PRACTICAL SOURCE CODE WEAVING FOR DISTRIBUTED WORKFLOW ABSTRACTIONS

Silviu-George PANTELIMON[1], Radu-Ioan CIOBANU[2], Ciprian DOBRE[3]

Aspect-oriented programming (AOP) is a modern software development paradigm that helps automate programming and increase software quality. A frequently used technique in AOP is called aspect weaving, which allows developers to reduce boilerplate code (lines of code repeated in multiple places with little to no variation), making it easier to add new features to existing source code. However, most frameworks that support AOP in various programming languages, with few exceptions, have not taken full advantage of this technique, especially when working on multiple projects with a shared code base (e.g., distributed systems like microservices). This paper proposes a novel approach that uses source code weaving to create complex workflows within a distributed application. By incorporating the workflow specifications directly into the code, our method aims to provide a unified abstraction for the distributed system, to simplify code and enable a better visibility of the distributed processes.

Keywords: code generation, categorial optics, programming paradigms

1. Introduction

In the current software development market, there are several libraries and frameworks for different purposes that enable easy-to-use Aspect-Oriented Programming (AOP) [9]. AOP is generally used to insert new features (not necessarily in a transparent manner) into existing code, which would be difficult to do otherwise. Suppose that we need to log the execution of some procedures in a complex application with many distinct modules. Here, logging becomes a cross-cutting concern because it needs to be executed in multiple parts of the code. With AOP, the logging aspect would encapsulate the logging logic and decouple it from the rest of the procedures.

[1]Ph.D. Student, Faculty of Automatic Control and Computer Science, National University of Science and Technology Politehnica Bucharest, Bucharest, Romania, silviu.pantelimon@upb.ro

[2]Professor, Faculty of Automatic Control and Computer Science, National University of Science and Technology Politehnica Bucharest, Bucharest, Romania, radu.ciobanu@upb.ro

[3]Professor, Faculty of Automatic Control and Computer Science, National University of Science and Technology Politehnica Bucharest, Bucharest, Romania, ciprian.dobre@upb.ro

In reality, by using AOP, developers may risk losing fine-grained control over application logging. An alternative in practice is to use dependency injection and an attribute/annotation processor to add a logging service as an object, where specified. For such a case, a form of weaving is still involved in many implementations, usually at compile-time, to avoid runtime overhead. With weaving techniques, we can generate code and introduce the logger into an annotated class readily available for use. For developers, the code becomes easier to write, more concise, and, therefore, more maintainable.

Using the same principles, but in a distributed context, we propose using attribute-oriented programming (@OP) with source code weaving to implement *distributed workflows*. Suppose that we want to implement a distributed workflow for registering a new publication from an external database in a CRIS[1] platform, which is an information system that stores and manages contextual metadata describing research-related impact indicators. Let us assume that we want to implement a distributed workflow for registering data about a new publication of a researcher. Any CRIS platform usually has two connected nodes: a data processing node, and a storage node connected to a database. As the publication arrives in the platform from an external source, it is sanitized and pre-processed by the first node, and then it is sent to the storage node to be persisted. On the storage node, a data aggregation process is triggered automatically for each publication author, so that the processing node can recalculate scientometric indicators.

By analyzing the specifications of the aforementioned CRIS platform, we can *interpret the workflow as a cross-cutting concern*. The idea is to encapsulate the specifications into generated code that would produce the desired (software) behavior for each distributed node. A widely used method would be to employ a saga pattern [5]. A saga is simply a sequence of transactions that update distributed services and publish a message or event to trigger the next transaction step. Many frameworks implement this pattern by declaring state machines in each node, which communicate with each other by exchanging events and triggering state transitions upon event reception. While this approach is undoubtedly practical, the downside is that the implementation is split between the distributed nodes, which means that we do not have a proper abstraction of the workflow, leading to intrinsic tangling in the system. Thus, this paper proposes an alternative AOP-based saga design construct and its corresponding technique to eliminate these problems.

In our approach, we encapsulate the specification as a single annotated interface in the shared code of the distributed system, representing every step of the workflow. For the case above, sanitizing the publication would be specified by a method in the annotated interface, which would then be used to weave the necessary source code. The developer would only have to use the generated code for their implementations separately for each program on different nodes.

[1]Current Research Information System

In this way, we can obtain a single contained specification that also encompasses the cross-cutting concern for the entire workflow as an abstraction in the code, with the added benefit that it involves more compile-time operations (rather than runtime) and is more maintainable for the developer.

The applicability of this solution is not limited to CRIS platforms. It is simply an example where we have experience implementing workflows that aggregate data from various sources. There are mission-critical business processes, like distributed transactions in e-Commerce or finance, that require a high degree of confidence in the code performing the transaction. Having parts of the code generated from some specifications kept in a single place may provide means by which such systems can be maintained more easily with faster time of development.

2. Similar Solutions

We should note some technologies used by the industry as a reference for software design. The most notable example that attempts this is JHipster [8], a framework that provides tools and a domain-specific language, JDL, to enable developers to rapidly implement web applications in Spring. However, some developers may find this framework difficult to learn as they need to understand the JDL and the command line tool. Besides, some may find that it adds unnecessary components by default and induces too much complexity to the software architecture. Therefore, many developers may opt out of using such a solution and prefer to write the necessary code for their purpose themselves, so any tool used should not be very invasive to what the developers are comfortable with. A good place where JHipster may be best used is in simple CRUD applications. For more complex applications with intricate business logic, it may not be as useful for these types of requirements. An alternative would be to have code generation tools integrated into the programming language itself with more customization support.

Another example in which code generation is employed in modern application development in synchronous communication between services is gRPC [6]. gRPC is a standard for remote procedure calls using the Protocol Buffer format to serialize the exchanged data. It is a good example where formal specifications are transposed into actual code. For example, in .NET, the gRPC library uses a .proto file where the contract and services are defined to generate the classes for the contracts, the abstract services which have to be implemented, and the clients for these services. It demonstrates how much of the source code can be derived from simple specifications.

While these tools work in practice, we would like to make some adjustments to these solutions. In both cases, having a distinct domain language is added to the development environment, which is similar to what AOP solutions such as AspectJ [7, 10] do. Another approach of weaving in features

is what the JPA[2] [3] does: from an interface, an implementation for complex database access is generated in compilation. The interface methods can be arbitrary, either following some naming conventions or being annotated with the corresponding database queries. The advantage of the JPA design is that the specifications are tightly coupled with the code and most IDEs[3] have custom support for the syntax. However, in this instance, unlike in AOP, no cross-cutting concern is addressed. But, looking at how the JPA functions, we would like to combine this approach of using an interface and @OP to address the distributed workflow as a cross-cutting concern, as will be described in the following sections.

3. Proposed solution

In the following, we describe our proposed solution, in which we intend to avoid some common pitfalls of working with an AOP framework, such as duplicated cross-cutting code or the application of incorrect advice [1].

For our goals, we must provide some basic assumptions for our development environment:

- Whenever we talk about AOP, other paradigms, or design patterns, we need to be aware that there has to be integration with the IDE. The developer has to be provided with both the necessary tools to work with the solution, but also with information about potential problems with the implementation.
- We assume that the implementation of the distributed system shares a common codebase for each node implementation, so we can share the metadata in our @OP approach to weave code into multiple projects all at once.
- We assume the workflows are structured as a directed acyclic graph, in representation and in the implementation. We impose this constraint to restrict the number of possible states the system is in, and to ensure that the workflow terminates.
- The workflow executes in an asynchronous manner and, as in most modern distributed cloud applications, we assume the asynchronous communication channel is a message queue.

With these assumptions, we propose the following steps in implementing a workflow, as shown in figure 1:

- First of all, the developer creates a library that is shared among the different projects where we can declare our specifications of the workflow.
- In the shared library, an interface (as in OOP) representing the workflow is declared, decorated by an attribute identifying the interface as workflow specification.

[2]Java Persistency API

[3]Integrated Development Environments

- The programmer explicitly declares what event objects are used to realize the saga, by where they occur in the methods of the interface, declaring what steps are taken in the saga and what sequence of events triggers these steps.
- On an incomplete or bad definition of the workflow, a code analyzer prompts a compilation error specific to the workflow declaration, which the programmer has to resolve to be able to compile the solution.
- After the correct definition of all the steps of the workflow, the generator emits the source code to handle the steps. With respect to modern programming standards, part of the generated code will be abstract (such as interfaces), defining a contract between the rest of the generated code that is concrete and the programmer that will implement these abstractions.
- The programmer implements these interfaces as handles in the distributed nodes with the appropriate code, thus enabling the workflow to be implemented in distinct processes.
- The handlers re-enter the compilation pipeline and are used for the generation of some other utility functions, such as the dependency injection declarations or the generation of event consumers to call the event handlers.

To exemplify, we shall take the aforementioned example and add a few more steps, as shown in figure 2.

It can be seen here that we added a few attributes to the interface that will change the implementation details. The important thing to note here is that we use the interface as both an abstraction and a formal specification, to encapsulate the workflow structure. Although we could have specified the workflow as a directed acyclic graph in a distinct file, we actually want to apply type theoretical and categorial theoretical operations on it, and a tighter coupling with the programming language would confer us a few advantages. For instance, we can use the syntax trees and semantic model exposed by the compiler to have fine grained control over what code is produced.

Going further, we can view the interface as a category where methods are objects and event types are morphisms between them. This nuanced view carries with itself operations known in category theory which can be applied to our example on multiple levels. Having the workflow as data structure in our code we can therefore apply functors as in figure 3 so we can map each method onto interfaces for event handlers and the event messages onto generic type specializations. This mapping as described cannot be expressed properly in most languages even if they support generic programming, and this is the reason why we need code generation.

The code generation process allows us to take the structure provided by the interface and metadata and transform it via a functor preserving the structure of the interface enabled workflow. The important aspect here is that the structure of the interface is evaluated by the compiler and generator, so the

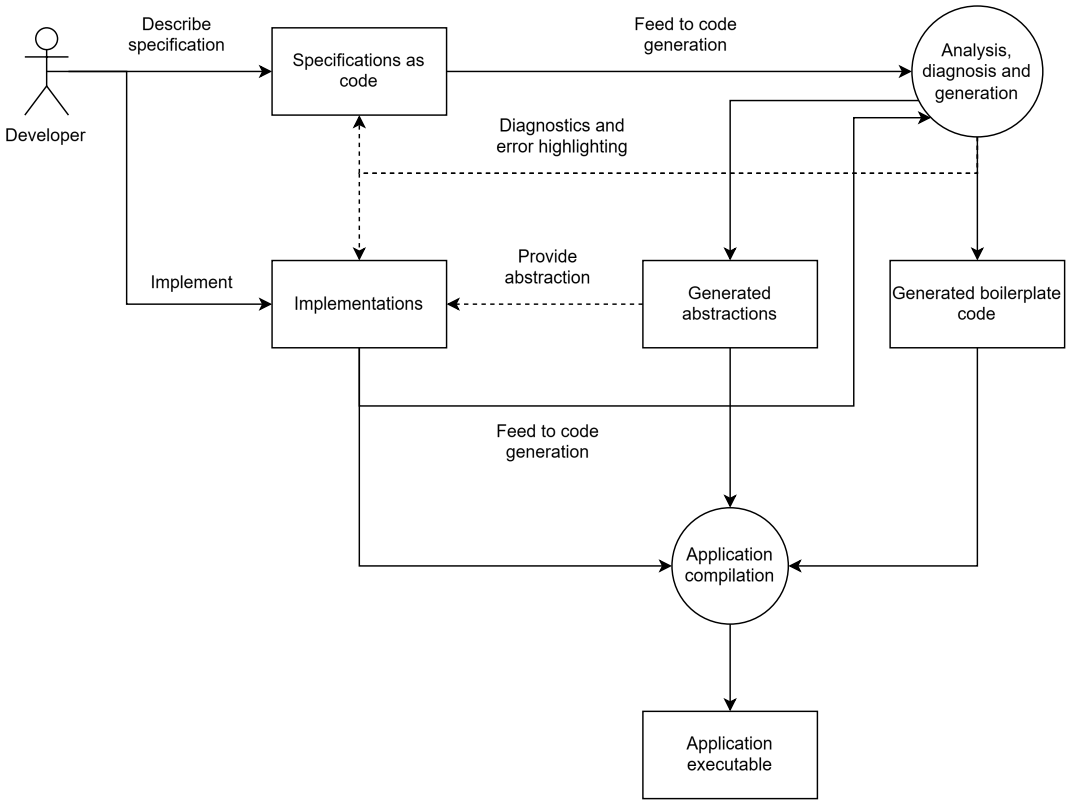

FIG. 1. The proposed development and compilation pipeline.

```
[Workflow(defaultHandlerLifetime: Lifetime.Scoped, defaultDeliveryTimeout: 5000)]
public interface IPublicationWorkflow {
    public (SanitizedPublication, PublicationMetadata) Sanitize(RawPublication rawPublication);
    public PublicationSaved SavePublication(SanitizedPublication sanitizedPublication);
    public PublicationProcessed RecalculateMetrics(PublicationMetadata publicationMetadata);
    [HandlerOptions(deliveryTimeout: 4000)]
    public void Finalize(PublicationSaved publicationSaved, PublicationProcessed publicationProcessed);
}
```

FIG. 2. A workflow specification example.

structure is visible to them as an internal semantic model. From evaluating the syntax and with access to that internal model, we can map it to other models. Therefore, we can implement functors over the abstract model of the workflow via maps between these internal models.

Furthermore, we can obtain interesting properties for the code generation. Primarily, we can compose the functors to enable a modular code generation process if necessary. Secondarily, we can use the functors to template the saga structure like in a dependent type. The workflow described as an interface could, in principle, be used to parameterize a generic type with a specific behavior. For example, one could implement classic functional types such as

FIG. 3. Example of a mapping.

lists over the workflow. In this context, rather than one event triggering the workflow, a collection of events are processed by a cumulative workflow which could succeed if any or all of individual workflows succeed. With this perspective, we can work with the workflow structure on multiple levels: conceptual, code and behavioral.

Regarding the generated code placement, the workflow interface is declared in a shared library. All projects need to view the same workflow topology in the first place, and also need to have the same data contracts provided to correctly communicate between each other. The generator extracts the model from the interface and transforms each individual method in that interface into a handler interface serving as an abstraction placed in the same library, to be visible for projects using them. Multiple projects could use the same abstraction. For instance, we could have a project using the abstraction to implement the production logic and another to implement tests cases for the workflow. The handler interface may add or encapsulate data types in their signature, in addition to the given definition in the corresponding method.

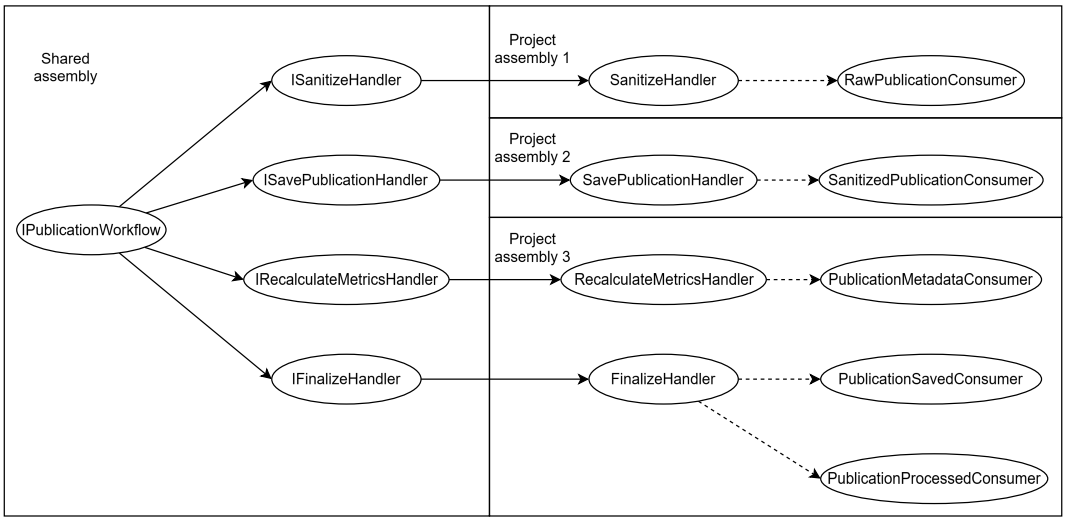

FIG. 4. Code generation placement.

As in figure 4, the shared library/assembly where the workflow interface is declared will contain, after the code generation, the handler interfaces that can be seen by other projects referencing that shared code.

When a project implements a handler interface, the code generator will see the new class and re-trigger the generation process. In this step, there may be different components that need to be created if the handler implementation is present. The implementation itself is a minimal requirement to implement other components such as the message consumer. A consumer in the broader context of asynchronous communication is an endpoint that waits for an event to trigger some logic (the logic in this case being implemented by the handler). However, we need to make it clear that the consumer should not use the implementation directly, but instead use the abstraction as provided. The code generation process must be subjected to the same standards and level of scrutiny as manually written code, for instance, by respecting SOLID principles (Single-responsibility, Open-closed, Liskov Substitution, Interface Segregation, Dependency Inversion).

While there are some advantages of using AOP over @OP with source code weaving, like the fact that it adds new features to already existing code, with our approach the concepts AOP deals with are transformed to help us implement the workflows:

- Aspect: the workflow encapsulated in the interface becomes the aspect or module that cuts across multiple projects/processes.
- Joint point: the joint points become implementations of generated abstractions that need to be executed to enable the business logic.
- Advice: The advice becomes a specific implementation wrapping around a joint point like an event consumer. Basically, in our approach this is

a case of inversion of control, as the joint point is called by the advice rather than triggering it.
- Pointcut: There is no proper correspondent in this case, there are no regular expressions to associate a joint point with an advice. At most, the pointcut is represented by implementing the generated abstractions.

The whole point of changing the paradigm and employing more sophisticated methods is to get the desired application behavior with fine-grained control. Moreover, employing some inversion of control is beneficial to modern applications as it streamlines the application control flow. While it may seem that our approach is rather different than common AOP, this paradigm is the standard in terms of encapsulating cross-cutting concerns. So, we simply employ another method for a specific use-case to overcome the shortcomings of the original paradigm, while keeping the same notion of aspects.

An advantage code generation can give us is that, for specific instances, we can generate code blocks needed to implement the logic for these use-case. As in our example, the last step requires two messages of different types to arrive and produce a result. In a scenario where we cannot assume the order of message arrivals because the communication is asynchronous, we build a custom logic. We store each event in a database with a shared generated sequence number and call the event handler on both only by the consumer who saved the greatest sequence number and has all the event messages. It is a simple logic for which we can generate appropriate code while the developer does not need to take more steps than in other scenarios.

4. Implementation Details

4.1. Source Generators with Roslyn

For our implementation we used the Roslyn compiler platform [11] for source generation in .NET for C#. Roslyn is a versatile tool providing us with the necessary features to implement out solution. It supports both code generators and code analyzers for diagnostics.

To explain how the code generator works, we need to understand how the Roslyn API works while the programmer codes. Whenever the source code in a project is changed, the compiler sends it to the Roslyn pipeline, where there is a code analyzer or generator that is called after the source code is parsed.

Roslyn uses two models to represent the output after parsing the source files: a syntax tree representation of the various symbols defined in the language's grammar (in this case in C#) and a semantic model that puts the symbols found in the syntax tree in context and attributes them a meaning.

When writing a code generator, we need to work with both, but not in the same way the pipeline uses them. While many examples of code generators we found use raw string templates to generate source code, this is not sustainable, neither for our purposes nor from a general programming perspective. Instead, we do the reverse set of operations from the pipeline: we start from an already

existing semantic representation for which we construct the syntax tree, and then reify it to code.

The reason for choosing this approach is that we can generate code more dynamically instead of defining code templates for every extremely specific instance, and because we can use the already existing Roslyn API to compose source code with syntax tree builders.

On top of the Roslyn API, we added an intermediary representation using the semantic model. For each symbol we extract from the semantic model (e.g., classes, interfaces, method members, type definitions, etc.), we specifically represent it as an object tailored for that instance. We do it because, on one hand, we extract these representations from the semantic model exposed by the compiler API. On the other hand, we also create new representations, adding them to the compilation pipeline. For example, we can extract a class representation in a *ClassRepresentation* object (a class defined by us) from an *ITypeSymbol* Roslyn exposes, so we get information such as the members it contains or its methods. We can also build a new class representation by constructing it directly, if we need to. In both instances, we can build the syntax tree and add the source file from the same representation, but it is easier working directly with the Roslyn models instead of switching back and forth between the syntax tree and the semantic model. We simply transform the intermediary representation into another and then reify it.

Additionally, while we are still using the Roslyn API to simplify the syntax tree building, we wrapped it into extension methods[4]. The benefit of that is that the syntax is easier to read and write by using method chaining rather than by nested function calls, a technique which itself is nothing new but is worth noting that it is facilitated by using extension methods. From what we have seen, using method-chaining and encapsulating frequently used syntax can significantly reduce the code required to build a specific syntax tree.

Other features of Roslyn that we used are incremental generators. In short, the developer builds the generation pipeline via basic functional operations (projections, filters, aggregations) from the syntax trees and semantic model, and sets the output, the source files, and the diagnostics. Incremental generators ensure that, on each step in the specified pipeline, the intermediary values are cached. At each execution, if the new values generated at each step remain the same as the previous cached values, the next step is not triggered again.

[4]Extension methods are methods added to a class/interface after it was written, but called like normal methods.

4.2. Implementing the Generation Functor

To start describing the generation functor, we should firstly describe the source category, which, in our case, is the annotated interface. Let $\mathcal{WF}$ be a generic workflow interface defined as follows:

- $Ev(\mathcal{WF})$ - the set of event message types in the workflow.
- $Ob(\mathcal{WF}) = \mathcal{M} \cup \mathcal{EP}$ - the set of objects, i.e., the set of methods $\mathcal{M}$ in the interface and set of entry-points $\mathcal{EP}$ in the workflow, an entry-point being a operation that triggers events from the outside of the workflow to initiate some logic.
- $Hom(\mathcal{WF}) = \{path_{a,b} | path_{a,b} = \bigcup_{i \in \mathbb{N}, x_i \subseteq Ev(\mathcal{WF})} x_i; a, b \in Ob(\mathcal{WF}); if\ a = b\ then\ path_{a,b} = \varnothing\}$ - the set of morphisms which are the sequences of event messages sent from a method or an entry-point to another method where composition is the concatenation of the sequences.

Decomposing the workflow, we can easily describe the functor piece-wise through simple maps:

- A bijective function $f_{Ev} : Ev(\mathcal{WF}) \to Ev(\mathcal{WF}_\mathcal{G})$ and a dependent type $Event <>$ such that $\forall a \in Ev(\mathcal{WF})\ f_{Ev}(a) = Event < a >$. Basically, we can take for granted a generic type in the programming language and this function is implemented by the language.
- A function $f_M : \mathcal{M} \to \mathcal{H}_\mathcal{G}$ that rewrites the methods as handler interfaces. If a method is a tuple $m = (name, ret\_type, [par\_type]) \in \mathcal{M}$ then $f_M(m) = (\$"I\{name\}Handler", [("Handle", ret\_type, [f_{Ev}(par\_type)])])$. Then the set $Ob(\mathcal{WF}_\mathcal{G}) = \mathcal{H}_\mathcal{G} \cup \mathcal{EP}$.
- A function $f_{Hom} : Hom(\mathcal{WF}) \to Hom(\mathcal{WF}_\mathcal{G})$ such that $\forall a \in Hom(\mathcal{WF})$ and $\forall x \in a$ then $f_{Ev}(x) \in f_{Hom}(a)$.

So we can define the target category $\mathcal{WF}_\mathcal{G}$ such that the code generator functor $F_G : \mathcal{WF} \to \mathcal{WF}_\mathcal{G}$ is defined as the tuple (f_{Ev}, f_M, f_{Hom}). This functor represents the code generation only for the event handlers as an example. The functor is further illustrated here:

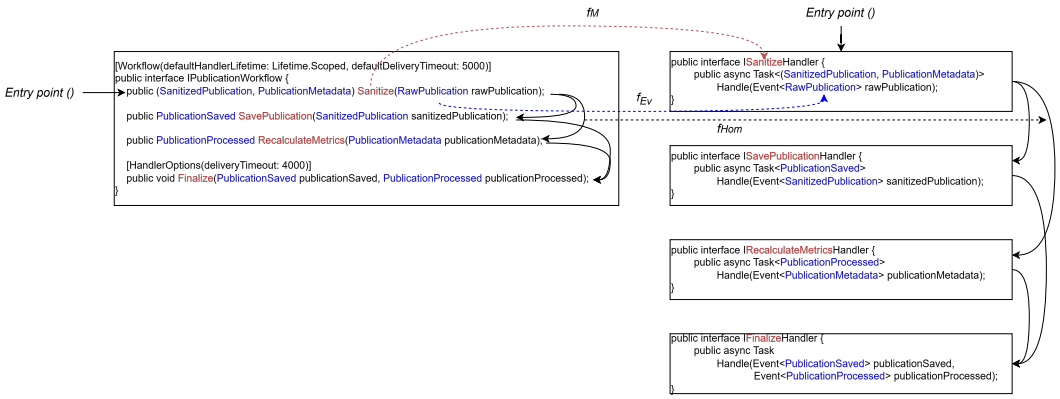

FIG. 5. The functor mapping illustrated element by element.

In reality, we can devise more functors such as this depending on our requirements. Moreover, the programming language we are using already implements genericity. If we need to modify the generated code, we will only need to tweak the f_M function. For example, we could introduce a new parameter in the handler interfaces if we need to pass additional data structures.

This example intended to show how easily we can encapsulate the code generation as a functor and use categories to conceptualize the workflow. We could have done this without a reference to categories and use a simple set theoretical model, but there is one nuance to this method. Because the way we described our categories, the composition of paths is preserved. This means, non-formally, that how the workflow behaves, i.e., the chain of events represented by the interface, is conceptually preserved in the resulting code.

4.3. Code Generation with Optics

An idea we toyed with was to involve optics in the code generation process. Optics are bidirectional data accessors [4], and we find this idea interesting for several reasons. The first one is that they might modularize code generation steps, while another reason is that outside of Haskell implementations, they were not adopted as design patterns in object-oriented languages.

Definition 4.1. *Let there be two categories $\mathcal{C}$ and $\mathcal{D}$ with objects $S, A \in \mathcal{C}$ and $T, B \in \mathcal{D}$. A (generalized) $(Ⓛ, Ⓡ)$-optic from (S, T) to focuses (A, B) is, according to* [4]*, an element of:*

$$\mathbf{Optic}_{Ⓛ,Ⓡ}((A,B),(S,T)) := \int^{M \in \mathcal{M}} \mathcal{C}(S, M Ⓛ A) \otimes \mathcal{D}(M Ⓡ B, T)$$

where $\mathcal{M}$ is a monoidal category with two (left) actions $Ⓛ$, $Ⓡ$ acting on $\mathcal{C}$ respectively $\mathcal{D}$.

The usual representation found in the literature is shown in figure 6. Conceptually, an optic decomposes an object via an action, getting the focused object and a context. The context is used to recompose another object via a potentially different action. From a programming perspective, we can imagine that these objects are types. We use the optic to extract an inner data structure and by passing the context we can reassemble another type. If we had replaced the two actions with the cartesian product, basically composing types as tuples, the optic would simply deconstruct and reconstruct data structure to access and write nested data.

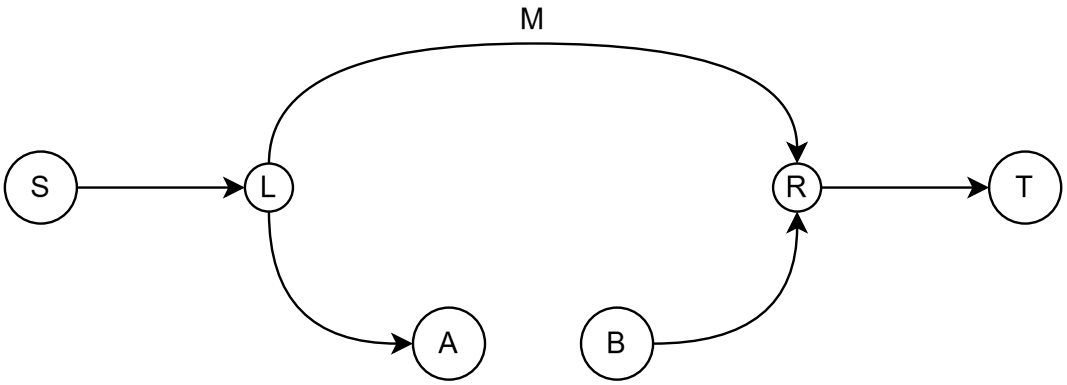

FIG. 6. A graphical representation of a generic optic.

Depending on what actions are used, the optic reduces to a specific "shape" via a Yoneda reduction. For example, if the first action is a product, the optic is a lens, while if the other action is a coproduct, the optic is a prism[5].

For the sake of keeping the implementation description short, we will focus on lenses and adapters to show how we implemented the source generation.

Definition 4.2. *A (classical) lens from* (S, T) *to focuses* (A, B) *is an element of:*

$$\mathbf{Lens}((A, B), (S, T)) := \mathcal{C}(S, A) \otimes \mathcal{D}(S \times B, T)$$

Definition 4.3. *An adapter from* (S, T) *to focuses* (A, B) *is an element of:*

$$\mathbf{Adapter}((A, B), (S, T)) := \mathcal{C}(S, A) \otimes \mathcal{D}(B, T)$$

The usefulness of optics relies on the fact that compatible optics can be composed to create other optics in the desired shape. However, as with all patterns, they become even more useful the more degrees of freedom we have acting on them. Therefore, we also use an interesting property of optics:

Theorem 4.1. *Let there be functors* $F : \mathcal{C} \to \mathcal{C}'$ *and* $G : \mathcal{D} \to \mathcal{D}'$ *exhibiting tensorial (left-)costrength respectively tensorial (left-)strength over the respective actions of an* $(Ⓛ, Ⓡ)$*-optic. The pair* (F, G) *determines a functor between optics according to* [2]*:*

$$\mathbf{Optic}_{F,G} : \mathbf{Optic}_{\mathcal{C},\mathcal{D}} \to \mathbf{Optic}_{\mathcal{C}',\mathcal{D}'}$$

This a useful property we can use in some instances. To exemplify, we can take the following example to transform a workflow interface into multiple event handle interfaces.

[5]As generalized optics are defined an optic can be both a lens and a prism.

FIG. 7. The elementary code generation steps organized into optics.

We can identify several steps to access components from the workflow interface and structure the resulted code. We group the elementary steps into a lens and two adapters as shown in figure 7[6].

One adapter changes the method signature to return a Task (a type of promise) and add a cancellation token parameter like most asynchronous methods have in C#. The other adapter extracts the method signature and adds the name "Handle" to a method signature to get a proper interface method.

The lens simply extracts the method list from the workflow interface and encapsulates a method into an interface with information from the workflow interface to generate a name for that handler interface.

From these building blocks, we can first compose the two adapters. Because the resulted adapter has the same focuses, we can shortcut them and append them like a simple transformation to the lens. The resulted optic is still a lens with the same signature as the original.

Finally, we can apply a functor on the resulted lens as the pair (Identity, IEnumerable), lifting this lens to another. The identity functor has costrength with any type of operation and the IEnumerable[7] functor is strong in relation to the cartesian product, so we can apply it on the lens. The purpose is to get the same type of focuses at the end, so we can once again shortcut them into a single transformation.

Putting all together into code in object-oriented semantics, the schema resulted in figure 8 becomes a simple function built through method-chaining as in figure 9. The C# library used to implement optics for the source generator is published by us on a public Gitlab[8].

[6]The diagram notation is borrowed from Bruno Gavranović.

[7]IEnumerable is an interface providing iterators for collections and acts as a functor in C#.

[8]https://gitlab.com/Say10/saytenoptics

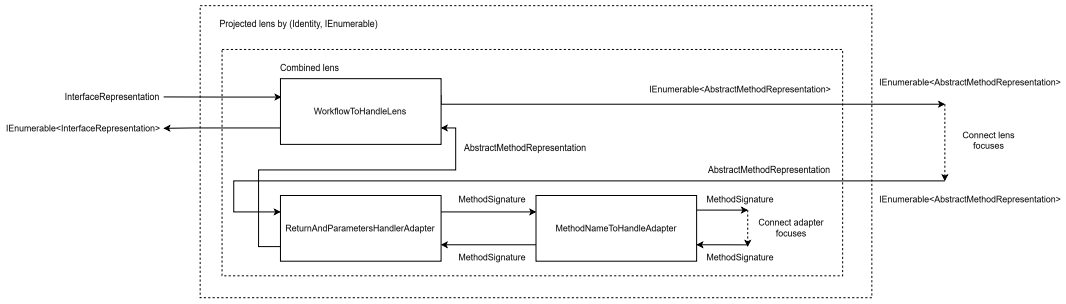

FIG. 8. The final transformation resulted by optic composition and functor application.

```
var generateHandlerInterfaces =
WorflowToHandlerLens
    .PrependUpdate(
        ReturnAndParametersHandlerAdapter
            .Compose(MethodNameToHandleAdapter)
            .CondenseAdapter())
        .ApplyEnumerableOnRightLens()
    .CondenseLens();
```

FIG. 9. A code generation transformation enabled by optics.

To make a disclaimer here, we do not not claim that this approach is the best method for generating code by step-wise transforming code representations. It is, however, a method worth studying for code modularization techniques. In this example, if we would need to change the generated code, we could simply insert another optic where we shortcut the focuses.

Another way the mentioned functor between optics helps in the constructing code generation pipelines is that we can shortcut optics by "adjusting" the focuses through the functors. For example, we can use elements of these types of lenses and apply the functors and compose the two morphisms (an example that can be extended to other types of optics):

$$\mathbf{Lens}((A, FA), (S, T)) \to \mathbf{Lens}((FA, FA), (FS, T)) \to [FS, T]$$

$$\mathbf{Lens}((GA, A), (S, T)) \to \mathbf{Lens}((GA, GA), (S, GT)) \to [S, GT]$$

An example of how prisms can be used in our code generation method is the case of generating diagnostics. In the example bellow we have a prism which is also a lens that has two operations, a "match" operation that either produces a list of error diagnostics if their parameters are not valid or just passes the interface and a "build" operation that takes two interface representations and outputs a list of diagnostics regarding whatever or not the two interfaces share data types in their signatures.

Definition 4.4. *A (generalized) prism from* (S, T) *to focuses* (A, B) *is an element of:*

$$\mathbf{Prism}((A, B), (S, T)) := \mathcal{C}(S, T \bullet A) \otimes \mathcal{D}(B, T)$$

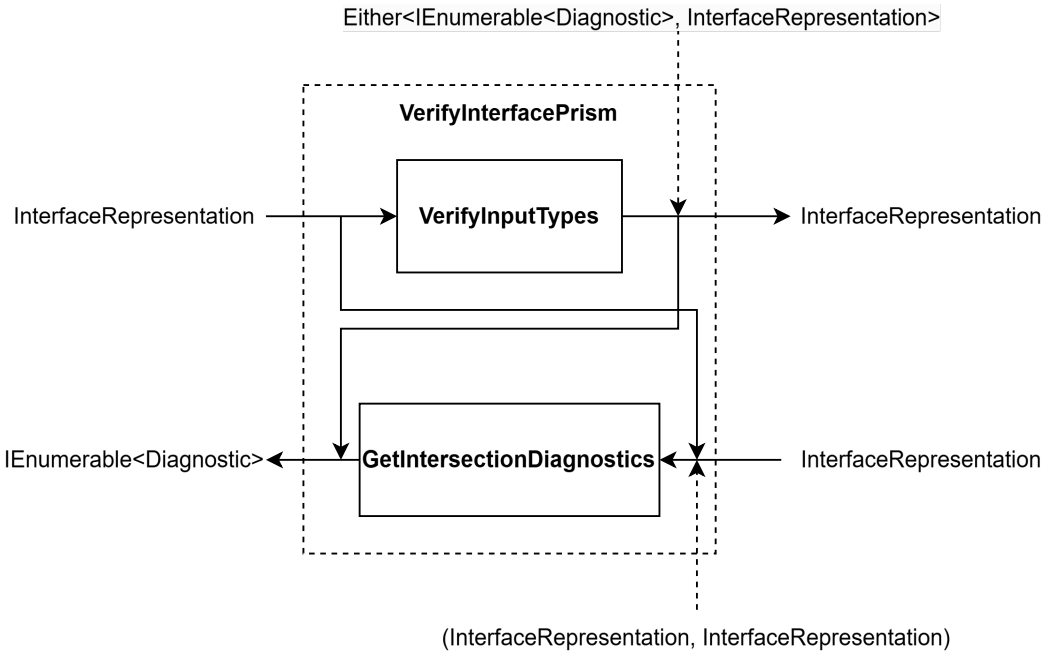

FIG. 10. An example of a prism also acting as a lens that analyzes an interface definition and outputs a list of diagnostics.

While in this case we can directly connect the focuses of the prism to obtain the necessary transformation as in figure 10, it would not give us the desired behavior because it would pass the same interface to the "build" operation and get us the wrong diagnostics. Instead we can represent this prism ignoring that it is also a lens as in figure 11.

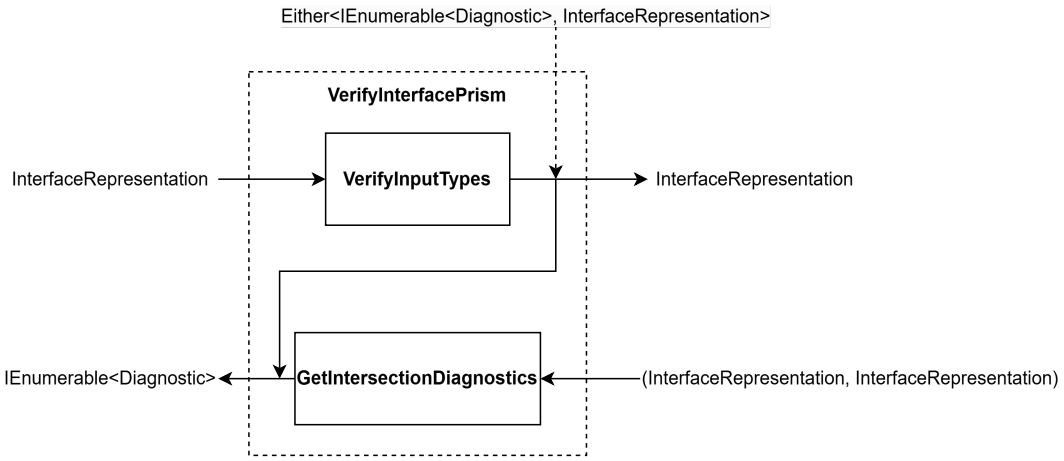

FIG. 11. The same prism represented without the lens aspect.

We can see here that we cannot connect the two focuses together because they are of different types. Instead we can apply a functor that is costrong in relation to the cartesian coproduct, namely the functor F that $\forall a,\ F(a) = a \times R$, in this case R is another interface representation.

The functor is costrong because $a \times (b + c) \rightarrow b + (a \times c)\ \forall a, b, c$ in a cartesian closed category.

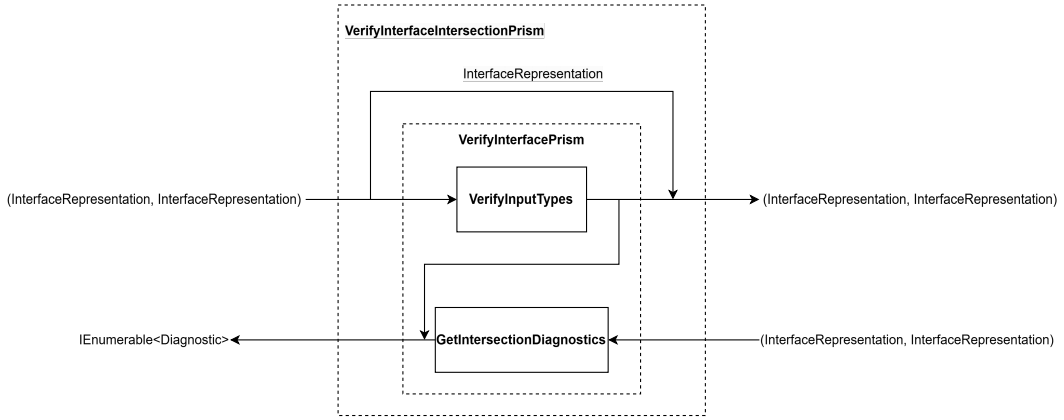

FIG. 12. The prism resulted after applying the functor.

As seen in figure 12 we get the desired focuses which we can connect, but to retain the same prism shape the costregth of the functor is applied resulting in the prism from figure 13.

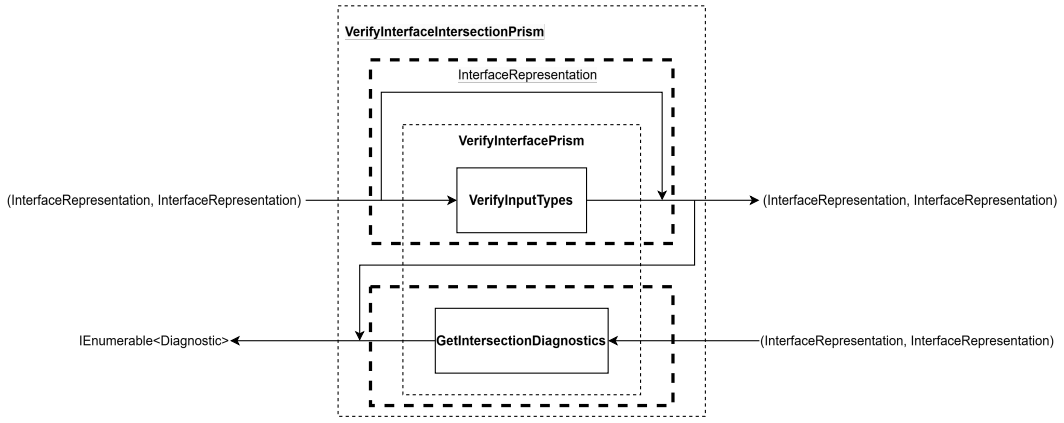

FIG. 13. The prism resulted after applying the costrength of the functor.

This new prism we can use to get a transformation that for any two interfaces can produce a list of diagnostics or compose it with other optics to build a more complex pipeline.

We should note that a problem we faced working with optics is that, while composition is relatively easy once the compatible parts were identified, the optics themselves were difficult to express in object-oriented semantics. The reason stems from the language's implementation of generic programming and type inference, which does not always translate well from purely functional setting (e.g. proper functor implementations). These limitations are not characteristic to C#, but also other very common programming languages such as Java and Kotlin.

Our final remark on this approach is that we can combine the usage of optics with the incremental source generator. We can use the optics alone to transform code and use the caching mechanism when passing from one optic to another to avoid re-computing an output from an optic.

5. **Perspectives on the evaluation**

We have thought about evaluating our proposed solution in one way or another, but there are two aspects to consider here. First, with respect to the code generation, the entire code generation and analysis process takes very little time to finish. We tried to measure this on different workstations, but the entire pipeline finishes in less than one millisecond. It has less to do with our usage of optics and more to do with the compiler platform itself and how it is optimized.

Second, there is the question of the generated code. We would not insist on the performance evaluation as much as we would on the utility of this solution compared to others. The main idea of this solution is to improve the development time for developers and comprehension of the workflow. This is

difficult to quantify, as it highly depends on the developer's comfort in using such solutions. However, we imagine that the best way to provide a proper comparison is to take an existing system and migrate the code to our solution. The metrics to compare before and after the migration would be the number of reported incidents related to the workflow and the time to repair these issues.

We leave the evaluation as a topic for future investigation as it relates to the development process itself, which is hardly quantifiable.

6. **Conclusions**

In this paper, we have shown a novel approach to modern distributed applications software development. In our proof-of-concept, we have employed concepts related to AOP, but changed the framework used to accommodate more sophisticated development tools for particular applications.

Our solution employs the usage of source code generation, which has become a trend in many current application development frameworks used in the market. This is no surprise, as they can overcome some programming language limitations.

A contribution we made here was to use optics as an experimental method to transform code in the desired shape. We hope that, in the future, we can expand on this idea from multiple angles. For one, we would like to involve other types of optics (e.g. prisms, grates, traversals), and secondly, we would like to have a better formalization of the code generation process as functors enabled by optics.

We want to keep the discussion of this paper open-ended, because there is still a lot of research to be done, not only from a formal perspective but also from the perspective of software development practices.

REFERENCES

[1] *Alves, P., Figueiredo, E., Ferrari, F.*: Avoiding code pitfalls in aspect-oriented programming. In: Programming Languages: 18th Brazilian Symposium, SBLP 2014, Maceio, Brazil, October 2-3, 2014. Proceedings 18. pp. 31–46. Springer (2014)

[2] *Balan, A., Pantelimon, S.G.*: Optics, functorially. 17th International Workshop on Coalgebraic Methods in Computer Science (CMCS 2024) (2024)

[3] *Böck, H.*: Java persistence api. In: The Definitive Guide to NetBeans™ Platform 7, pp. 315–320. Springer (2012)

[4] *Clarke, B., Elkins, D., Gibbons, J., Loregian, F., Milewski, B., Pillmore, E., Román, M.*: Profunctor optics, a categorical update. Compositionality **6** (2024)

[5] *Garcia-Molina, H., Salem, K.*: Sagas. ACM Sigmod Record **16**(3), 249–259 (1987)

[6] *Giretti, A.*: Creating an asp. net core grpc application. In: Beginning gRPC with ASP. NET Core 6: Build Applications using ASP. NET Core Razor Pages, Angular, and Best Practices in. NET 6, pp. 155–221. Springer (2022)

[7] *Gradecki, J.D., Lesiecki, N.*: Mastering AspectJ: aspect-oriented programming in Java. John Wiley & Sons (2003)

[8] *Halin, A., Nuttinck, A., Acher, M., Devroey, X., Perrouin, G., Baudry, B.*: Test them all, is it worth it? assessing configuration sampling on the jhipster web development stack. Empirical Software Engineering **24**, 674–717 (2019)

[9] *Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.*: Aspect-oriented programming. In: ECOOP'97—Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings 11. pp. 220–242. Springer (1997)

[10] *Soares, S., Laureano, E., Borba, P.*: Implementing distribution and persistence aspects with aspectj. ACM Sigplan Notices **37**(11), 174–190 (2002)

[11] *Vasani, M.*: Roslyn Cookbook. Packt Publishing Ltd (2017)