the problem domain, it is possible to describe each context consistently and unambiguously. In this approach, the same language (metamodel) is used for describing a specific aspect in different contexts of the problem domain. However, each context (subdomain) can have its own model and model repository, so in asynchronous collaborative modeling, it is not necessary to load all the specifications of the problem domain, but each team can load, describe and modify the specifications of the areas assigned to it as modeling tasks. Thus, subdomains can be the basis for division of modeling activities and task assignment among different teams. However, the operational cost of maintaining multiple repositories and the interdependencies between different subdomains, and integrating them to produce an overall view of the system, is the price that should be paid for reducing complexity, and improving scalability and collaboration.

Reusability at the metamodel level is similar to the first approach. However, if the problem domain is decomposed into cohesive parts with minimal interdependencies, the reusability of partial models will be improved. In this approach, we need to define a heavyweight metamodel before starting the modeling process of different subdomains. Also, making a change in the metamodel can affect the models of multiple subdomains. As a result, iterative-incremental development of models and metamodels becomes challenging and, from this perspective, lacks the necessary agility.

4.3 Third Approach

The third approach manages complexity at both the metamodel and model levels. To describe a specific aspect in different contexts (parts) of the problem domain, the modeling language (abstract syntax) appropriate for each context is created as a metamodel, and the problem domain is then described from that perspective (aspect) by using context-specific languages. In this approach, separation of concerns helps manage complexity. Also, instead of defining a large and complex metamodel for each aspect that considers all context-related concerns and details, several lightweight and context-specific metamodels are developed to describe the different contexts of the problem from that perspective by using different modeling languages (metamodels).

This improves the reusability and maintainability of the metamodel and related models: if one metamodel changes, we only need to maintain the compatibility of its corresponding models. It also allows for gradual and evolutionary modeling and contributes to the agility of the modeling process. By assigning the tasks related to the modeling and metamodeling of each context to a team, different teams can concurrently collaborate in the modeling process, thus enhancing collaborative modeling.

In this approach, the overall view of the problem domain from a specific perspective (aspect) is generated by integrating the partial models of different contexts of the problem domain, and model transformations play a key role in this regard. The complexity of integration is the cost that should be paid to improve cooperation, scalability and mutual independence of teams as to the modeling process. Identifying and distinguishing the different contexts of the problem domain can be challenging: if the logical boundaries between the different contexts of the problem are not well identified, integration will become difficult. There are several strategies for decomposing the metamodel. A coarse-grained metamodel can be decomposed by considering the following goals: increase the cohesion of fine-grained metamodels, form autonomous teams, and improve participation and cooperation in the modeling process. The Bounded Context pattern [22] can be used as a guideline and mechanism to decompose metamodels with respect to these goals.

Specialized fields usually have their own language and literature, which can be the basis for decomposing a coarse-grained metamodel into several fine-grained domain-specific metamodels, thus producing cohesive metamodel and models. The independence of teams in developing different parts of the software system may be the basis for deciding how to break up the metamodel. Reducing inter-team dependencies allows different teams to work in parallel. This improves agility and cooperation in modeling activities. In co-located teams, it is thus possible to exchange information effectively, build trust and promote collaboration. As a rule of thumb, the Bounded Context pattern can be used as a guideline and starting point for decomposing a coarse-grained metamodel. Later on, two context-specific metamodels can be merged and assigned to a single team according to other concerns and criteria, including: reducing inter-team dependencies, saving on integration/operational costs, and reducing the collaboration costs resulting from geographical distribution of the teams. All of these benefits make the third approach a wise choice for agile MDD.

5 Modeling as a Service and Model Transformation as a Service

As seen in the previous section, the third approach to management of modeling complexity improves agility and collaboration in the modeling process. However, the main challenge in this approach is to integrate partial models and provide a high-level view. Although partial models of different contexts describe the problem domain from the same aspect (e.g., the structural aspect), these models are heterogeneous because each partial model conforms to a different metamodel. Therefore, in order to achieve an overall view of the system from a specific aspect, we need to integrate these heterogeneous partial models. To address this problem, we must first determine the types of relationships that exist among the partial models.

5.1 Types of Relationships Between Models

Metamodel/model decomposition should be such that different contexts have the least interdependence. However, in practice, these contexts are not isolated from each other. For example, a model of infrastructure services can be shared and used by other contexts. But at times, the same service is remodeled to enhance team independence and strengthen control over service specification. Inspired by [22], we have identified four categories for classifying the natures and types of relationships between models: separate context, shared context, duplicate context, and conformist context.

Separate Context. The simplest situation is when two partial models have nothing to do with each other and do not need the information of the other model to describe their own domain. Under such circumstances, changes in each of these partial models are not disseminated to the other, and their integration would not provide more information than the pre-integration information.

Shared Context. In this case, part of the information is shared by two partial models and has the same specifications. Describing this shared context requires collaborative modeling (synchronous or asynchronous), and coordination between the teams responsible for each of the partial models. If all of these partial models are stored in a central repository, we will need access control mechanisms so that members of different teams can only access the shared part. However, if each team has its own repository, storing the shared context specifications in a separate, shared repository facilitates access control management and collaboration among team members. However, in this case, each team would need to manage two repositories (one private and one shared), and would also have to integrate the model specifications stored therein. In the Shared Context category, reusing models and avoiding redundancy is the main concern.

Duplicate Context. In this case, some of the information is shared by two partial models, but the burden of coordination and collaborative modeling between the two teams is such that sharing and collaborating in the modeling process (for sake of reuse) costs more than redefining and describing the shared context by each team. In this case, having autonomous teams has a higher priority than reusing artifacts. Although teams can still exchange information through Agile practices such as Scrum-of-Scrums, each team produces its own specifications for the shared context. The price that is paid for this level of flexibility is the possibility of creating semantic inconsistencies.

Conformist Context. In this case, one of the partial models (a downstream model) depends on the information of another model (an upstream model) and these models are defined and maintained by two different teams (supplier and consumer). The supplier has complete independence of action in making design decisions, but the design decisions made by the consumer must be aligned with and conform to the upstream model. Therefore, tracking changes in the upstream model and disseminating it to the downstream model is the responsibility of the consumer. For example, in MDA, CIM models provide information for PIM models, and the relationships between them are conformist. The driving forces behind the CIM models are the rules and constraints that govern the business domain, and PIM models are required to comply with these rules and restrictions, and the design models at the PIM level are in line with the business domain models at the CIM level.

5.2 Loosely Modeled Relationships

As the type and nature of the relationships among the partial models becomes clear, an important question that arises is how the relationships should be modeled in order to facilitate the integration process. Adherence to the two fundamental principles of "high cohesion" and "low coupling" seems to be a suitable strategy. Metamodel decomposition (through the third approach) should first be applied to maximize the cohesion of the conforming model, and in contrast, the relationships between concepts in different models should be loosely modeled in order to minimize coupling. Loosely modeled relationships between two models, or between their elements, promotes inter-team and intra-team collaboration.

Traditional modeling approaches model the relationships among the elements in a tightly coupled manner. Suppose a team of designers intend to model the structural aspects of a system in collaboration with each other in the form of class diagrams. Suppose that there are two classes called *Order* and *Customer* in this model, which are identified by two members of the team. There is an Association relationship between *Order* and *Customer*, but the Association relationship between them cannot be defined before defining the classes themselves. This will tie the design steps of the two team members together because modeling the relationship between the two elements is highly dependent on the presence of both at the moment of relationship definition.

A model can describe a situation, but to do so in the realm of modeling, we should not have to realize all aspects of the constituent elements of that situation. Designers usually model the system from the perspective of an outside observer, while the problem space can be viewed from the perspective of each of its constituent elements. In the previous example, *Order* describes the situation from its perspective as being related to the *Customer*, but the presence or absence of the *Customer* element at the moment of describing this situation does not change the reality of the problem; this is only a technical concern to consistently define the model.

Therefore, relationships should be modeled asynchronously and loosely. The cost of this approach is that the model may sometimes be inconsistent, but this type of inconsistency can be resolved by completing the modeling process. In the long run, it seems that improving flexibility, enhancing participation and collaboration, reducing dependency and increasing scalability outweigh the temporary inconsistency of the domain model. In this regard, we have introduced the idea of modeling and model transformation as a service in which the problem domain is described in terms of different domain concepts, each concept being embodied in the form of a service. This service makes it possible to define a concept from different perspectives. Each concept can be described from three perspectives: structural, functional and behavioral. It also provides essential functionalities required to query the structural, functional, and behavioral specifications of a concept.

The structural dimension expresses the characteristics and relationships of that concept with other concepts. The functional dimension focuses on the functionalities that a concept can provide. The behavioral dimension describes how this concept interacts with other concepts to fulfill its role. The behavioral specification of a concept is described from two perspectives: the responsibilities that the outside observer owes to that concept and the facts about that concept that the outside observer may be interested in knowing. This observer can be another domain concept, or the system as a whole.

In this approach, instead of modeling the problem domain only from the designer's point of view, we describe it from the perspective of each concept in the different contexts that make up the domain. The justifications behind this strategy are: information hiding, reducing unnecessary coupling, managing complexity through domain decomposition, and improving collaboration in the modeling process. For example, if *Order* has a unidirectional relationship with *Customer*, from *Order*'s point of view, there is a specific relationship with a concept called *Customer*, but *Customer* does not need to know anything about this relationship in the *customer management* context. In other words, *Customer* does not even know that a concept called *Order* is present in the problem domain because describing the problem domain from a *customer management* perspective does not require any such knowledge. In addition, *Order* does not need to know about all the *customer*-related attributes defined in the *customer management* context (e.g., date of birth). The loose connection between *Order* and *Customer* can be realized by using an event-driven architecture. *Order* states that it has a specific relationship (with certain name and attributes) with *Customer*. From an event-driven point of view, it can be interpreted that *Order* is interested in being informed when *Customer* is modeled, or if it already exists in the scope of the problem domain. When *Customer* is described in the problem domain, it announces the fact of being existent in the form of a published event. Using the Publisher/Subscriber pattern, it will be possible to model the relationships between two concepts in a loose manner.

This allows for asynchronous modeling, which improves participation and collaboration of team members in the modeling process. On the other hand, problem domain decomposition and describing it from the perspective of each domain concept allows complexity management at fine-grained levels. It is the modeler's responsibility to provide a macro view through the integration of these micro views. Since each domain concept is realized in the form of a self-contained service, it will be possible to provide a high-level view through the integration of services.

By identifying the subdomains, it is possible to model the system in the form of a hierarchical structure from coarse-grained to fine-grained. The problem domain resides at the highest level and consists of one or more subdomains, each of which contains relevant domain concepts. Each subdomain can be considered as a composite service that acts as a wrapper and includes services corresponding to domain concepts.

In other words, a subdomain provides access to these services from the outside world indirectly and is responsible for maintaining the consistency and transactional integrity of its internal concepts. The problem domain acts as a facade over the entire system and ensures consistency and transactional integrity at the system level. This hierarchical structure makes it possible to discover concepts (services) and resolve the relationships between them, similar to what happens in a DNS. The whole process can be done asynchronously by exchanging messages and using the publisher/subscriber pattern.

5.3 Model-Driven Development by Using Service-Oriented Paradigm

MDA is the main architecture adopted in MDD endeavors [23], and several methodologies have been proposed in its support [13]. MDA follows a layered structure (Fig. 2) while our proposed model-driven architecture has an onion structure (Fig. 3).

In current model-driven practices, modelers have focused on design models and usually start their work from the beginning by creating PIM-level models, completely ignoring the CIM-level. Lack of modeling at the CIM level can lead to various problems. Firstly, design models are affected by design decisions and solution issues, so they are less reusable than CIM-level analysis models; secondly, design models are not understandable by the end user, and domain experts cannot validate them; and thirdly, there is a semantic gap between the abstract high-level concepts used by domain experts and the abstract concepts used by modelers and designers, which is a major source of accidental complexity [20, 24]. Due to changes in requirements, bridging the gap manually is not cost-effective in terms of time and effort [24]. Lack of analysis models at the CIM level prevents the automatic production of PIM models based on CIM models. As a result, it is not possible to automatically publish changes in the requirements and analysis models to the design models.

In our proposed architecture, the problem domain is first described in terms of concepts and their interrelationships, and CIM-level models are defined. In line with the idea of "concept as a service", domain concepts are described from structural, functional and behavioral perspectives. A simplified version of the proposed abstract syntax (metamodel) for describing each domain concept is shown in Fig. 4.

Examining a concept from a structural perspective determines what properties that concept has and how it relates to other concepts in the problem domain. Examining a concept from a functional perspective aims to identify the functionalities that it can provide. Concepts in the real world usually need to interact and use the services provided by other concepts to fulfill their roles and tasks. In the behavioral dimension of a concept, the element of time and the sequence of interactions and communications between concepts play a key role. The behavioral specification of a concept expresses the dynamic aspect of that concept, while the structural and functional specifications describe its static aspects.

During the modeling process, the problem domain metamodel is first described by identifying the domain concepts, properties, and the relationships between them in a textual form based on the proposed abstract syntax. This metamodel includes the concepts, domains, and domain concepts of the problem domain. Then, by creating new instances of the domain concepts, domain objects are created to form the domain model.

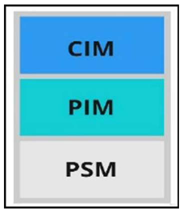


Fig. 2. OMG model-driven architecture

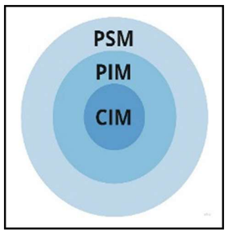


Fig. 3. Proposed model-driven architecture

One of the problems with conventional modeling approaches is that existing facilities for defining metamodels cannot be used at the model level as well. So if we need a new type, we have to define it explicitly at the metamodel level. To overcome this drawback, the notion of multilevel modeling was proposed, which allows in-depth definition of a language in more than two levels [25]. Two techniques have been proposed to extend the standard modeling approach: potency-based multilevel modeling [25] and Orthogonal Classification Architectures (OCA) [26, 27].

Potency-based multilevel modeling allows the domain to be described at multiple levels. In this method, the elements in the model have two facets at the same time: type and instance. For this reason, elements are called 'Clabjects', a combination of Class

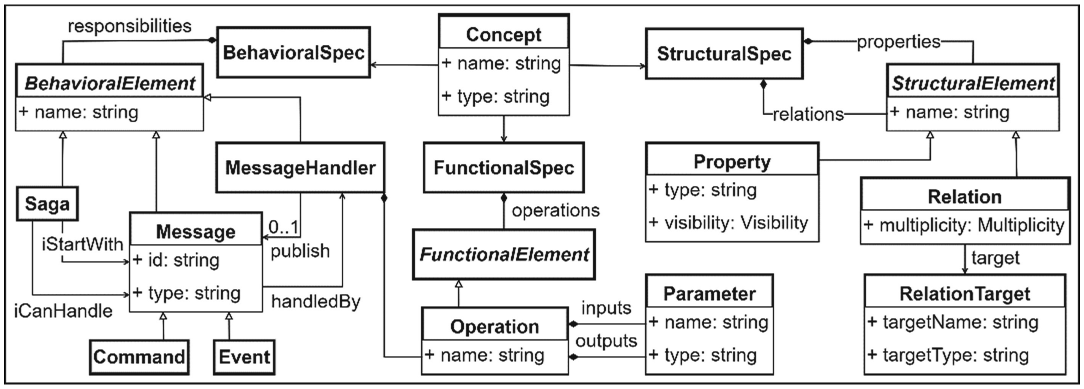


Fig. 4. Describing a concept from three perspectives

and Object that exhibits the characteristics of both. In OCA [26], two orthogonal typing systems are proposed, one based on ontology and the other based on linguistics [27].

In our "multilevel modeling as a service" idea, we extend OCA by adding a third dimension: relational. From an ontological perspective, the elements of the model are logically described as defined in the ontology hierarchy. From a linguistic point of view, the physical dimension of the elements is discussed, which refers to the concepts and structures that are necessary to construct and represent that element in models. The relational dimension focuses on the relationship between two elements of two different models. This dimension is embodied in our proposed solution in the form of an onion architecture: PIM-to-CIM relation, PSM-to-PIM relation, and Code-to-PSM relation.

In Fig. 5, an example of multilevel modeling from ontological (O0, O1, and O2) and linguistic (L0, L1 and L2) dimensions is shown. By analyzing the problem domain and exploring the subdomains ("Domain"s) and concepts, domain concepts are first constructed ("DomainConcept"s). Then, by creating new instances of the domain concepts, domain objects ("DomainObject"s) are created that actually form the domain model. A DomainObject has two facets: it is an instance of its ontological upper level Domain-Concept (e.g., in Fig. 5: *Film* is an instance of *Product*), and it can be considered as a template for instantiation of its ontological lower level, and thus play the role of a DomainConcept for the level below (e.g., in Fig. 5: *Film* is a type for *StarWars*).

Figure 6(b) shows the relational dimension of the proposed multilevel modeling approach. PIM-to-CIM, PSM-to-PIM and Code-to-PSM relations are loosely modeled. In line with the ideas of "modeling as a service" and "concept as a service", each layer provides access to its model elements and their descriptions to its higher layer, through services corresponding to these elements. Therefore, the loosely modeled relationship between the elements of each layer with its lower layer elements can be resolved using service discovery, service call, and the publisher/subscriber pattern.

In our approach (Fig. 6(b)), CIM-level models are created with three objectives:

1. Improve reusability by creating analysis models.
2. Improve understandability: Models at the CIM level are more understandable to the end user and domain experts, and in line with agile values, increase their collaboration and participation in the modeling and validation process.

3. Enable (semi)automated generation of PIM-level models by integrating CIM-level models with the design decisions and concerns described at the PIM level, without polluting CIM-level models with solution domain issues and implementation details.

To promote the participation and cooperation of domain experts and the development team in the modeling process, a common language is required. To this aim, we have introduced a method called CRAC (standing for "Concept-Responsibilities-Asynchronous Collaboration"), which aims to explore the problem space and reach a common language (Ubiquitous Language [22]) between domain experts and the development team. We will further explain this method in the next section. The CRAC analysis model thus produced is transformed into a concept-based model and a set of corresponding structural, functional and behavioral aspects, which constitute the CIM-level models and are represented in the form of self-contained and self-descriptive services at the CIM level. PIM-level services can obtain the specifications of a concept from different perspectives by calling specific concept-related services at the CIM level.

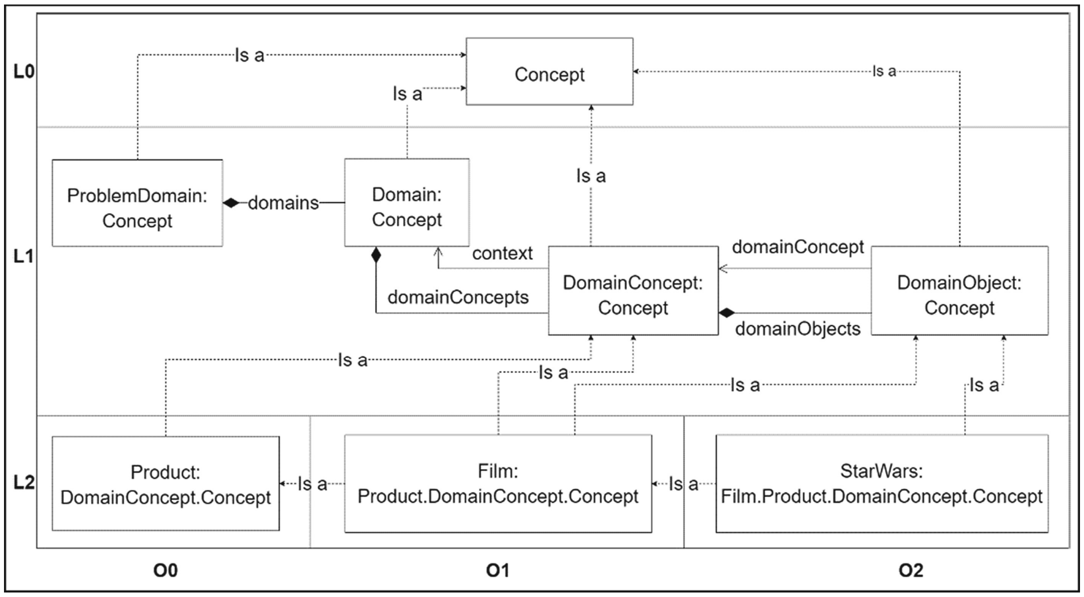


Fig. 5. An example of multilevel modeling from linguistic and ontological perspectives

Next, we enter the realm of the solution domain. To do this, we need to describe design details and solution concerns. However, these specifications do not directly apply to CIM-level models, but are rather expressed using the specific language of the PIM layer (Design-level DomainConcepts). If the partial models described at the PIM level require the specifications of CIM-level concepts, they refer to the concepts and models described at the CIM level (via the relational dimension of the proposed multilevel modeling approach) without the need to redefine this information at the PIM level. Integrating design details at the PIM level with the specifications of CIM-level concepts is done automatically using the built-in or user-defined model transformation services.

The idea of "model transformation as a service" is based on an event-driven architecture that allows reactive model transformations. In other words, model transformations

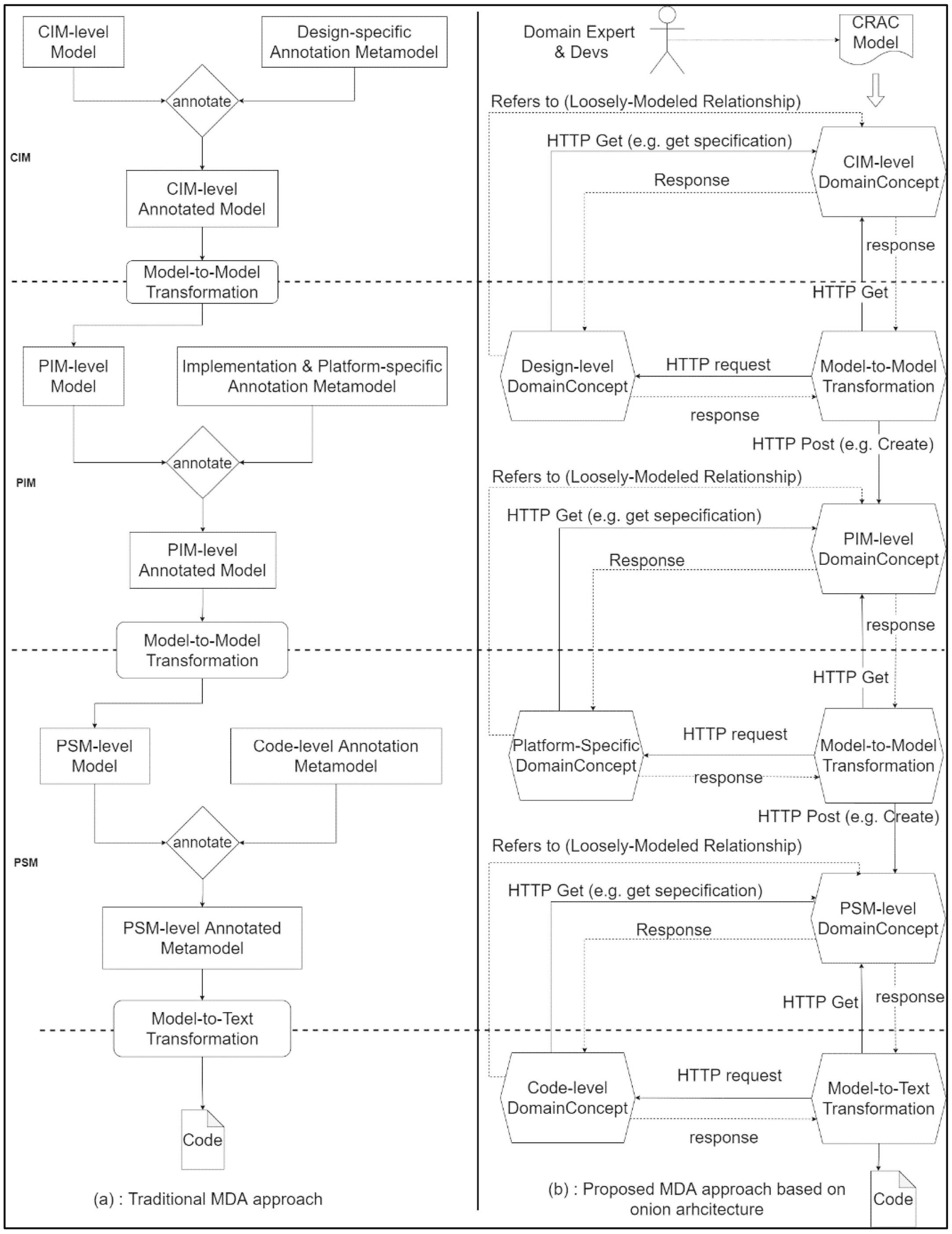

CIM-level Model
Design-specific Annotation Metamodel
annotate
CIM
CIM-level Annotated Model
Model-to-Model Transformation
PIM-level Model
Implementation & Platform-specific Annotation Metamodel
annotate
PIM
PIM-level Annotated Model
Model-to-Model Transformation
PSM-level Model
Code-level Annotation Metamodel
annotate
PSM
PSM-level Annotated Metamodel
Model-to-Text Transformation
Code
(a) : Traditional MDA approach
Domain Expert & Devs
CRAC Model
Refers to (Loosely-Modeled Relationship)
HTTP Get (e.g. get specification)
Response
CIM-level DomainConcept
response
HTTP Get
Design-level DomainConcept
HTTP request
Model-to-Model Transformation
response
HTTP Post (e.g. Create)
Refers to (Loosely-Modeled Relationship)
HTTP Get (e.g. get sepecification)
Response
PIM-level DomainConcept
response
HTTP Get
Platform-Specific DomainConcept
HTTP request
Model-to-Model Transformation
response
HTTP Post (e.g. Create)
Refers to (Loosely-Modeled Relationship)
HTTP Get (e.g. get sepecification)
Response
PSM-level DomainConcept
HTTP Get
response
Code-level DomainConcept
HTTP request
Model-to-Text Transformation
response
Code
(b) : Proposed MDA approach based on onion arhcitecture

Fig. 6. Traditional MDA vs. proposed MDA approach

can be performed not only at the user's request, but also in response to the creation or modification of domain objects. One of the main components of a model transformation is the source model. When a model transformation service is defined, it declares that it is interested in receiving events related to the domain objects that correspond to its source model (Fig. 7 – A.3: Subscribe to the event based on source model). By creating or modifying a domain object, a relevant event is published (Fig. 7 – B.3: Publish an event) and delivered to the model transformation services that subscribe to the event (Fig. 7 – C.1: Handle the event). Upon receiving this event, the model transformation service initiates the process of de-serializing the event and extracting the domain object identifier, retrieving domain object specifications from corresponding services (Fig. 7 – C.2), performing transformation steps (Fig. 7 – C.3), and creating one or more target domain objects conforming to the destination model (Fig. 7 – C.4).

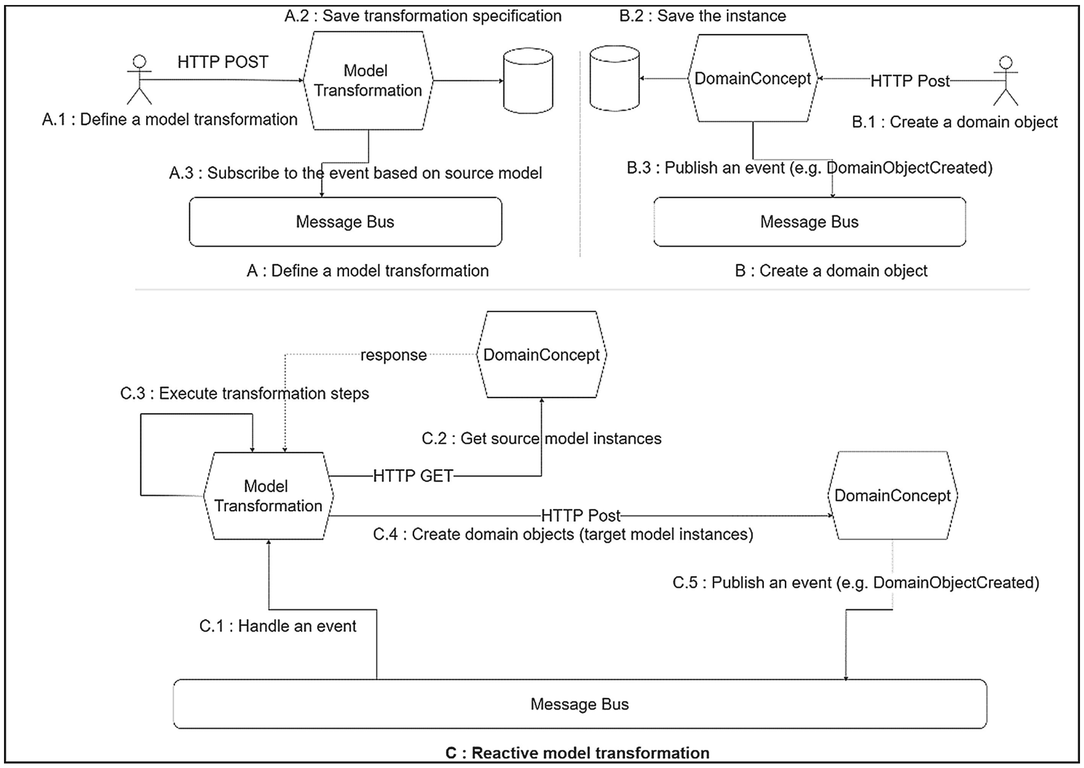

A.2 : Save transformation specification
B.2 : Save the instance
HTTP POST
Model Transformation
DomainConcept
HTTP Post
A.1 : Define a model transformation
B.1 : Create a domain object
A.3 : Subscribe to the event based on source model
B.3 : Publish an event (e.g. DomainObjectCreated)
Message Bus
Message Bus
A : Define a model transformation
B : Create a domain object
response
DomainConcept
C.3 : Execute transformation steps
C.2 : Get source model instances
HTTP GET
Model Transformation
DomainConcept
HTTP Post
C.4 : Create domain objects (target model instances)
C.5 : Publish an event (e.g. DomainObjectCreated)
C.1 : Handle an event
Message Bus
C : Reactive model transformation

Fig. 7. High-level architecture of reactive model transformation.

In current model-driven practices (Fig. 6(a)), CIM-level model elements are tagged to provide design details that are not present in the CIM-level models, and model transformations use these annotations to conduct transformation steps and produce PIM-level models. This approach contaminates the analysis model with design concerns and reduces its readability and expressiveness. If design decisions change, we will need to re-annotate the analysis model. While in our proposed approach, CIM models remain intact and are not corrupted by solution domain issues. Rather, these details are described separately using PIM-level domain concepts, and are automatically combined with CIM models to generate PIM models. The same scenario exists between the Code, PSM, and PIM layer models, as shown in Fig. 6(b).

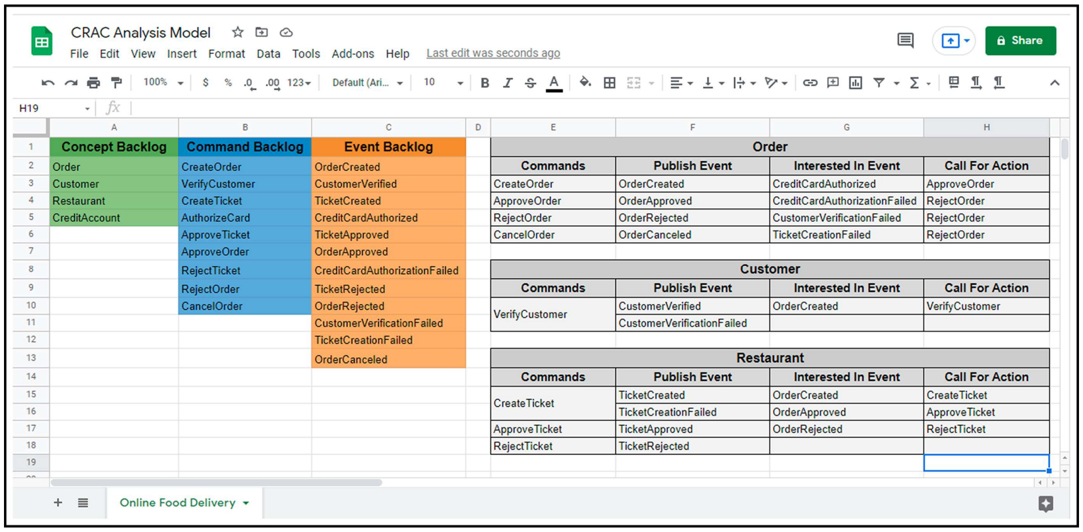
5.4 CRAC Method

In this method, we first identify the concepts of the problem domain. Each concept plays a role in the problem space, and other concepts in this context have expectations of it that can be interpreted and expressed in terms of its responsibilities. A concept's responsibilities can be "accomplished" or "failed", and it is possible to deduce a set of facts or events that explain this situation. For example, when a responsibility is successfully performed, it can be inferred that the pre-conditions and post-conditions associated with that responsibility have been met.

Concepts can also be interested in a set of facts and events in order to fulfill their responsibilities. They can also react to an event when being informed about a fact. This information is described in the form of a model consisting of these elements: Concepts, Commands that are executed by a concept, Events that are published as a result of command executions, Events that a concept is interested to know about it, and Commands that are executed in reaction to the events of interest. For each concept, these four pieces of information can be inserted on both sides of a card called a CRAC card (*Commands, Publish Event, Interested in Event,* and *Call for Action* columns).

In order to improve collaboration between domain experts and the development team, a Google Spreadsheet can be used to describe and access this information simultaneously. Figure 8 and Fig. 9 show the partial analysis model of an online food ordering system produced by the CRAC method. The system must be able to receive orders (*'CreateOrder')*. If an order is submitted successfully, the *'OrderCreated'* event will be published. *'Restaurant'* is interested in *'OrderCreated'* events. When this event occurs, it asks the kitchen to issue a ticket (*'CreateTicket'*) for the order; the kitchen can accept this order and issue a receipt (*'TicketCreated')*, or not issue it due to running out of food (*'TicketCreationFailed'*). The identity of the owner of the order should be verified when the order is created; customer identity may be approved (*'CustomerVerified')* or rejected (*'CustomerVerificationFailed'*). When the customer is verified and the order receipt is issued by the restaurant, the customer's credit card should be checked; at this stage, the card may be approved (*'CreditCardAuthorized')* or rejected (*'CreditCardAuthorizationFailed'). 'Order'* is interested in these events to confirm or reject the order: if the *'CreditCardAuthorized'* event occurs, *'Order'* invokes the *'ApproveOrder'* command and the *'OrderApproved'* event is published as a result; otherwise, it invokes *'RejectOrder'* and the *'OrderRejected'* event is published.

Similarly, *'Restaurant'* needs to know if the order is approved or not: if the order is approved, *'Restaurant'* approves the issued receipt by invoking the *'ApproveTicket'* command, which will result in the publication of the *'TicketApproved'* event; but if the order is rejected, *'Restaurant'* rejects the ticket by invoking the *'RejectTicket'* command, and *'TicketRejected'* will be published as a result. Other requirements of the online food delivery system can be analyzed and modeled in the same fashion, but this is not our goal in this paper. As shown in this example, the CRAC method can help better understand the problem domain and express business rules and processes through a chain of commands and events. This modeling approach is understandable to domain experts and end users,


CRAC Analysis Model
File Edit View Insert Format Data Tools Add-ons Help
Last edit was seconds ago
Share
Online Food Delivery

Concept Backlog	Command Backlog	Event Backlog
Order	CreateOrder	OrderCreated
Customer	VerifyCustomer	CustomerVerified
Restaurant	CreateTicket	TicketCreated
CreditAccount	AuthorizeCard	CreditCardAuthorized
	ApproveTicket	TicketApproved
	ApproveOrder	OrderApproved
	RejectTicket	CreditCardAuthorizationFailed
	RejectOrder	TicketRejected
	CancelOrder	OrderRejected
		CustomerVerificationFailed
		TicketCreationFailed
		OrderCanceled

Order			
Commands	Publish Event	Interested In Event	Call For Action
CreateOrder	OrderCreated	CreditCardAuthorized	ApproveOrder
ApproveOrder	OrderApproved	CreditCardAuthorizationFailed	RejectOrder
RejectOrder	OrderRejected	CustomerVerificationFailed	RejectOrder
CancelOrder	OrderCanceled	TicketCreationFailed	RejectOrder

Customer			
Commands	Publish Event	Interested In Event	Call For Action
VerifyCustomer	CustomerVerified	OrderCreated	VerifyCustomer
	CustomerVerificationFailed		

Restaurant			
Commands	Publish Event	Interested In Event	Call For Action
CreateTicket	TicketCreated	OrderCreated	CreateTicket
	TicketCreationFailed	OrderApproved	ApproveTicket
ApproveTicket	TicketApproved	OrderRejected	RejectTicket
RejectTicket	TicketRejected		

Fig. 8. CRAC analysis model of an online food ordering system

and the terms used for naming the concepts, commands, and events are parts of a language that is common among the development team(s), domain experts and end users.

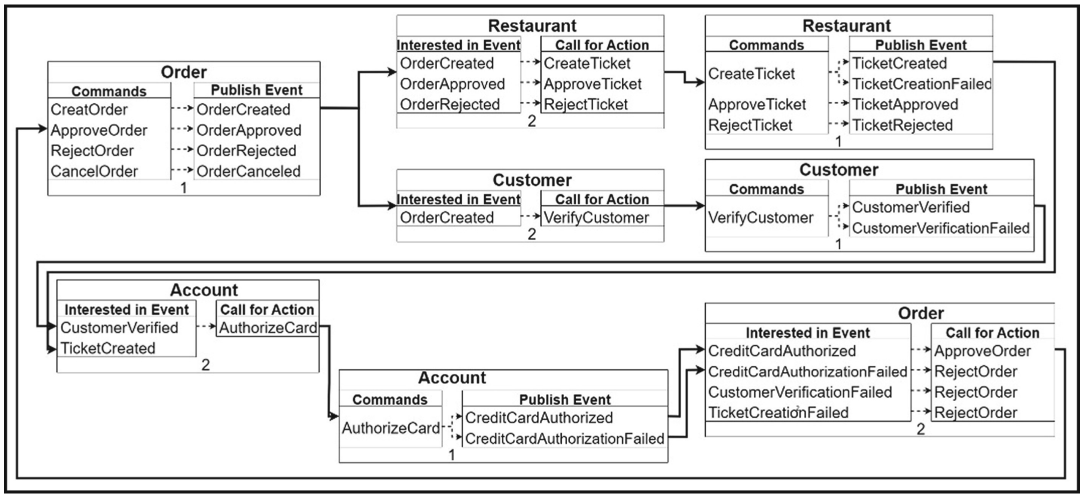


Fig. 9. Analysis model: CRAC cards

6 Conclusion and Future Work

Our preliminary analysis of the proposed approach shows that the idea of modeling and model transformation as a service is in line with the values, principles and best practices of agile modeling. Describing the problem domain in terms of concepts and modeling the relationships among these concepts in a loose manner facilitates collaboration throughout the modeling process and improves scalability in terms of the number of modelers involved. Model decomposition enhances complexity management and addresses scalability challenges in terms of artifact size. It also allows for iterative-incremental modeling and can enhance agility due to loosely-modeled relationships, the onion architecture, multilevel modeling as a service, and reactive model transformation.

The proposed CRAC method fosters mutual and shared understanding between domain experts and development team members, and facilitates collaboration and user involvement in the modeling process at the CIM level. The tool used for modeling at this level is simple and understandable to non-technical users.

In our proposed approach, production of high-level models from lower-level models enhances the reusability of modeling artifacts. Therefore, at each level of modeling, one can focus only on the specific concerns of that level. Realizing the idea of “model transformation as a service” in the form of an event-driven architecture makes it possible to automatically propagate the changes occurring in lower-level models to higher-level ones. Moreover, by integrating and composing fine-grained model transformation services, it is possible to execute reactive model transformations concurrently or as chains.

Applying a service-oriented approach in modeling and model transformation allows for the use of different patterns and architecture styles such as the microservice architecture. Examining the two areas of service-orientation and modeling, and establishing a semantic correspondence between the issues and challenges of these two fields will be one of our future research activities. This will help us apply the patterns and techniques used in the service-oriented paradigm to solve the problems and challenges of model-driven development. Providing a model-driven development platform (MDDPlatform) by using the service-oriented approach has also been planned as a future activity. The goal of MDDPlatform would be to support all the functionalities required to fully realize the ideas of modeling and model transformation as a service.

References

1. Matinnejad, R.: Agile model driven development: an intelligent compromise. In: International SERA Conference, pp. 197–202 (2011)
2. Wegener, H.: Agility in model-driven software development? Implications for organization, process, and architecture. In: OOPSLA Workshop on Generative Techniques in the Context of Model Driven Architecture, vol. 23 (2002)
3. Whittle, J.: Agile versus MDE - friend or foe? In: Workshop on Extreme Modeling, vol. 1089 (2013)
4. Essebaa, I., Chantit, S.: Model driven architecture and agile methodologies: reflexion and discussion of their combination. In: Federated Conference on Computer Science and Information Systems, pp. 939–948 (2018)
5. Mahé, V., Combemale, B., Cadavid, J.: Crossing model driven engineering and agility. In: Workshop on Model-Driven Tool and Process Integration (2010)
6. Alfraihi, H., Lano, K.: Practical aspects of the integration of agile development and model-driven development: an exploratory study. In: Flexible MDE Workshop, pp. 399–404 (2017)
7. Ambler, S.W.: Agile modeling: a brief overview. In: Workshop of the pUML Group, pp. 7–11 (2001). https://dl.gi.de/20.500.12116/30849
8. Zhang, Y., Patel, S.: Agile model-driven development in practice. IEEE Softw. **28**(2), 84–91 (2011)

9. Ambler, S.W., Lines, M.: Choose your WoW: a disciplined agile delivery handbook for optimizing your way of working. Project Management Institute (2020)
10. Chantit, S., Essebaa, I.: Towards an automatic model-based Scrum methodology. Procedia Comput. Sci. **184**, 797–802 (2021)
11. Bernaschina, C.: ALMOsT.js: an agile model to model and model to text transformation framework. In: Cabot, J., De Virgilio, R., Torlone, R. (eds.) ICWE 2017. LNCS, vol. 10360, pp. 79–97. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60131-1_5
12. Alfraihi, H., Lano, K.C.: The integration of agile development and model driven development: a systematic literature review. In: International Conference on Model-Driven Engineering and Software Development, pp. 451–458 (2017)
13. Asadi, M., Ramsin, R.: MDA-based methodologies: an analytical survey. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 419–431. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69100-6_30
14. Mognon, F., C. Stadzisz, P.: Modeling in agile software development: a systematic literature review. In: Silva da Silva, T., Estácio, B., Kroll, J., Mantovani Fontana, R. (eds.) WBMA 2016. CCIS, vol. 680, pp. 50–59. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-55907-0_5
15. Jolak, R., Wortmann, A., Chaudron, M., Rumpe, B.: Does distance still matter? Revisiting collaborative distributed software design. IEEE Softw. **35**(6), 40–47 (2018)
16. Sebastián, G., Gallud, J.A., Tesoriero, R.: Code generation using model driven architecture: a systematic mapping study. J. Comput. Lang. **56**, 100935 (2020)
17. Kolovos, D., et al.: MONDO: scalable modelling and model management on the cloud. In: CEUR Workshop, pp. 44–53 (2015)
18. da Silva, E., Maciel, R., Magalhães, A.: Integrating model-driven development practices into agile process: analyzing and evaluating software evolution aspects. In: International Conference on Enterprise Information Systems, pp. 101–110 (2020)
19. Schonbock, J., Etzlstorfer, J., Kapsammer, E., Kusel, A., Retschitzegger, W., Schwinger, W.: Model-driven co-evolution for agile development. In: Hawaii International Conference on System Sciences, pp. 5094–5103 (2015)
20. Alam, O., Corley, J., Masson, C., Syriani, E.: Challenges for reuse in collaborative modeling environments. In: MODELS Workshops, pp. 277–283 (2018)
21. Uhl, A.: MDA is ready for prime time. IEEE Softw. **20**(5), 70–72 (2003)
22. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Longman, Boston (2003)
23. da Silva, A.R.: Model-driven engineering: a survey supported by the unified conceptual model. Comput. Lang. Syst. Struct. **43**, 139–155 (2015)
24. Combemale, B., Deantoni, J., Baudry, B., France, R., Jézéquel, J.-M., Gray, J.: Globalizing modeling languages. Computer **47**, 68–71 (2014)
25. Atkinson, C., Kühne, T.: The essence of multilevel metamodeling. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 19–33. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45441-1_3
26. Atkinson, C., Kennel, B., Goß, B.: The level-agnostic modeling language. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 266–275. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19440-5_16
27. De Lara, J., Guerra, E., Cuadrado, J.S.: When and how to use multilevel modelling. ACM Trans. Softw. Eng. Methodol. **24**(2), 1–46 (2014)