Domain-Driven Design Representation of Monolith Candidate Decompositions

Miguel Levezinho[1][0000−0003−0995−2094], Stefan Kapferer[2][0009−0007−1097−7965], Olaf Zimmermann[2], and António Rito Silva[1][0000−0001−9840−457X]

[1] INESC-ID, University of Lisbon Instituto Superior Técnico, Lisbon, Portugal
{miguel.levezinho,rito.silva}@tecnico.ulisboa.pt
[2] OST Eastern Switzerland University of Applied Sciences, Rapperswil, Switzerland
{stefan.kapferer,olaf.zimmermann}@ost.ch

Abstract. Microservice architectures have gained popularity as one of the preferred architectural approaches to develop large-scale systems. Similarly, strategic Domain-Driven Design (DDD) gained traction as the preferred architectural design approach for the development of microservices. However, DDD and its strategic patterns are open-ended by design, leading to a gap between the concepts of DDD and the design of microservices. This gap is especially evident in migration tools that identify microservices from monoliths, where candidate decompositions into microservices provide little in terms of DDD refactoring and visualization. This paper proposes a solution to this problem by extending the operational pipeline of a multi-strategy microservice identification tool, called Mono2Micro, with a DDD modeling tool that provides a language, called Context Mapper DSL (CML), for formalizing the most relevant DDD concepts. The results are validated with a case study by comparing the candidate decompositions resulting from a real-world monolith application with and without CML translation.

Keywords: Domain-Driven Design · Microservices · Migration.

1 Introduction

Microservice architectures have become one of the architectures of choice for emerging large enterprise applications [23, 6]. This adoption results from the advantages of partitioning a large system into several independent services, which provide qualities such as strong boundaries between services; independent development, testing, deployment, and scaling of each service; and service-tailored infrastructures [19, 10, 27]. On the other hand, topics such as how to distribute the system and the consistency model might stagger the design early on. The use of a monolith architecture, where the business logic of the system is interconnected, has the advantage that it does not require early modularization. The neat identification of modules occurs through refactorings, after initial development, which allows one to explore the application domain first [11].

Therefore, it is common practice to start with a monolith and, as the system grows in size and complexity, migrate to a more modular architectural approach, such as a modular monolith [12] or a microservice architecture. Since this architectural migration is not trivial [25], recent research has proposed approaches and tools to help the migration process [1, 2].

This has led to the development of Mono2Micro, a modular and extensible tool for the identification of microservices in a monolith system [20]. Mono2Micro focuses on identifying transactional contexts to inform its generated candidate decompositions [22]. To this end, it integrates several approaches, such as static code analysis of monolith accesses to domain entities [29], dynamic analysis of monolith execution logs [4], lexical analysis of abstract syntactic trees of monolith methods [8], and analysis of the history of monolith development [21]. Furthermore, Mono2Micro supports a set of measures and graph views to evaluate the quality of the generated candidate decompositions [28].

However, as with most research on the identification of microservices in monolith systems, Mono2Micro does not allow software architects to further model generated candidate decompositions using Domain-Driven Design (DDD) [7], which has shown good results on microservice design [33] and growing interest in the industry [34]. Instead, Mono2Micro representations of candidate decompositions are based on sequences of read and write accesses to the monolith domain entities, which are difficult to work with when trying to redesign the original monolith system and its functionalities for a modular architecture.

This paper addresses this problem by providing a representation of the Mono2Micro candidate decompositions in terms of automatically generated tactical and strategic DDD patterns. In this way, software architects can work on candidate decompositions from the perspective of DDD.

This is achieved by extending the operational pipeline of Mono2Micro with a connection to Context Mapper, a DDD-focused modeling tool that provides a Domain-Specific Language, named Context Mapper DSL (CML). CML supports the declarative description of DDD domain models, using DDD concepts as building blocks of the language [16]. With this goal in mind, the following research questions are raised:

- **RQ1**: How can current approaches to the identification of microservices in monolith systems be extended to include DDD?
- **RQ2**: Can the results of a candidate decomposition based on entity accesses be represented in terms of DDD?
- **RQ3**: Can an architect benefit from the use of a tool that integrates DDD when analyzing and working on a candidate decomposition?

To answer these research questions, a real monolith system was used as a case study. The resulting candidate decompositions of this system were generated with and without the new DDD modeling capabilities and then compared.

The remainder of this paper is structured as follows. Section 2 goes over the current literature on DDD application and microservice identification tools. Section 3 gives some background on the Mono2Micro and Context Mapper tools. Section 4 presents the solution to the aforementioned research questions. Section 5 provides the validation of the solution with a case study application, and in Section 6 the results and answers to the research questions are discussed. Finally, Section 7 concludes the paper.

2 Related Work

The application of DDD in microservice development, although widely practiced, is still poorly formulated [30], the focus being in terms of modeling tools that leverage tactic and strategic DDD patterns [34].

Most research extends existing standards to convey DDD concepts. They use annotated constructs, such as in [26], where a mapping from DDD to UML is presented with the use of annotations inside UML class constructs, or in [18], where an annotation-based DSL was developed to scope objects and attributes within the concepts of DDD. However, they do not support all DDD patterns, especially strategic ones such as *Bounded Context* relationships, which are useful when modeling microservices from candidate decompositions.

Context Mapper is an exception to this, providing a DSL to model tactic and strategic DDD patterns [16], rapid model prototyping by deriving *Domains* and *Bounded Contexts* from use case definitions [17], and integration with other technologies such as Microservice Domain-Specific Language (MDSL) [15].

Other research also explores the extensibility of DDD to better fit other stages of software development. In [13] they define *Domain Views*, which enable different stakeholders to perceive the domain model with their respective knowledge base. The Context Mapper tool also provides *Domain Views* through the definition of types of *Bounded Context* and *Context Maps* [16].

On the other hand, there has been extensive and recent research on tools for the identification of microservices in monolith systems [1]. However, these tools, such as Mono2Micro [20], do not provide output that enables DDD-based editing and modeling, they mostly provide decompositions that are service-oriented and not domain-oriented.

To our knowledge, the only tool that supports the reverse engineering of DDD concepts is the Discovery Library tool [16]. It produces domain models from Spring Boot[3] service APIs using discovery strategies. Through code analysis, it finds specific Spring Boot annotations and maps them to the corresponding DDD concepts.

3 Background

To better inform the integration of Context Mapper into the Mono2Micro pipeline, this section gives an overview of the architecture of both tools and compares them.

[3] https://spring.io/projects/spring-boot

3.1 Mono2Micro

Mono2Micro is a migration tool that provides candidate monolith decompositions composed of clusters of domain classes. This work initially focused on the identification of microservices driven by the identification of transactional contexts [22], but other strategies have been added [4, 8, 21].

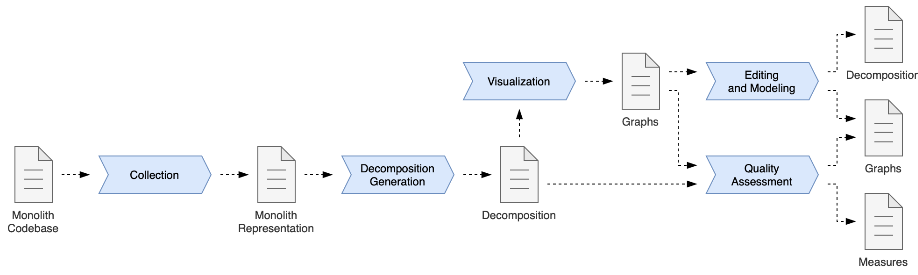

Fig. 1. The five stages of the Mono2Micro operational pipeline [20]. Each stage can use the output of former stages as input.

Mono2Micro is designed as a pipeline, which is represented in Figure 1. The five stages of the pipeline are:

1. *Collection*: Implements several static and dynamic code collection strategies to represent monoliths, including representations based on accesses to source code domain entities, functionality logs, and commit history and authors.
2. *Decomposition Generation*: Partitions the monolith domain entities into clusters using a set of similarity criteria, with a focus on producing good quality decompositions.
3. *Quality Assessment*: Compares the decompositions and calculates the measures that are used to evaluate the generated decompositions. The measures include coupling, cohesion, size, and complexity.
4. *Visualization*: Depicts decompositions in the form of graphs with multiple levels of detail. Nodes and edges can represent different elements, depending on the chosen collection strategy.
5. *Editing and Modeling*: Provides an interface with operations to modify the automatically generated decompositions so that the architect can refine them. Quality measures are also automatically recalculated, if applicable.

Each stage is composed of one or more modules that output artifacts for the next stage in the pipeline. The underlying model of the tool that makes up these modules and artifacts is also built with several extension points, making it possible to support multiple decomposition strategies.

However, this pipeline does not include any way for an architect to model candidate decompositions using DDD after the *Decomposition Generation* stage.

More concretely, in the *Visualization* stage, graph representations of the decomposition include cluster-based views of the decomposition domain entities, and functionality-based views that represent its sequence of accesses to domain entities ("Graphs" in Figure 1). There are no DDD-based views that show the model of each candidate microservice. Likewise, the *Editing and Modeling* stage does not contain any operations related to the application of DDD. This is where Context Mapper comes in.

3.2 Context Mapper

Context Mapper is a modeling framework that provides a DSL to design systems using DDD concepts. This DSL, henceforth called Context Mapper DSL (CML), was developed to unify the many patterns of DDD and their invariants in a concise language [16]. Figure 2 shows an example of the CML syntax, with the declaration of a *Context Map* containing two *Bounded Contexts*.

```
 1 ContextMap InsuranceContextMap {
 2    contains CustomerManagement
 3    contains CustomerSelfService
 4
 5    CustomerManagement [U] -> [D] CustomerSelfService
 6 }
 7
 8 BoundedContext CustomerManagement {
 9    Aggregate Customers {
10       Entity Customer {
11          aggregateRoot
12
13          - List<Address> addresses
14          String name
15       }
16
17       Entity Address {
18          String city
19          int postalCode
20       }
21    }
22
23    Application {
24       Service CustomersService {
25          void createCustomer(String name)
26          @Customer getCustomer(String name)
27       }
28    }
29 }
30
31 BoundedContext CustomerSelfService {
32 }
```

Fig. 2. Example syntax of CML, containing the syntax for defining a *Context Map* (1-6); *Bounded Contexts* (8-29, 31-32); *Aggregates* (9-21); *Entities* (10-15,17-20); and *Services* (24-27).

Within *Bounded Contexts*, one can define *Aggregates*, which consist of a group of closely related domain objects that form a unit for the purpose of data consistency. This consistency is enforced inside the *Aggregate* by its root *Entity*, which represents the only entry point. For example, in Figure 2 the `Customers` aggregate has the `Customer` entity as its root.

Although DDD focuses on the *Domain Layer* of systems, where the business logic is residing, a CML `Bounded Context` can also represent the *Application Layer*, which manages services that call different parts of the system, including processes in other layers. Using the `Application` keyword, *Application Services* can be defined, among other constructs, and contain operations like `createCustomer` and `getCustomer` as represented in Figure 2.

In addition to the CML language, Context Mapper also contains other utilities to facilitate modeling activities. These include the following:

1. *Discovery Library*: Implements several strategies to reverse engineer source code artifacts and represent them in CML [14].
2. *Architectural Refactoring*: Includes operations to refactor and transform CML code for easier modeling.
3. *Diagram Generators*: Provide translators to visualize CML artifacts in diagram form, such as UML representations of *Bounded Contexts* and BPMN maps of *Aggregate* states.

Each of these features has similarities with the features in Mono2Micro. First, the Discovery Library performs a similar job as the Collectors of Mono2Micro, but more importantly, it provides a way to generate CML from its input. Second, the Architectural Refactoring (AR) module supports the architect on the edition and modeling of CML models, as the Editing and Modeling stage of Mono2Micro. However, AR operations are built on DDD concepts. Finally, the Diagram Generators module can provide ways to view a candidate decomposition from the perspective of DDD, also something missing in Mono2Micro, which presents decompositions as a graph of clustered domain entities. It also includes generation of service contracts in the Microservice Domain Specific Language (MDSL), which is another DSL for specifying microservices, and that can lead to direct code generation for Open API, gRPC, Jolie, GraphQL, and plain Java.

4 Solution Architecture

The proposed extension to the Mono2Micro microservice identification pipeline provides a representation of candidate decompositions in CML, so that DDD can be used for modeling and refactoring activities. Figure 3 shows this extension in terms of modules and their input and output artifacts. The top process bar represents the relevant stages of the Mono2Micro pipeline, and the different colors separate existing modules from new ones. The following sections, each corresponding to one of the research questions, explain each module and artifact in more detail.

4.1 Tool Integration

Mono2Micro and Context Mapper are built with an emphasis on modularity and extensibility. This makes it viable for Context Mapper to integrate into the Mono2Micro pipeline. However, it is still important to respect the models of each tool to avoid compromising their internal cohesion. In practice, this meant pursuing a low-coupling solution when connecting the tools. This solution was achieved by leveraging on the Discovery Library (DL).

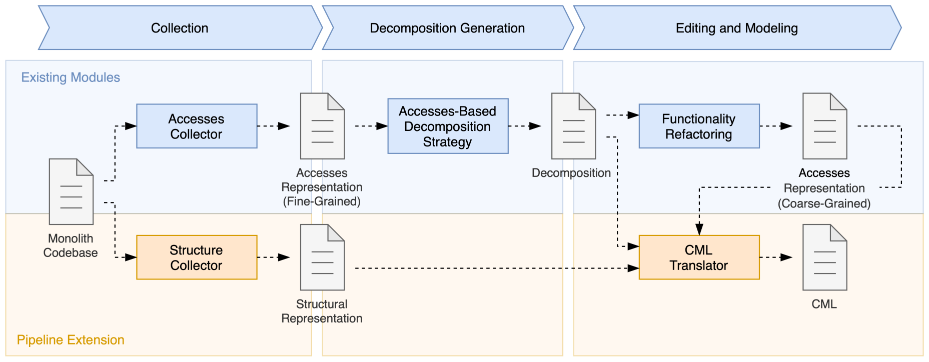

Fig. 3. Mono2Micro pipeline extension to support CML representation of candidate decompositions. In blue, the relevant pipeline steps (top) and modules. In orange, the extension to the pipeline, composed of the addition of new modules.

As described in Section 3, the DL is a standalone tool capable of generating CML code. This is done using discovery strategies that translate input into CML. Since the DL was designed to be highly extensible, it also provides an API for the creation of these discovery strategies. Using this API, the Mono2Micro pipeline was extended with a module that defines new discovery strategies capable of translating candidate decompositions into CML. This module is represented by the *CML Translator* in Figure 3.

The *CML Translator* has two stages. In the first, the internal representations of a decomposition in the Mono2Micro model are used to create a JSON contract that contains all the information needed to map a candidate decomposition to CML. This contract serves as input for the new discovery strategies and adds a layer of decoupling between the Mono2Micro model and the DL model, ensuring that changes made to the former do not inadvertently propagate to the latter. In the second stage, the new discovery strategies translate the contract to an internal representation of CML in the DL model. This model is, in turn, automatically converted to actual CML code.

4.2 DDD Mapping

For the new discovery strategies to perform the translation to CML, the concepts that form a candidate decomposition must be mapped to the DDD concepts first. Since DDD and its concepts are structural in nature [7], a candidate decomposition was also structurally defined, based on its internal representation in Mono2Micro. A candidate decomposition is composed of three key concepts: **entities**, which represent domain classes in the monolith; **clusters**, which represent a set of entities grouped by similarity criteria through a clustering algorithm; and **functionalities**, which represent sequences of read/write accesses to entities in one or more clusters. Mapping a candidate decomposition to DDD corresponds to mapping these three concepts and understanding what information is needed from Mono2Micro once a DDD concept is chosen. Figure 4 shows a summary of the achieved mappings, which are discussed in the next paragraphs.

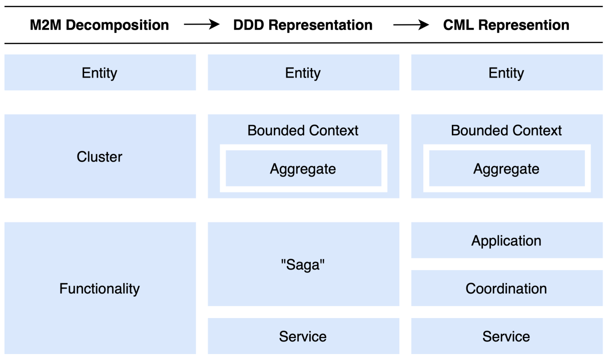

Fig. 4. Mapping strategy of candidate decomposition concepts from Mono2Micro (M2M) to DDD and CML.

Entity Mapping The entities of a candidate decomposition, by definition, are already based on the concept of *Entity* from DDD, which facilitates this mapping. The main difference is that Mono2Micro does not require the internal structure of entities to generate candidate decompositions, while in DDD and CML the internal state and relationships with other entities are relevant information to model an *Entity*. To guarantee a more complete translation of candidate decomposition entities into CML, a new source code collector module was added to the Mono2Micro Collection stage, aptly named *Structure Collector* as shown in Figure 3. This module uses the Spoon Framework library [24] to analyze and collect structural information from entities in the monolith domain, including entity names, entity attributes, and relationships between entities, i.e. composition or inheritance.

Cluster Mapping The main criteria that dictate how entities are clustered in the Mono2Micro Decomposition stage are based on transactional similarity. This means entities commonly accessed together (i.e. read/write) during the same transactions are more likely to belong in the same cluster. Similarly, a DDD *Aggregate* is defined as a group of tightly coupled domain objects that can be seen as a unit for the purpose of data changes during transactions, which makes it a good fit to represent a cluster. However, the concept of cluster also fits the concept of a DDD *Bounded Context*. This is because clusters define physical boundaries between microservices in a candidate decomposition and can be evaluated based on coupling with other clusters in the same decomposition. This dual mapping of the cluster concept could be achieved with different variations in the number of generated *Bounded Contexts* and *Aggregates*, but in the end the chosen mapping was to take each cluster and generate a corresponding *Bounded Context* and single *Aggregate* inside it, which in turn contains all the entities in the cluster. This does not mean that the end product is to have one *Aggregate* per *Bounded Context*. It is important to remember that the generated CML code is by no means final and that further refactoring is expected by the architect doing the modeling. Starting from this initial mapping that satisfies the definition of a cluster, architects have the ability to further refine the model by partitioning the generated *Aggregate* of each *Bounded Context* using not only entity access information, but also the new structural context of entities that is not available in Mono2Micro. Additionally, since entities that share structural relationships in the monolith can likely end up in different clusters after decomposition, the *Context Map* definition is updated with a relationship between *Bounded Contexts* in the direction of referenced *Entities*. This reference is also replaced with a reference to a newly created local *Entity*, which represents the outer *Entity* locally, so that the architect can better visualize which references need to be refactored. This case is shown in Figure 5.

```
Aggregate Tournaments {

  /* This entity was created to reference the 'Question' entity of the
   * 'Questions' aggregate. */
  Entity Question_Reference

  Entity Topic {
    String name
    - Set<Question_Reference> questions
  }
}
```

Fig. 5. Generated CML example, representing an *Aggregate* that contains 2 *Entities*. Since `Topic` referenced an *Entity* in its fields not present in the *Aggregate*, `Question_Reference` was generated locally to replace this reference.

Functionality Mapping Functionalities are more challenging to represent in DDD since each functionality is composed of a sequence of read and write accesses to entities, which is a concept very particular to Mono2Micro and without apparent DDD equivalent concept. Additionally, the sequence of accesses that represents a functionality can be quite extensive. The reason for this is the fine-grained nature of the accesses collected from monolith code, due to their object-oriented design. This contrasts with the coarse-grained communication that is expected between microservices to avoid distribution communication costs. Without resolving this granularity issue, it becomes very impractical to represent functionalities compactly. Fortunately, Mono2Micro provides a Functionality Refactoring tool that rewrites the functionalities of a candidate decomposition as Sagas [3, 5]. The tool converts several fine-grained microservice invocations into some coarse-grained ones, and is represented in Figure 3 as part of the pipeline. Refactoring functionalities as Sagas also makes a possible map to DDD more adequate. Although the Saga pattern is not a DDD pattern, in practice it can be used in conjunction with DDD to model distributed transactions [9]. To supply a construct for the representation of Sagas meeting the current requirements, an expansion to the CML syntax was proposed and implemented in Context Mapper, which allows for the definition of distributed workflows without specifying the communication model of the process. For the current functionality mapping use case, this new concept can be used to simply state the steps of the saga, without any implementing technology commitments. This construct is called *Coordination*, and is based on the coordination property of Sagas that specifies whether the steps of a Saga are orchestrated or choreographed [9].In CML, *Coordinations* can be used to coordinate defined *Service* operations, the same way a Saga coordinates steps. Figure 6 shows an example of the syntax in CML. *Coordinations* are defined within the *Application* layer of a *Bounded Context*. To reference a *Service* operation, a coordination step is divided into three segments, separated by the `::` symbol: The name of the Bounded Context where the operation is defined; the name of the application *Service* where the operation is defined; and the name of the operation. Functionalities that do not access other *Bounded Contexts* are simply mapped to a *Service*, also defined in the *Application* layer of the *Bounded Context* where they are defined.

4.3 CML Representation and Interaction

When using Mono2Micro, architects now have the option to convert candidate decompositions to CML using the translation strategy mentioned so far. Like all CML discovery strategies, the initial representation of the candidate decomposition in CML is not final. Further refactoring is expected. However, an effort was made to automatically create a good starting point. The names of entities, clusters, and functionalities from the initial decomposition are maintained and used for naming *Entities*, *Aggregates*, *Bounded Contexts*, and *Coordinations* in CML. The conversion of the format of functionalities to Sagas also creates additional constructs, in the form of *Service* operation calls, which correspond to *Coordination* steps in CML. These operations make up the interface of each *Bounded Context* in CML, but there is no straightforward name that can be used to name each operation. As such, several access-based naming heuristics were implemented in the translation strategy. The architect can further customize the level of detail they want the name to have regarding access information: **Full Access Trace** transcribes the entire ordered entity access sequence that happens within an operation into the name of that operation; **Ignore Access Types** omits the type of access to entities in operation names, i.e. read/write, replacing it with a `"ac"` prefix; **Ignore Access Order** omits the type and order of access to entities in the operation names. Each heuristic used reduces the number of generated operations, at the cost of entity access details. In its most reduced form, each operation name shows which entities are accessed in that step. Access information is also added to each translated entity in the form of a comment, showing metrics related to the percentage of external and local accesses to the entity in comparison with the total external and local accesses to the *Bounded Context*. In contrast to these heuristics, there is also the option to generate generic names for operations that are not access-based.

```
1 BoundedContext Tournament {
2    Application {
3       Coordination UpdateTournament_Coordination {
4          Tournament :: Tournament_Service :: updateTournament_step0;
5          Quiz :: Quiz_Service :: updateTournament_step1;
6          Tournament :: Tournament_Service :: updateTournament_step2;
7       }
8    } ...
9 } ...
```

Fig. 6. *Coordination* construct in CML. The steps of the *Coordination* (4-6) represent ordered calls to *Service* operations (10,20,11).

5 Case Study

Quizzes-Tutor (QT)[4] is an online quizzes management application developed for educational institutions. It can be used to create, manage, and evaluate quizzes composed of varying types of question formats. Teachers can add questions related to topics of the courses they preside over, while students can answer these questions within quizzes. Other functionalities include the creation of quiz tournaments between students, question proposals from students, and ways to discuss question answers. This real-world monolith, composed of 46 domain entities and 107 functionalities, was used as a case study to validate the Mono2Micro pipeline extension, which provides DDD modeling capabilities.

5.1 Decomposition Generation

To start the validation, a candidate decomposition for the QT application must be generated and chosen. To this end, around 2000 candidate decompositions were generated with different values of similarity criteria and number of clusters. Candidate decompositions were then filtered on the basis of the values of their measures. The heuristic used was to order decompositions based on their coupling value in ascending order, and then based on their cohesion value in descending order, to prioritize decompositions with low coupling and high cohesion. Of the top 100 results, the candidate decomposition with the lowest complexity value was chosen. Figure 7 also shows the clusters view of the decomposition shown in Mono2Micro. Data for this candidate decomposition can be seen in Table 1, with two noteworthy pieces of information.

To start with, the complexity of each cluster is very high. The complexity measure represents the migration cost of the functionalities in a cluster. This migration cost is measured as the cost of re-designing from an ACID context to a distributed one. Of the initial 107 functionalities, 31 involve distributed calls

[4] https://quizzes-tutor.tecnico.ulisboa.pt/

Table 1. Candidate decomposition measures for the QT case study.

Cluster	Entities	Functionalities	Cohesion	Coupling	Complexity
Cluster0	6	7	0.81	0.185	787.571
Cluster1	27	107	0.212	0.657	106.832
Cluster2	4	11	0.727	0.179	431.091
Cluster3	9	35	0.654	0.753	322.486

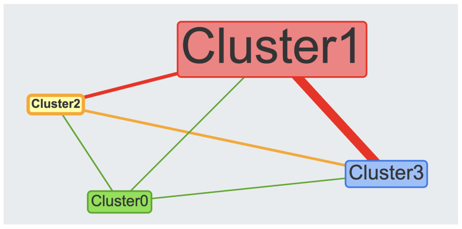

Fig. 7. Mono2Micro decomposition visualization with fine-grained interaction between clusters. Edges represent functionalities shared between clusters.

composed of several hops between clusters that drive the complexity high. This is because functionalities are still represented by the fine-grained monolith interactions between entities that can now be in different clusters. To reduce this complexity, the *Functionality Refactoring* tool represented in Figure 3 is used to create coarse-grained interactions between clusters. Table 2 shows the reduction of invocations for some of the QT functionalities. Applying this complexity reduction also makes it viable for functionalities to be represented in a structural language such as CML. Otherwise, any translation strategy would culminate in thousands of operation definitions for just a subset of the functionalities, as the FGI (Fine-Grained Interaction) values show in Table 2. The other noteworthy piece of information is the high number of entities inside Cluster1 compared to the other clusters, which means that the entities inside this cluster are more entangled when it comes to the functionalities that use them, and are more difficult to separate without creating an overly complex decomposition. It is also the reason for the non-optimal levels of cohesion and coupling in this cluster. At this stage, when some manual refactoring of functionalities is needed, modeling using DDD can help.

Table 2. Refactored functionalities for QT case study. CGI stands for Coarse-Grained Interaction, and FGI stands for Fine-Grained Interactions.

Name	#Clusters	CGI	FGI	Reduction%
concludeQuiz	3	4	73	94.52
getQuizByCode	3	4	33	87.88
getQuizAnswers	4	8	84	90.48
exportCourseExecutionInfo	4	9	110	91.82
importAll	3	5	119	95.8
createQuestion	2	3	24	87.5
getQuizAnswers	4	8	92	91.30

The candidate decomposition is translated into CML by the discovery strategies, which outputs a `.cml` file with a representation of the candidate decomposition. Figure 8 shows a modified snippet of the generated CML, related to the `ConcludeQuiz` functionality of a decomposition. Without any optimization, the translation generates a total of 121 operations, used by 31 Coordinations that represent the distributed functionalities. With naming heuristics, the number of service calls can be reduced to 87 using *Full Access Trace*, to 84 using the *Ignore Access Types*, and to 48 using the *Ignore Access Order*. Table 3 shows the reduction of generated services according to which heuristics are used per cluster. Regarding entity generation, a total of 11 reference entities were generated to signal structural dependencies between entities, also shown in Table 3, and every entity is generated with information on the number of accesses to it, from the total *Bounded Context* accesses (external and local), as shown in Figure 8. This information can help to refactor the decomposition further.

```
BoundedContext Cluster3 {
   Application {
      Coordination ConcludeQuiz_Coordination {
         Cluster3 :: Cluster3_Service :: acQuestionDetails_acOption;
         Cluster1 :: Cluster1_Service :: acQuiz_acQuizAnswer_acQuestion;
         Cluster0 :: Cluster0_Service :: acAnswerDetails;
         Cluster1 :: Cluster1_Service :: acStudent_acDashboard;
      }
      Service Cluster3_Service {
         void acQuestionDetails_acOption;
      }
   }
   Aggregate Cluster3 {
      /*
       * Metrics:
       * - Percentage of external accesses: 16.46% (13/79)
       * - Percentage of local accesses: 16.41% (21/128) */
      Entity QuestionDetails {
         Integer id
         - Question_Reference question
      }
      /* This entity was created to reference the 'Question' entity of the
       * 'Cluster1' aggregate. */
      Entity Question_Reference
   }
}
BoundedContext Cluster1 {
   Application {
      Service Cluster1_Service {
         void acQuiz_acQuizAnswer_acQuestion;
         void acStudent_acDashboard;
      }
   }
   Aggregate Cluster1 {
      /*
       * Metrics:
       * - Percentage of external accesses: 10.06% (33/328)
       * - Percentage of local accesses: 8.37% (41/490) */
      Entity Question {
         Integer id
         String title
         String content
         - Course course
         - Set<Topic> topics
      }
   }
}
BoundedContext Cluster0 { ... }
```

Fig. 8. Snippet of the generated CML related to the functionality `ConcludeQuiz`. Service operation names were truncated.

6 Discussion

This section discusses the findings of applying the DDD-based extension to the operational pipeline of Mono2Micro by analyzing how the implemented solution

Table 3. Generated CML constructs. The number of services is represented by four values: No heuristics used; *Full Access Trace* used; *Ignore Access Types* used; and *Ignore Access Order* used. The number of entities is represented by two values: original entities and reference entities. The most accessed entity is based on external accesses to the *Bounded Context*.

Cluster	#Services	#Entities	Most Accessed Entity
Cluster0	15/7/6/4	6/4	QuizAnswerItem (35.14%)
Cluster1	59/53/52/34	27/3	Quiz (12.2%)
Cluster2	11/5/5/2	4/0	QuestionAnswerItem (30.0%)
Cluster3	36/22/21/8	9/4	QuestionDetails/Image (16.46%)

and the results of its application in the case study answer the research questions raised in the introduction of this paper.

6.1 Results Validation

Starting with the first research question (RQ1), to evaluate whether the Mono2Micro operational pipeline can be extended to integrate DDD, through the use of CML, the first step taken was to measure the level of modularity and extensibility of the solution. First, modularity deals with how divided a system is into logical modules, improving separation of concerns and internal cohesion. The solution is composed of two new modules in Mono2Micro, the *Structure Collector* and *CML Translator*. In terms of cohesion, both modules respect the pipeline architecture of Mono2Micro, and are placed accordingly inside it based on their responsibilities. Second, extensibility deals with how open for extension the features of a system are without putting at risk their core structure, improving the addition of new functionality. The *Structure Collector* was designed from scratch. It provides abstractions for the collection of data from new frameworks and other types of structural data. The *CML Translator* is an extension of the DL API, so it follows that the discovery strategies implemented have the same design and are also open to extension by providing abstractions.

Moving on to the second research question (RQ2), the mappings of the cluster, entity, and functionality concepts demonstrate how a candidate decomposition can be represented with DDD concepts. Mono2Micro entities are already based on the concept of DDD *Entities*, so the mapping is consistent in this regard. In the case of clusters, consistency was maintained by mapping each of them to a *Bounded Context* and an *Aggregate*. For the mapping of functionalities, the sequence of accesses to entities that composed them was first converted into a structured Saga. This significantly reduced the complexity of the sequence in terms of size and hops between clusters, and made it simpler to represent with DDD. Sagas were mapped to *Coordinations* in CML, which encode an ordered sequence of service calls, just as Sagas encode a sequence of steps.

Finally, in regard to the third research question (RQ3), the case study shows how an architect can benefit from the use of this extension. In the case of entity representation, it is possible to observe the attributes of each entity and also the structural refactorings that must be made in existing entities. In the case of clusters, *Aggregates* can now be defined and used to further partition a cluster and its entities based on access patterns, access percentages, and the structural information provided at generation time. In the case of functionalities, the architect now has the option of editing their Saga representation in CML, by editing the generated *Coordinations*. Mono2Micro only allowed the creation of fine-grained functionality traces, without any way to edit or visualize them in a graphical representation. Using CML, these functionalities can be modeled as *Coordinations* and edited in the language. Furthermore, *Coordinations* can be visualized in BPMN, and the service naming heuristics allow the architect to reduce the number of generated service calls. This does not reduce the number of functionality steps but increases the level of reuse of services.

6.2 Practical Relevance and Adoption

The problem addressed with the presented approach is of high relevance to practitioners, mainly software engineers and architects, who need to modernize existing monolith applications. Such monolith applications without an inner modular structure, a.k.a. *Big Ball of Mud* [32], turned out to be a huge challenge in practice for several reasons, which can also be seen as use cases for our solution presented in this paper:

- **Economic reasons**: Maintaining a *Big Ball of Mud* is often becoming expensive for software companies. Changing such applications, adding new features, or fixing bugs, often takes too much time because of the intertwined code base and dependencies within the system.
- **Scalability and "Cloud readiness"**: Many companies have to decompose their applications to migrate to the cloud. The monolithic architecture approach is not scalable and does not fit the requirements for cloud deployment.
- **Autonomous teams and "DevOps"**: Many companies aim to implement agile development approaches in which teams develop and operate their part of an application autonomously [31]. A team should be able to make its own design and architectural decisions. This requires loosely coupled (micro-)services or at least a system with loosely coupled modules. The structure of the organization (teams) defines the architecture of the software.

As already mentioned, the adoption of DDD for service decomposition is widespread in the software industry. Both our tools, Mono2Micro[5] as well as Context Mapper[6] are open-sourced and, at least individually, have already gained some adoption in industry and real-world projects. The proposed approach is therefore foreseen as an important contribution and support for practitioners who want to: use a tool that automatically suggests decompositions for an existing monolith system; and want to express their future architecture and service

[5] https://github.com/socialsoftware/mono2micro
[6] https://github.com/ContextMapper

decomposition in terms of DDD patterns and follow the "domain-driven" approach. Once a CML model is available, practitioners can benefit from all Context Mapper features: iterative and agile modeling, architectural refactorings, model visualization (diagram generators), or even code generation.

6.3 Threats to Validity

With respect to internal validity, the functionalities used in the decomposition and CML mapping process are all linear in nature. This is due to the existent static entity access collection tool in Mono2Micro, which flattens code branches into a single access sequence in a depth-first fashion. However, previous research that used the same sequences to develop the Saga representation of functionalities has shown this has little impact on the final results [5], and support for multiple traces per functionality is being developed.

In terms of external validity, the current implementation assumes the use of Java and the Spring Boot JPA Framework to collect entity access and structure information, but the process is general enough to be applicable to other programming languages and frameworks. The modules that assume these limitations are also built with abstractions for the implementation of other technologies.

7 Conclusion

This paper proposes a solution pipeline for the lack of DDD in migration tools, composed of the integration of Context Mapper, a modeling framework that provides a DSL to represent DDD patterns, into the Mono2Micro decomposition pipeline, a robust microservice identification tool.

The proposed solution achieves the integration by defining a mapping of concepts between tools, whilst respecting each of the tool models. To support this mapping, the solution includes several new modules and modifications, including a new static collector of entity structural information, a contract for effective communication between the tools, a translation strategy to generate CML from Mono2Micro decompositions, i.e. entities, clusters, and functionalities, an extension to the CML syntax to support concepts from decomposition in the form of *Coordinations*, and new diagram generators from CML based on translated decompositions.

The artifacts developed in the project are publicly[7] available together with the description of the procedures necessary to use them.

This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) through projects UIDB/50021/2020 (INESC-ID) and PTDC/CCI-COM/2156/2021 (DACOMICO)

[7] https://github.com/socialsoftware/mono2micro/tree/master/tools/cml-converter

References

1. Abdellatif, M., Shatnawi, A., Mili, H., Moha, N., Boussaidi, G.E., Hecht, G., Privat, J., Guéhéneuc, Y.G.: A taxonomy of service identification approaches for legacy software systems modernization. Journal of Systems and Software **173**, 110868 (2021)
2. Abgaz, Y., McCarren, A., Elger, P., Solan, D., Lapuz, N., Bivol, M., Jackson, G., Yilmaz, M., Buckley, J., Clarke, P.: Decomposition of monolith applications into microservices architectures: A systematic review. IEEE Transactions on Software Engineering **49**(8), 4213–4242 (2023). https://doi.org/10.1109/TSE.2023.3287297
3. Almeida, J.F., Silva, A.R.: Monolith migration complexity tuning through the application of microservices patterns. In: Software Architecture. pp. 39–54. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-58923-3_3
4. Andrade, B., Santos, S., Silva, A.R.: A comparison of static and dynamic analysis to identify microservices in monolith systems. In: Tekinerdogan, B., Trubiani, C., Tibermacine, C., Scandurra, P., Cuesta, C.E. (eds.) Software Architecture. pp. 354–361. Springer Nature Switzerland, Cham (2023)
5. Correia, J., Rito Silva, A.: Identification of monolith functionality refactorings for microservices migration. Software: Practice and Experience **52**(12), 2664–2683 (2022). https://doi.org/10.1002/spe.3141
6. Di Francesco, P., Lago, P., Malavolta, I.: Migrating towards microservice architectures: An industrial survey. In: 2018 IEEE International Conference on Software Architecture (ICSA). pp. 29–2909 (2018). https://doi.org/10.1109/ICSA.2018.00012
7. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison Wesley (2003)
8. Faria, V., Silva, A.R.: Code vectorization and sequence of accesses strategies for monolith microservices identification. In: Garrigós, I., Murillo Rodríguez, J.M., Wimmer, M. (eds.) Web Engineering. pp. 19–33. Springer Nature Switzerland, Cham (2023)
9. Ford, N., Richards, M., Sadalage, P., Dehghani, Z.: Software Architecture: The Hard Parts. O'Reilly Media, Inc. (2021)
10. Fowler, M.: Microservice trade-offs (2015), https://martinfowler.com/articles/microservice-trade-offs.html
11. Fowler, M.: Monolith first (2015), https://martinfowler.com/bliki/MonolithFirst.html
12. Haywood, D.: In defence of the monolith (2017), https://www.infoq.com/minibooks/emag-microservices-monoliths/
13. Hippchen, B., Giessler, P., Steinegger, R., Schneider, M., Abeck, S.: Designing microservice-based applications by using a domain-driven design approach. International Journal on Advances in Software (1942-2628) **10**, 432 – 445 (12 2017)
14. Kapferer, S.: A Modeling Framework for Strategic Domain-driven Design and Service Decomposition. Master's thesis, University of Applied Sciences of Eastern Switzerland (2020). https://doi.org/10.13140/RG.2.2.22950.68167
15. Kapferer, S., Zimmermann, O.: Domain-driven service design. In: Dustdar, S. (ed.) Service-Oriented Computing. pp. 189–208. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-64846-6_11
16. Kapferer., S., Zimmermann., O.: Domain-specific language and tools for strategic domain-driven design, context mapping and bounded context modeling. In: Proceedings of the 8th International Conference on Model-Driven Engineering and

Software Development - MODELSWARD,. pp. 299–306. INSTICC, SciTePress (2020). https://doi.org/10.5220/0008910502990306

17. Kapferer, S., Zimmermann, O.: Domain-driven architecture modeling and rapid prototyping with context mapper. In: Model-Driven Engineering and Software Development. pp. 250–272 (2021). https://doi.org/10.1007/978-3-030-67445-8_11
18. Le, D.M., Dang, D.H., Nguyen, V.H.: On domain driven design using annotation-based domain specific language. Computer Languages, Systems and Structures **54**, 199–235 (2018). https://doi.org/10.1016/j.cl.2018.05.001
19. Lewis, J., Fowler, M.: Microservices (2014), http://martinfowler.com/articles/microservices.html
20. Lopes, T., Silva, A.R.: Monolith microservices identification: Towards an extensible multiple strategy tool. In: 2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C). pp. 111–115 (2023). https://doi.org/10.1109/ICSA-C57050.2023.00034
21. Lourenço, J., Silva, A.R.: Monolith development history for microservices identification: a comparative analysis. In: 2023 IEEE International Conference on Web Services (ICWS). pp. 50–56 (2023). https://doi.org/10.1109/ICWS60048.2023.00019
22. Nunes, L., Santos, N., Rito Silva, A.: From a monolith to a microservices architecture: An approach based on transactional contexts. In: Software Architecture: 13th European Conference, ECSA 2019, Paris, France, September 9–13, 2019, Proceedings. pp. 37–52. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-29983-5_3
23. O'Hanlon, C.: A conversation with werner vogels. Queue **4**(4), 14–22 (May 2006). https://doi.org/10.1145/1142055.1142065
24. Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., Seinturier, L.: Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. Software: Practice and Experience **46**, 1155–1179 (2015). https://doi.org/10.1002/spe.2346
25. Ponce, F., Márquez, G., Astudillo, H.: Migrating from monolithic architecture to microservices: A rapid review. In: 2019 38th International Conference of the Chilean Computer Science Society (SCCC). pp. 1–7 (2019). https://doi.org/10.1109/SCCC49216.2019.8966423
26. Rademacher, F., Sachweh, S., Zündorf, A.: Towards a uml profile for domain-driven design of microservice architectures. In: Software Engineering and Formal Methods. pp. 230–245. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-74781-1_17
27. Richardson, C.: Developing transactional microservices using aggregates, event sourcing and cqrs. InfoQ (2017), https://www.infoq.com/minibooks/emag-microservices-monoliths/
28. Santos, N., Rito Silva, A.: A complexity metric for microservices architecture migration. In: 2020 IEEE International Conference on Software Architecture (ICSA). pp. 169–178 (2020). https://doi.org/10.1109/ICSA47634.2020.00024
29. Santos, S., Silva, A.R.: Microservices identification in monolith systems: Functionality redesign complexity and evaluation of similarity measures. Journal of Web Engineering **21**(5), 1543–1582 (2022). https://doi.org/10.13052/jwe1540-9589.2158
30. Singjai, A., Zdun, U., Zimmermann, O.: Practitioner views on the interrelation of microservice apis and domain-driven design: A grey literature study based on grounded theory. In: 2021 IEEE 18th International Conference on Software Architecture (ICSA). pp. 25–35 (2021). https://doi.org/10.1109/ICSA51549.2021.00011
31. Tune, N., Millett, S.: Designing Autonomous Teams and Services. O'Reilly Media, Incorporated (2017)

32. Vernon, V.: Domain-driven Design Distilled. Addison-Wesley (2016)
33. Vural, H., Koyuncu, M.: Does domain-driven design lead to finding the optimal modularity of a microservice? IEEE Access **9**, 32721–32733 (2021). https://doi.org/10.1109/ACCESS.2021.3060895
34. Özkan, O., Önder Babur, van den Brand, M.: Domain-driven design in software development: A systematic literature review on implementation, challenges, and effectiveness (2023). https://doi.org/10.48550/arXiv.2310.01905