Accelerating Software Development Through Integrated Domain-Driven Program Synthesis

by

Nathanael Mercaldo

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

August 2023

APPROVED:

_____

Professor George Heineman, Major Thesis Advisor

_____

Professor Rose Bohrer, Reader

_____

Professor Craig Shue, Head of Department

Contents

1 Introduction **7**
- 1.1 Problem Statement . . . 7
- 1.2 Motivation . . . 13
- 1.3 Background . . . 15
- 1.4 Proposed Solution . . . 16
- 1.5 Method of Evaluation . . . 17

2 Approach **18**
- 2.1 Technology . . . 18
  - 2.1.1 Background . . . 19
  - 2.1.2 Architecture . . . 20
  - 2.1.3 Domain Models in CoGen . . . 23
  - 2.1.4 Synthesis Specifications in CoGen . . . 23
- 2.2 Methodology . . . 27
  - 2.2.1 Background . . . 27
  - 2.2.2 Definition . . . 28
  - 2.2.3 Inspiration . . . 32
- 2.3 Design Patterns . . . 32
  - 2.3.1 Policy Pattern . . . 33
  - 2.3.2 Type Projection Pattern . . . 33

3 Related Work **34**
- 3.1 Synthesis Paradigms . . . 34
  - 3.1.1 Deductive Synthesis . . . 34
  - 3.1.2 Type-Driven Program Synthesis . . . 35
  - 3.1.3 Inductive Synthesis . . . 35
  - 3.1.4 Counterexample-guided Inductive Synthesis . . . 35
  - 3.1.5 Program Sketching . . . 35
  - 3.1.6 Natural Language Driven Synthesis . . . 36
  - 3.1.7 Diagram Driven Synthesis . . . 36
- 3.2 Synthesis Systems . . . 36
  - 3.2.1 GUI Boilerplate Synthesis . . . 36
  - 3.2.2 Control Systems Synthesis . . . 36
  - 3.2.3 Compiler Code Synthesis . . . 37
  - 3.2.4 API Code Synthesis . . . 37
  - 3.2.5 Multi-Domain Synthesis . . . 37
- 3.3 Comparison to CoGen . . . 37

4 Case Study - GUI Domain **39**
4.1 Overview . . . 39
4.2 Domain Description . . . 39
4.3 Target Description . . . 39
4.4 Application of Methodology . . . 40
4.4.1 Distillation - 1 - JavaFX . . . 40
4.4.2 Modeling - 1 - JavaFX . . . 40
4.4.3 Synthesis - 1 - JavaFX . . . 40
4.4.4 Distillation - 2 - JavaFX . . . 41
4.4.5 Modeling - 2 - JavaFX . . . 41
4.4.6 Synthesis - 2 - JavaFX . . . 41
4.4.7 Distillation - 1 - QT . . . 41
4.4.8 Modeling - 1 - QT . . . 42
4.4.9 Synthesis - 1 - QT . . . 42
4.4.10 Distillation - 1- libGDX . . . 42
4.4.11 Modeling - 1 - libGDX . . . 42
4.4.12 Distillation - 1 - LibGDX . . . 42
4.5 Results . . . 43
4.5.1 Artifacts . . . 43
4.5.2 Domain Model . . . 43
4.5.3 JavaFX . . . 45
4.5.4 QT . . . 48
4.5.5 LibGDX . . . 49
4.6 Evaluation and Reflection . . . 51

5 Case Study - Robotics Domain **52**
5.1 Overview . . . 52
5.2 Target Description . . . 52
5.3 Domain Description . . . 52
5.4 Application of Methodology . . . 52
5.4.1 Distillation - 1 . . . 52
5.4.2 Modeling - 1 . . . 53
5.4.3 Synthesis - 1 . . . 53
5.4.4 Distillation - 2 . . . 53
5.4.5 Modeling - 2 . . . 53
5.4.6 Synthesis - 2 . . . 54
5.4.7 Distillation - 3 . . . 54
5.4.8 Modeling - 3 . . . 54
5.4.9 Synthesis - 3 . . . 54
5.4.10 Modeling - 4 . . . 54
5.4.11 Synthesis - 4 . . . 55
5.5 Results . . . 55
5.5.1 Artifacts . . . 55
5.5.2 Domain Model . . . 55
5.5.3 ROS Node . . . 57
5.5.4 ROS Node Listeners . . . 59
5.5.5 User logic fragments . . . 60
5.5.6 ROS Messages . . . 61
5.5.7 ROS Loop Class . . . 62
5.6 Evaluation and Reflection . . . 62

6 Case Study - 3D Rendering Domain **63**
6.1 Overview . . . 63
6.2 Target Description . . . 63
6.3 Domain Description . . . 63
6.4 Application of Methodology . . . 64
6.4.1 Distillation - 1 . . . 64
6.4.2 Modeling - 1 . . . 64
6.4.3 Synthesis - 1 . . . 65
6.4.4 Modeling - 1.1 . . . 65
6.4.5 Distillation - 2 . . . 65
6.4.6 Modeling - 2 . . . 66
6.4.7 Synthesis - 2 . . . 66
6.4.8 Distillation - 3 . . . 66
6.4.9 Modeling - 3 . . . 66
6.4.10 Synthesis - 3 . . . 66
6.5 Results . . . 66
6.5.1 Artifacts . . . 66
6.5.2 Domain Model . . . 66
6.5.3 Vulkan App . . . 67
6.6 Evaluation and Reflection . . . 68

7 Evaluation **69**
7.1 Quantitative Analysis . . . 69
7.2 Qualitative Analysis . . . 70

8 Conclusion and Future Work **72**
8.1 Domain Model Composition . . . 72
8.2 Domain Model and Synthesis Specification Reuse . . . 72
8.3 Generating Game Engine Components . . . 72
8.4 IDE Tooling . . . 73

List of Figures

1.1 Sample Boilerplate code . . . 9
1.2 Desired GPS Location Method . . . 10
1.3 Swing Boilerplate Code . . . 11
1.4 C++ Boilerplate Code . . . 12
1.5 Illustration of boilerplate overlap and variability . . . 13
1.6 Span of synthesis approaches . . . 16

2.1 System context diagram illustrating CoGen inputs and outputs . . . 20
2.2 Simplified system structure diagram illustrating the layered architecture of CoGen 21
2.3 Illustration of generator tree organization . . . 22
2.4 EpCoGen Fibonacci Implementation . . . 24
2.5 Synthesis tree for Fibonacci example . . . 26
2.6 More sophisticated synthesis tree . . . 27
2.7 High-level illustration of the CDMS execution flow (execution begins in the "Coding" state) . . . 29

3.1 Span of synthesis approaches . . . 34

4.1 UML diagram of GUI domain model . . . 44
4.2 Synthesis tree for JavaFX target - 1 . . . 45
4.3 Synthesis tree for JavaFX target - 2 . . . 46
4.4 Synthesis tree for JavaFX target - 3 . . . 47
4.5 Synthesis tree for QT target . . . 48
4.6 Synthesis tree for LibGDX - 1 . . . 49
4.7 Synthesis tree for LibGDX - 2 . . . 50
4.8 Synthesis tree for LibGDX - 3 . . . 51

5.1 UML diagram of robotics domain model . . . 55
5.2 ROS Node synthesis tree - 1 . . . 57
5.3 ROS Node synthesis tree - 2 . . . 58
5.4 ROS Node Listener synthesis tree . . . 59
5.5 User logic fragments synthesis tree . . . 60
5.6 ROS Message synthesis tree . . . 61
5.7 ROS Loop synthesis tree . . . 62

6.1 Rendering domain UML diagram . . . 67
6.2 Vulkan synthesis tree . . . 68

Abstract

Program synthesis technology promises to accelerate the development of software systems by automating supporting concerns not directly related to the problem or domain under consideration. In this thesis, we present a novel methodological approach to multi-domain program synthesis and demonstrate the capability of our approach to accelerate the software development process by mitigating the need to write and maintain boilerplate code. Simultaneously, we demonstrate how our approach promotes increased system modularity and extensibility all while leading to simpler system implementations.

Acknowledgements

I would like to express my deepest thanks to the following individuals for making this thesis not only a possibility but a truly enriching adventure.

- Thanks to Dr. George Heineman, for making this thesis possible and providing at every step, thoughtful guidance and deep insight into the problem at hand.
- Thanks to Dr. Jan Bessai, for the many hours of intercontinental collaborative coding and his brilliant solutions to many a code generation dilemma.
- Thanks to Dr. Rose Bohrer, for providing insightful and valuable feedback on the final drafts of this thesis.
- Thanks to Dr. Elvis Foster, for laying the foundations of my software engineering knowledge and making grad school a possibility.
- Thanks to my family, for the ludicrously low rent, and unending support and encouragement.
- Thanks to Jennifer, for keeping me grounded, and expressing many a reminder to eat and sleep amidst pressing deadlines.
- Thanks to Antony and Kenny, for tolerating my frequent over-sharing of thesis-related topics.

Chapter 1

Introduction

For the last 50 years, Program Synthesis – the process of automatically generating useful source code from a high-level specification – has remained a highly active and promising field of study in Computer Science. Some researchers have declared that robust and efficient program synthesis is "the holy grail" of Computer Science. [8]. Indeed, program synthesis has the potential to revolutionize the manner and efficiency with which software is developed and maintained. One channel through which such a feat may be achieved is the acceleration of certain software engineering tasks which tend to exhibit a large degree of development overhead while only indirectly contributing to the immediate problem solution.

In this thesis, we investigate how to apply program synthesis to the generation of *Boilerplate code*, "Sections of code that are repeated in multiple places with little to no variation" [4]. The primary contribution of this thesis is the formulation and execution of three case studies demonstrating the generation of a notable amount of boilerplate code in three real-world scenarios. Synthesis is achieved using an existing synthesis framework developed by George Heineman and Jan Bessai called CoGen. In addition, we present a novel methodology designed to aid in the effective integration and application of CoGen within existing development processes and workflows. We evaluate our results through a reflective written analysis of the development efforts involved in each of our proposed case studies.

1.1 Problem Statement

As mentioned previously, boilerplate has been described as "Sections of code that are repeated in multiple places with little to no variation" [4]. For the purposes of this thesis, we will extend this definition to include code *not directly relevant to the functional requirements of a software system* but instead, non-functional requirements or constraints placed on the system by its environment. We claim that boilerplate code introduces additional overhead into the software development process in three key ways. Firstly the presence of boilerplate code introduces what Fred Brooks calls "accidental complexity" into a codebase, complexity not related to the problem at hand but instead one or more auxiliary concerns. This additional system complexity increases the difficulty of reasoning over a given codebase [6]. Secondly, boilerplate code tends to obscure the underlying business logic codified by a given system thus increasing the difficulty of system adaption to new functional requirements. Thirdly, boilerplate code can induce a greater degree of coupling with external dependencies such as libraries, services or platforms, thus locking a system into a particular context that may or may not support future system requirements. In concert, the stated effects incur friction in the software development process thus leading to longer development times.

In order to ground the argument made above we now illustrate three common sources of boilerplate in everyday code and explain how each example contributes to increased development

overhead. The first type of original point occurs in the following scenario. Imagine system A leverages another system B, however, system B was designed to solve a number of additional problems beyond those of interest to system A. In this case, additional boilerplate must be written to tailor system B to the needs of system A. Even worse, system A may even be forced to integrate directly with the unnecessary functionality of system B. Generally, the more problems a supporting system solves, the more boilerplate is required to properly configure the supporting system for a particular use case.

The code in Figure 1.1 intends to access GPS information on an Android device as shown in Figure 1.2. The engineer is forced to fully engage with the Android access control subsystem even though they do not intend on publishing their application.

```
public class MainActivity extends AppCompatActivity {
  private static final int LOCATION_REQUEST_CODE = 1000;
  private FusedLocationProviderClient fusedLocationClient;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);

      fusedLocationClient = LocationServices.getFusedLocationProviderClient(this);

      // Check for permissions
      if (ActivityCompat.checkSelfPermission(this,
          Manifest.permission.ACCESS_FINE_LOCATION) !=
          PackageManager.PERMISSION_GRANTED &&
             ActivityCompat.checkSelfPermission(this,
                 Manifest.permission.ACCESS_COARSE_LOCATION) !=
                 PackageManager.PERMISSION_GRANTED) {
          // Request permissions
          ActivityCompat.requestPermissions(this, new
              String[]{Manifest.permission.ACCESS_FINE_LOCATION,
              Manifest.permission.ACCESS_COARSE_LOCATION}, LOCATION_REQUEST_CODE);
      } else {
          getLastLocation();
      }
  }
  @Override
  public void onRequestPermissionsResult(int requestCode, @NonNull String[]
      permissions, @NonNull int[] grantResults) {
      switch (requestCode) {
          case LOCATION_REQUEST_CODE: {
              if (grantResults.length > 0 && grantResults[0] ==
                  PackageManager.PERMISSION_GRANTED) {
                  getLastLocation();
              } else {
                  Toast.makeText(this, "Location permission denied",
                      Toast.LENGTH_SHORT).show();
              }
              break;
          }
      }
  }
}
```

Figure 1.1: Sample Boilerplate code

```
public class MainActivity extends AppCompatActivity {

  private void getLastLocation() {
    if (ActivityCompat.checkSelfPermission(this,
        Manifest.permission.ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED &&
               ActivityCompat.checkSelfPermission(this,
                   Manifest.permission.ACCESS_COARSE_LOCATION) !=
                   PackageManager.PERMISSION_GRANTED) {
           return;
       }
    fusedLocationClient.getLastLocation()
      .addOnSuccessListener(this, new OnSuccessListener<Location>() {

       @Override
       public void onSuccess(Location location) {
         if (location != null) {
            Toast.makeText(MainActivity.this, "Lat: " + location.getLatitude() + ", Lon:
                " + location.getLongitude(), Toast.LENGTH_LONG).show();
         }
       }
    });
  }
}
```

Figure 1.2: Desired GPS Location Method

The purpose of this example is not to undermine the importance of security enforcement, it is instead to demonstrate how additional functionality (access control) offered by a supporting system (i.e., the Android Runtime system) increased the complexity of primary system implementation (i.e., the app).

Another key type of boilerplate is code required to integrate with a particular framework or software library. Figure 1.3 contains a concrete example of such boilerplate in the context of the Swing GUI framework.

```
public class SimpleApp {
    private JFrame frame;
    private JPanel panel;
    private JButton button;
    private JTextField textField;
    private JLabel label;

    public SimpleApp() {
        frame = new JFrame("Simple Swing App");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        panel = new JPanel();
        frame.add(panel);
        placeComponents(panel);
        frame.setVisible(true);
    }

    private void placeComponents(JPanel panel) {
        panel.setLayout(null);

        label = new JLabel("Enter text:");
        label.setBounds(10, 20, 80, 25);
        panel.add(label);

        textField = new JTextField(20);
        textField.setBounds(100, 20, 160, 25);
        panel.add(textField);

        button = new JButton("Click me!");
        button.setBounds(10, 80, 120, 25);
        panel.add(button);

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                label.setText(textField.getText());
            }
        });
    }
}
```

Figure 1.3: Swing Boilerplate Code

Here the engineer must be cognizant of multiple arbitrary aspects of the Swing GUI framework not related to the goal of simply drawing a text input, label and button on the screen.

One final example of boilerplate is code encountered when interfacing with an external system via a specific communication protocol. Figure 1.4 is an example related to communicating with an SQL server in C++. A significant amount of boilerplate code is required to initialize connectivity with the database server as well as manage connections and connection failure modes.

```
#include <iostream>
#include <mysql_connection.h>
#include <driver.h>
#include <exception.h>
#include <resultset.h>
#include <statement.h>

int main() {
    sql::Driver* driver;
    sql::Connection* con;
    sql::Statement* stmt;
    sql::ResultSet* res;

    // Non-boilerplate BEGIN **********
    const std::string host = "tcp://127.0.0.1:3306";
    const std::string user = "your_username";
    const std::string password = "your_password";
    const std::string database = "your_database_name";
    // Non-boilerplate END **********

    try {
        driver = get_driver_instance();
        con = driver->connect(host, user, password);
        con->setSchema(database);
        stmt = con->createStatement();

        // Non-boilerplate BEGIN **********
        res = stmt->executeQuery("SELECT 'Hello, world!' AS _message");

        while (res->next()) {
            std::cout << "\t... MySQL replies: ";
            std::cout << res->getString("_message") << std::endl;
        }
        // Non-boilerplate END **********

        delete res;
        delete stmt;
        delete con;
    } catch (sql::SQLException& e) {
        std::cerr << "# ERR: SQLException in " << __FILE__;
        std::cerr << "(" << __FUNCTION__ << ") on line " << __LINE__ << std::endl;
        std::cerr << "# ERR: " << e.what();
        std::cerr << " (MySQL error code: " << e.getErrorCode() << ", SQLState: " <<
            e.getSQLState() << " )" << std::endl;
    }

    return EXIT_SUCCESS;
}
```

Figure 1.4: C++ Boilerplate Code

As you can see, boilerplate can quickly arise in a number of varied but common software development situations. Additionally, it is not uncommon for a software system to involve many such situations at once thus you can see how many projects may collect a non-insignificant volume

of boilerplate code over time.

We now discuss two key insights related to boilerplate code that we believe provide direct insight into a problem solution. Firstly, one may arrive at the observation that "One person's boilerplate is another person's business logic". If one wishes to eliminate a certain fragment of boilerplate without modifying the problem statement to no longer require said boilerplate, one must either integrate the problems solved by the boilerplate-requiring system into the immediate problem statement or they must swap out the supporting system for another supporting system. In the first case, the problem statement will likely need to be reduced in scope and complexity due to a reduction in supporting functionality. In the second case, the new supporting system will likely introduce its own boilerplate. We (humorously) refer to this property as "the law of conservation of boilerplate". In essence, the functionality of a system must be realized by some party. Such an observation suggests that the problem of boilerplate overhead reduction is best handled through progressive and systematic augmentation / automation of boilerplate-related concerns.

The next key insight is the realization that boilerplate of a particular type should, by definition, remain relatively uniform across occurrences within a single codebase or across multiple codebases. For example, the initialization code for a particular framework will likely be similar across all codebases leveraging the framework with slight variation related to configuration and/or degree of usage. Naturally, boilerplate code will exhibit some degree of variation depending on the usage context however in general we believe said variations will be predictable and less dramatic than problem-specific code. Both stated insights suggest the applicability of program synthesis to automating some degree of boilerplate development and maintenance. Figure 1.5 illustrates the stated property of boilerplate.

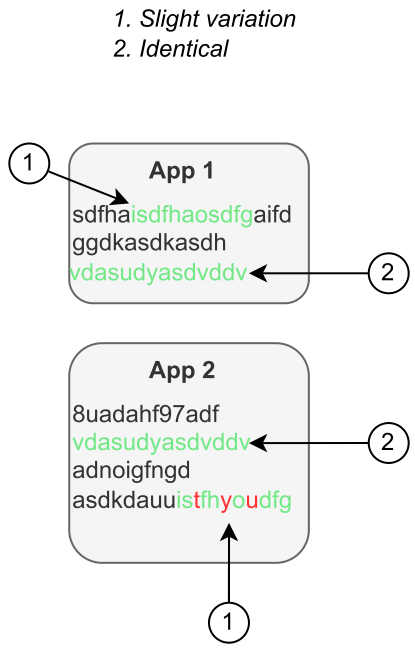

Figure 1.5: Illustration of boilerplate overlap and variability

1.2 Motivation

With a characterization of the boilerplate problem in hand we now consider a number of factors motivating our usage of program synthesis as a solution to the boilerplate problem.

First and foremost, iteratively applied program synthesis would allow for the gradual automa-

tion of boilerplate-related concerns one code fragment at a time. We identified such a capability as important earlier when we showed that one cannot simply eliminate boilerplate code without sacrificing functionality or transferring ownership of supporting system concerns. Secondly, the bounded variability of boilerplate code lends itself to generation via a model-driven approach. Here a model could be designed to encapsulate potential degrees of variation in boilerplate code. This model could then be configured differently depending on the problem thus allowing generated boilerplate code to adapt to a particular application or usage context.

In addition to the motivating factors above, we also present a number of additional factors we believe provide strong motivation for our chosen approach.

Automatic Generation of Common Patterns Boilerplate code need not refer only to high level concerns such as framework initialization etc, but also extremely granular patterns of code which are used frequently. For example, using a lightweight specification of object types, one could easily generate (and automatically maintain) factory classes associated with certain families of classes. Perhaps even more usefully, one could use code generation to assist in the maintenance of systems leveraging the Visitor pattern. When using the Visitor pattern, if a new Visitable type is introduced, all Visitors must be updated to reflect the handling of the new Visitable type. Updating concrete visitors could be easily automated using code generation.

Software Product Lines A software system must constantly evolve to facilitate newly discovered requirements. One issue with this process of extensional evolution however is that overtime, as scope increases, the complexity of a singular codebase can begin to increase rapidly as compromises are made to support a growing number of coexisting requirements within the same codebase. In order to slow down this combinatorial explosion of complexity, we argue that using code generation, one could synthesize entirely separate codebases which address certain subsets of requirements. One benefit of this approach is that often requirements are driven by particular user usecases/client contexts. This parallel synthesis has the effect that each individual codebase may be much simpler due to reduction in the number of interactions between requirements.

Inlining of Important Information Code generation allows the insertion of important data values at ANY location in the code including specifically at the location where the data is most relevant/important. For this reason, we are able to directly hardcode important information at relevant locations in code without needing to worry about the traditional drawbacks of value hardcoding. Said drawbacks are avoided as the information is still properly consolidated in the domain model.

Decoupling of Implementation Architecture from Domain Model Architecture Program synthesis enables the complete decoupling of domain-level concerns (the what) from implementation-level concerns (the how). Such a decoupling also allows the generation of multiple implementations each with its own architecture. For example, one architecture could favor performance while another readability or even education.

Another benefit of such decoupling is that code may be flatter in structure without becoming overly complex. This is because codification of critical system abstractions and concepts are offloaded to the domain model thus freeing up the generated code for tuning and specialization with respect to implementation-specific concerns. If the user wishes to understand the design they need only view the domain model. If the user wishes to understand the implementation they can look at the generated code while using the domain model as a mental model.

Language and Runtime Independence Using program synthesis, the engineer is capable of generating code for any target language or runtime from the same code specification. This

language/runtime independence is immensely useful, especially with regard to integrating newer systems with older systems.

Eliminating Dependencies on Deprecated Code Program synthesis solves the problem of code deprecation for free by allowing the extension of the domain without affecting code generated with respect to older domains (see expression-problem). One can simply synthesize n versions of a given software system (each separate from the other)

Practically this means that deprecated code need not interfere with new code i.e new code can evolve without concern for depreciated code as the older system can evolve in parallel as its own "product line".

Security Through Variation By generating entire systems programmatically we are able to inject perturbations during code generation with the property that program functionality remains invariant while program structure does not. Such an approach could greatly aid in software security as hackers would need to potentially reverse engineer a large number of system variations in order to realize exploits that work generally. [Credit for this idea goes to Prof George Heineman]

Performance Program synthesis has the potential to eliminate the performance cost of abstraction through decoupling of domain and implementation architecture. Since implementation code is a projection of the domain model (see Domain Driven Design) then the specific implementation details of a software system may vary independently of the domain. In other words, implementation code can be optimized with respect to the underlying platform while higher level conceptualizations/abstractions remain safely codified in the domain model, and thus remain safe from the complexity of one or more particular implementations.

Synthesis as Documentation By examining what aspects of the domain model produce which source code the engineer may more easily determine the purpose/role of a given source file via direct examination of actual source code.

We hope the reader now has a better understanding of the problem we are attempting to solve and motivation for our chosen solution path involving program synthesis. In summary, Boilerplate code is common and its presence increases development overhead, but unfortunately boilerplate remains a necessary evil and thus cannot be eliminated all at once. From this position, we posit automation of boilerplate concerns via program synthesis as a promising solution to the stated problem.

1.3 Background

Now that we have defined the problem at hand, let us discuss the existing body of work surrounding program synthesis and how our chosen technology fits into the big picture.

At a high level, the goal of program synthesis is to algorithmically generate useful source code as a function of a human-readable specification and/or model. Over the years many different approaches have been explored to realize this core idea, each approach with its own tradeoffs and capabilities. Approaches such as deductive synthesis and syntax-guided synthesis require that the user construct a highly detailed and logically rigorous model. This model is then used to infer what code should be generated as a consequence of model constraints. More relaxed approaches to program synthesis such as inductive synthesis or program sketching allow the user to partially specify or outline the general structure of the code they would like to generate and then allow the synthesis engine to fill in certain sections of missing code automatically. Note that such approaches may still leverage models, however, these models are less comprehensive and focus on supporting the code generation process rather than driving it entirely.

Figure 1.6 illustrates the span of synthesis approaches with respect to both the degree of explicit code specification as compared to degree of inference and also the computational intensity of various approaches.

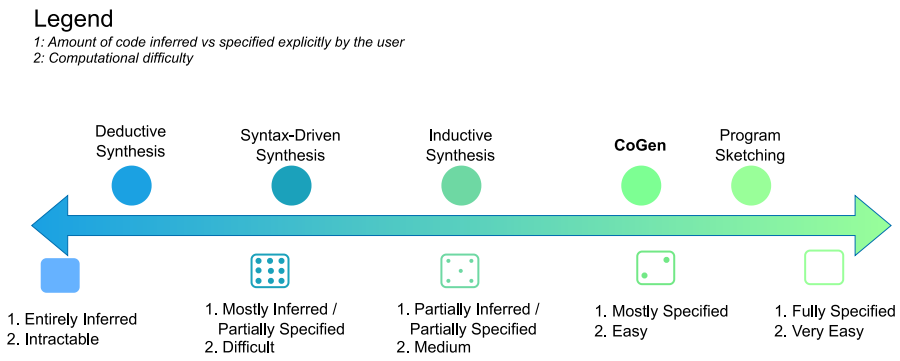

Figure 1.6: Span of synthesis approaches

Let us now discuss the computational difficulty aspect of each approach. Deductive approaches tend to be computationally difficult due to the vast space of possible programs which must be explored during model evaluation. This computational difficulty limits the size of problems that can be generated using such approaches generally pushing such approaches to more domain-specific applications. Conversely, inductive approaches are often far less computationally intense and thus lend themselves better to generating larger volumes of code. The synthesis technology chosen for this thesis (CoGen) falls towards the program sketching side of the axis. Under CoGen the structure of generated code is mostly specified in a language agnostic manner. A model is then referenced by the code specifications to determine how to generate certain restricted parts of the desired code artifact. Such an approach avoids the tractability issues caused by more inferred approaches thus allowing our solution to more easily generate a large degree of varied code.

Another key consideration when evaluating program synthesis technologies is the degree to which desired output is outlined explicitly in a specification vs inferred from an abstract model. In a purely model based or inferred approach models are constructed using highly structured modeling formalisms such as a formal logic. Such formalisms are powerful and allow one to make a number of useful claims about generated code but can be difficult to adapt to newly encountered requirements due to their inherent rigorousness. On the other hand, inductive and program sketching approaches involve less rigid specifications and thus tend to facilitate easier adaption. CoGen chooses to use a less rigid approach similar to program sketching, in fact CoGen does not make use of any form of specialized modeling or specification formalisms. Under CoGen code specifications and models are formulated in a traditional programming language namely Scala. Specifications may freely reference models to inform code generation decisions, however, model usage is not mandatory. This design decision allows CoGen to easily scale well to the intricate and varied demands of boilerplate code structure.

In summary, we believe CoGen achieves a good balance between sufficiently high-level specification of the generation process while still allowing a sufficient degree of control over the structure of generated code. These two properties are extremely desirable given the wide array of forms boilerplate code may take.

1.4 Proposed Solution

We leverage in this thesis an existing code generation technology called CoGen [2] which is specifically designed to allow for the maximum degree of flexibility in code generation. Even the best

hammer is useless if used as a shovel, hence we also introduce a novel *Coding, Distillation, Modeling, Synthesis* (CDMS) methodology designed to aid in the application of program synthesis to the synthesis of boilerplate in complex and varied codebases. Our methodology is also designed to facilitate coexistence of program synthesis with existing development methodologies. We hope that in concert the selected synthesis technology and proposed methodology will reduce, or in some cases eliminate entirely, the overhead of boilerplate code development and maintenance thus vastly accelerating software development efforts.

In summary, our approach must fulfill the following key requirements.

Technology Requirements

- Must support generating code in multiple domains, languages and programming paradigms
- Must avoid reliance on restrictive modeling formalisms.
- Must support specification of both model and code structure.

Methodology Requirements

- Must easily integrate synthesis with existing engineering processes
- Must reduce learning curve of effective approach usage
- Must be interoperable with existing software engineering methodologies

1.5 Method of Evaluation

To evaluate the effectiveness of CoGen technology and our *Coding, Distillation, Modeling, Synthesis* (CDMS) methodology, we apply both to three real-world case studies. The first case study exemplifies the automated synthesis of boilerplate GUI code to support the use of three application development frameworks namely, JavaFX, QT and LibGDX. The second case study demonstrates the comprehensive synthesis of supporting boilerplate code in a robotics framework known as ROS. Here we show how with little effort, the engineer can automatically synthesize boilerplate code required to deploy existing logic in a feature-rich ROS environment. The third case study demonstrates the synthesis of low-level Vulkan rendering code in a non-invasive manner wherein the engineer specifies exactly where in their existing code they would like synthesized code to be placed. This approach demonstrates how the engineer may otherwise maintain full control over codebase design while still benefiting from targeted boilerplate generation.

Chapter 2

Approach

Our approach to solving the problem of boilerplate code generation is two-fold. The technical aspect of our approach, namely code generation is realized using an existing program synthesis framework called CoGen. The issue of CoGen usage is addressed through a novel methodology we call CDMS. In essence, CDMS is designed to guide the developer in the application of CoGen to varied real-world codebases with the aim of gradually synthesizing boilerplate code present therein.

Before we begin a detailed discussion of both methodology and synthesis technology, we must define two key concepts that will play a significant role in both concerns. The concepts in question are that of Domain Model and Synthesis Specification.

Domain Model We derive the concept of Domain Model from the Domain Driven Design literature championed by Eric Evans [24]. Evans defines a domain as "A sphere of knowledge, influence, or activity". The subject area to which the user applies a program is the domain of the software. Further Evans defines model as "A system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain" [25]. In the context of this thesis, we specialize Evan's definition slightly by adding the constraint that "selected aspects" refer to select aspects of an existing codebase that pertain to "solving problems related to the domain". The discussion of the modeling phase of CDMS will go into further detail on how exactly to select these problem domain-relevant aspects.

As it pertains to our problem of boilerplate synthesis, the purpose of a domain model is to inform the process of boilerplate generation, thus allowing generated code to vary in accordance with the requirements of a specific problem. In short, code generators use the domain model as a reference when making key decisions regarding code structure or what code should be generated. By mutating the domain model configuration, an engineer may thus ensure generated code aligns with immediate problem requirements. Such control allows truly generalized and scalable boilerplate generation to any number of problem-specific contexts.

Synthesis Specification With the concept of domain model in hand, it is now time to define the concept of a synthesis specification. A synthesis specification is simply some form of specification that describes how to generate executable code from a particular domain model. A synthesis specification may be realized as a textual description, diagram, or in the form of code in some programming language. In this thesis, we will leverage both diagrammatic and code-based realizations.

2.1 Technology

We now turn our attention to a detailed description of CoGen, our chosen synthesis technology.

2.1.1 Background

CoGen is a specific part of a larger system developed by George Heineman and Jan Bessai called EpCoGen. EpCoGen was originally developed to assist in research involving the expression problem. The expression problem, although out of scope for this thesis, touches upon the fascinating problem of determining how to add new functionality to a codebase, (either through the addition of datatypes or operations), without requiring the modification of existing code. For the purposes of this thesis, we do not require the expression problem-related functionality (Ep) of EpCoGen thus we make use of only its code generation capabilities (CoGen).

At its core, CoGen is a powerful and generalized synthesis engine capable of generating arbitrarily complex code in multiple languages and programming paradigms. In this thesis, we adapt CoGen to the problem of boilerplate generation using the CDMS methodology. In addition, we made a number of small modifications to CoGen to enable the generation of additional code structures such as try/catch statements and importing of existing code into the code generation process.

CoGen is built around the following requirements.

- One should be capable of generating code in such as way as to allow the reuse of a given generation process in multiple situations. For example, the generation of getters and setters across multiple classes.
- All generated code must be bound to a particular context. This context shall be amended and evolve during the generation process until the final desired context state is achieved.
- The process of code generation must occur in a transaction-based manner wherein a set of transitions, each responsible for generating some fragment of code, are emitted and then later evaluated by the generator.

A key motivation for choosing CoGen as the technological backbone of our approach was its lack of assumptions regarding the underlying domains, implementation architectures, or languages a user wishes to work within. Under CoGen domain models and synthesis specifications are realized in the form of vanilla Scala and do not make use of any special formalisms such as domain-specific languages or obscure language extensions. The result of this design choice is the complete elimination of arbitrary restrictions on how the engineer may realize code synthesis to solve a specific problem.

In the following section, we provide a detailed technical overview of CoGen including a description of its architecture and capabilities.

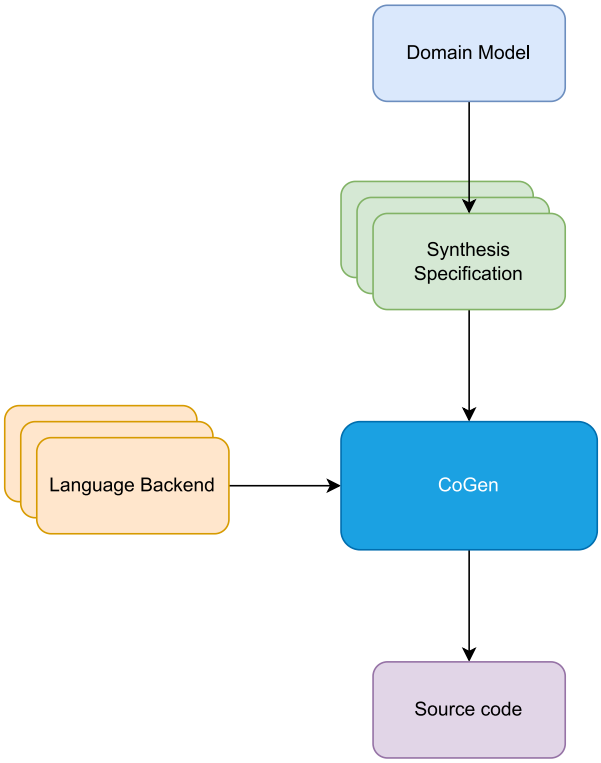

Figure 2.1: System context diagram illustrating CoGen inputs and outputs

2.1.2 Architecture

CoGen is built on a 3-Tier layered architecture. The first layer of the CoGen system is known as the *Implementation Provider* layer. It is the responsibility of this layer to facilitate the creation of user-defined code generation specifications and domain models. Layer 2 is called the *Code Generation* layer. This layer contains abstract codifications of common programming paradigms and capabilities. These capabilities are used by layer 1 synthesis specifications. Finally, layer 3 is called the *Language layer* and contains implementations of layer 2 capabilities/paradigms in a particular target language such as Java or C++.

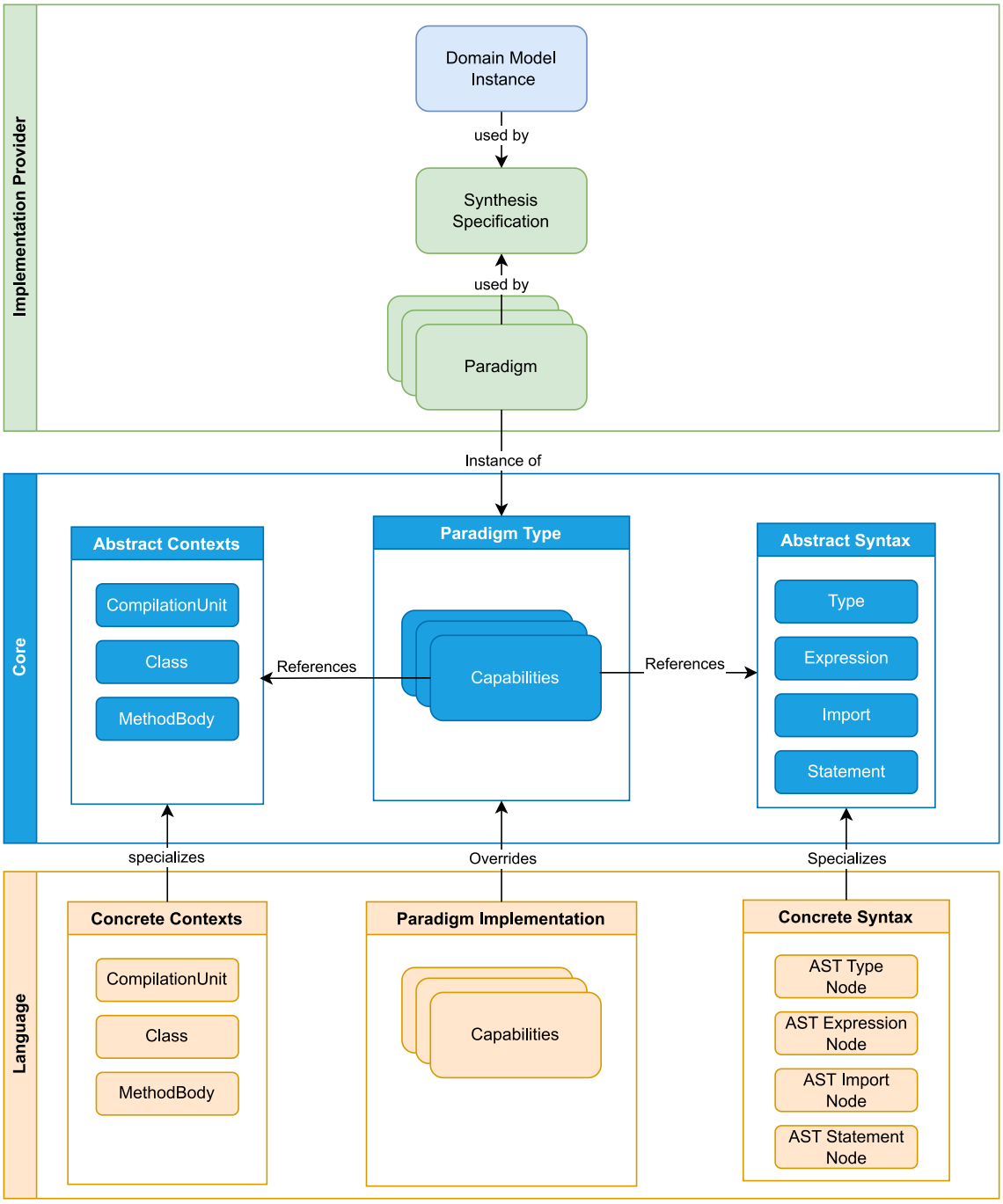

Figure 2.2: Simplified system structure diagram illustrating the layered architecture of CoGen

CoGen synthesis specifications are built around the concepts of a Generator and a Context. Any synthesized line of code is produced by a Generator in some Context. Upon instantiation, Generators are parameterized by a Context and some Abstract Syntax type representing the type of artifact produced by the generator. This approach allows for the powerful assertion that any generated code may be traced back to a specific generator executed in a specific context at any level of granularity. Additionally, the design allows for the introduction of useful extension points

which enable generators with the same role to produce slightly different code depending on the Context in which they are created.

In CoGen the type of code that can be generated is codified in the form of paradigms where each paradigm contains a set of one or more capabilities. Examples of paradigms include Imperative, Object Oriented, or Functional. Examples of specific paradigm capabilities include generation of Classes, adding methods or fields to classes in the case of OO or declaring or assigning variables in the case of Imperative. It is important to note that paradigms and associated capabilities are not always mutually exclusive. This allows for more effective factoring and code reuse across paradigms. One example of multi-paradigm use would be that of Object-oriented and Imperative as naturally, Object Oriented programming is largely an extension of the imperative paradigm. Now when specifying a paradigm capability, each capability must specify the context in which the capability may be applied. For example, one may only add a method to a class if they are in a class context.

CoGen synthesizes code in a two-pass manner. In the first pass, user-defined synthesis specifications contained in implementation providers instantiate any number of Generators. Generators can be instantiated in a nested manner. For example, a given Generator may be instantiated to generate a class declaration, but the code which instantiated the Generator may also instantiate additional Generators to, for example generate class methods. Given such composition potential, generators are represented internally by CoGen as an n-tree data structure. Usually, the root node of a generator tree consists of a project context. A project context may have one or more CompiliationUnit context children. Each CompilationUnit may have one or more Class or MethodContexts and finally a Methodbody context may contain code statements.

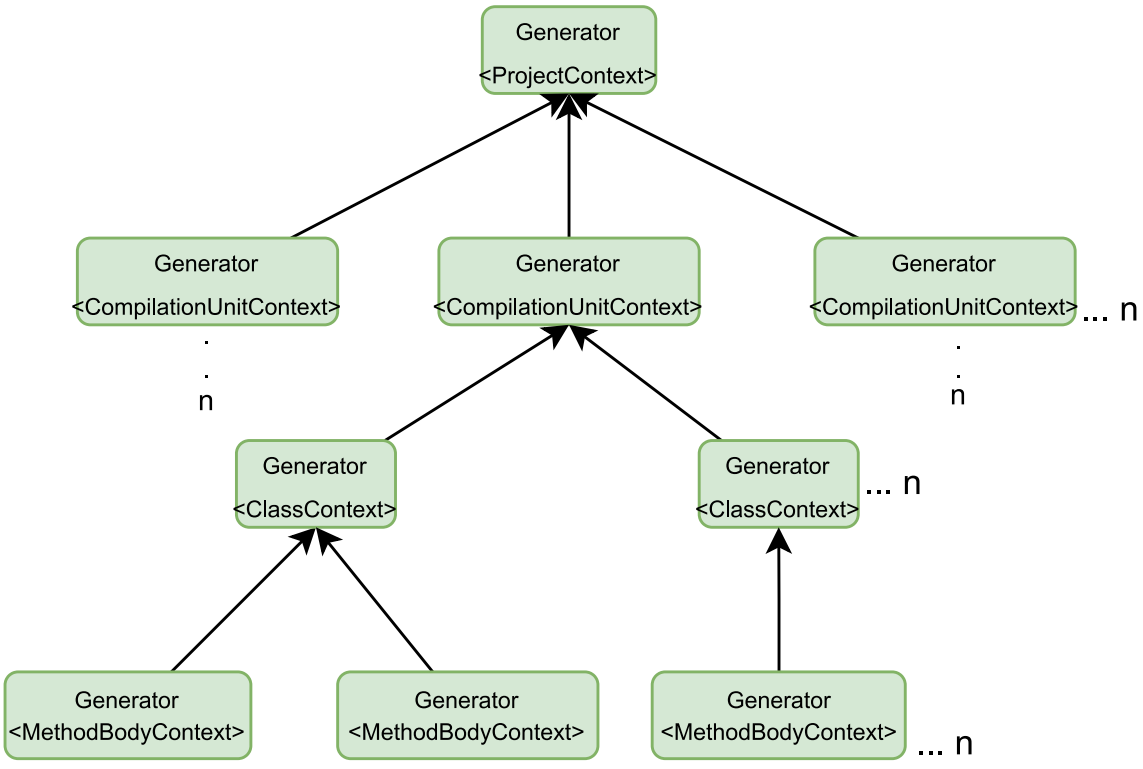

Figure 2.3: Illustration of generator tree organization

After the generator tree has been built, a second generation pass traverses the tree and executes generators in depth-first order. Upon generator execution, control flow transfers to the overridden

generation functionality in the language layer. The language layer code then constructs desired AST structure and mutates the context assigned to the generator as necessary. (Usually said updates involve appending newly generated AST structures to existing AST structures) Let us consider an example. Let us say we wish to generate a class method. Firstly we assume AST information for the body of the class method has previously been generated. We can make this assumption as the method body generator is a child of the class method generator and thus must have been executed previously in accordance with depth-first traversal. All that remains for the class method generator then is to append the method body abstract syntax subtree to the class AST stored in the generator's associated class context.

After all generators have been executed, the final AST tree will reside in the context object associated with the root generator in the generator tree. This AST tree can then be trivially written to disk as source code using language-specific pretty printing functionality.

Now that we have discussed the basic architecture of CoGen, we will, in the following sections focus on the usage of CoGen by providing a brief overview of how Domain models and synthesis specifications may be constructed in the framework.

2.1.3 Domain Models in CoGen

In CoGen, the user constructs domain models using Scala code. We suggest building domain models in an object-oriented fashion thus increasing the ease of conceptual modeling. A domain model should contain a primary class (usually called Domain or DomainBase) which is responsible for instantiating particular concepts and connecting domain model objects together as required. Conceptually, this primary class represents a particular configuration of the domain and is the key artifact passed to generators. In other words, generators reference the domain model foundation object for runtime information about domain model configuration. Fundamentally, although we provide the suggestions above, the only requirement on domain model design is that a domain model be structured in such a way that it can be easily passed into the constructor of one or more implementation providers. For examples of domain models see any of the three case studies.

2.1.4 Synthesis Specifications in CoGen

We now consider how to construct synthesis specifications that are consumable by CoGen. As mentioned previously, CoGen takes as input a domain model and one or more implementation providers and produces code in a desired target language. With regards to terminology, an implementation provider is simply a code-based realization of a synthesis specification that can be used by CoGen to generate source code. For the remainder of this thesis, we will use the two terms interchangeably. Implementation providers take the form of Scala code and are responsible for instantiating generators. In order to understand how a user goes about creating synthesis specifications we will explore a simple example. The code in 2.4 consists of a number of methods for generating the various components required to produce a program that generates the nth-Fibonacci number and prints it to the screen. The example also produces a unit test for testing the resultant program however we will focus our attention primarily on the Fibonacci related code.

```
def make_fibonacci(): Generator[MethodBodyContext, Expression] = {
  for {
    /** Method will have single n:Int parameter and return Int. */
    intType <- toTargetLanguageType(TypeRep.Int)
    _ <- setParameters(Seq(("n", intType)))
    _ <- setReturnType(intType)

    /** Get arguments to function and constant values. */
    one <- reify(TypeRep.Int, 1)
    two <- reify(TypeRep.Int, 2)
    args <- getArguments()

    /** Logic for whether n <= 1. */
    (name,tpe,n) = args.head
    le1 <- ffi.le(n, one)

    /** Form two subexpressions (n-1) and (n-2). */
    n_1 <- ffi.sub(n, one)
    n_2 <- ffi.sub(n, two)

    /** Apply twice, to fib(n-1) and fib(n-2); add together. */
    func <- findMethod(Seq(fibName))
    fn_1 <- apply(func, Seq(n_1))
    fn_2 <- apply(func, Seq(n_2))
    addExpr <- ffi.add(fn_1, fn_2)

    res <- ifThenElse(le1,
               Command.lift(n),       /** If (*@$n \leq 1$@*) */
               Seq.empty,             /** Empty "else if" */
               Command.lift(addExpr)) /** Else fib(n-1)+fib(n-2) */
  } yield res
}

def make_unit(): Generator[paradigm.CompilationUnitContext, Unit] = {
  for {
    _ <- addMethod("fib", make_fibonacci())
  } yield ()
}

def make_project(): Generator[paradigm.ProjectContext, Unit] = {
  for {
    _ <- addCompilationUnit(make_unit(), fibName)
    _ <- addCompilationUnit(addTestSuite("fibTest", make_test()), "fibTest")
  } yield ()
}
```

Figure 2.4: EpCoGen Fibonacci Implementation

Firstly we will make some general points regarding the syntax and semantics of the example. We will then perform an execution flow analysis of the example starting from the last method make_project and moving upwards. For more information on the specifics of the Scala programming language see [36].

Our first point involves the usage of `_ <- ...` and `<var> <- ...` syntax and associated for-comprehension. In CoGen, each method call within a for block returns an instance of a Generator

however in the case of `<var> <- ...`, the scala compiler will set `<var>` equal to the evaluated return type the generator. The underlying generator instance is still stored in the comprehended list but now the user may use an abstract syntax object as input to future instances generator. One important fact about abstract syntax objects (`<var>` in our example) is that they in fact act as handles to a concrete object which will be realized as an actual AST node during phase-2 of CoGen execution. In this sense `<var>` acts as a promise that at some point in the future, the concrete representation of the abstract syntax object will be realized by a generator implementation in the form of an AST node.

Now let us examine the example from the perspective of execution flow. First and foremost the make_project method creates a ProjectContext and then calls addCompilationUnit. addCompilationUnit tells CoGen to associate the results of executing the generators returned by make_unit and make_test generators with the current project context. Next make_unit tells CoGen to add a new method to a compilation unit where the body of this method consists of code generated by the generator instantiated by make_fibonacci. Note how unlike in previous methods, this method returns an Expression type. A non-Unit return type simply means that the results of running a given generator can be used as input to later generators. Additionally, a value of unit means that the generator just updates its parent context and does not return a particular artifact. We now turn our attention to the final method make_fibonacci. In this method, the statement of the Fibonacci recurrence are generated.

Presenting synthesis specifications in the form of source code can be difficult due to their large size and complexity. For this reason, we present a novel diagramming technique for the sole purpose of better expressing complex synthesis specifications to the reader. This alternative representation becomes incredibly important when we turn our attention to the case studies later in this thesis. We call our diagram type a Synthesis Tree. Below is a synthesis tree corresponding to the Fibonacci example.

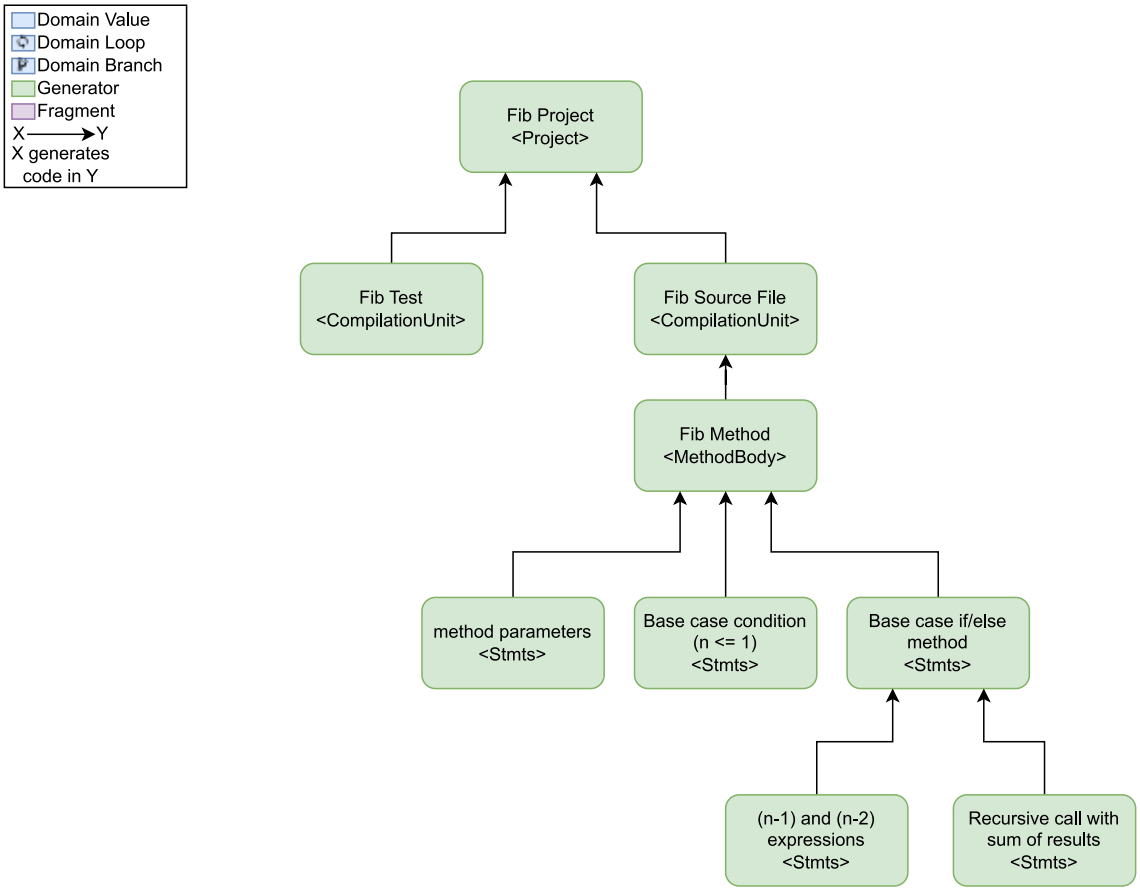

Figure 2.5: Synthesis tree for Fibonacci example

The proposed diagram type involves four additional figure types that are not present in the example above. Below is a synthesis tree from the Robotics case study which contains examples of these concepts.

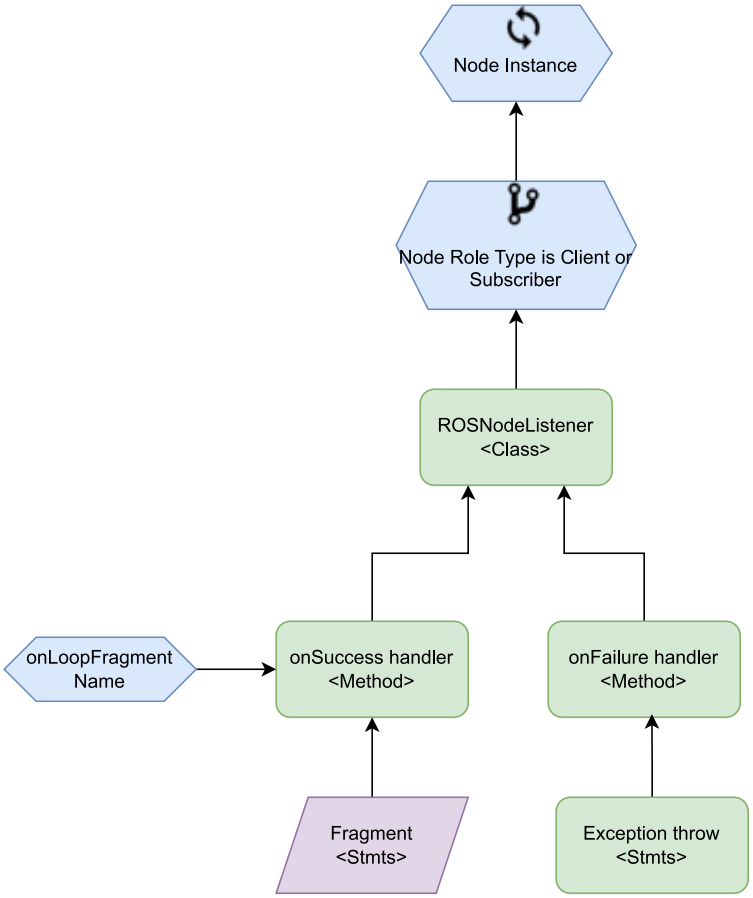

Figure 2.6: More sophisticated synthesis tree

Here the blue figures indicate aspects of generation that are directly dependent on the domain model. A blue figure with a loop icon indicates that the synthesis subtree of the blue figure is evaluated once for each object in a given domain model set. The blue figure with a branch icon indicates that depending on some domain model dependent logical condition, different code is synthesized. The blue figure with no icon indicates the direct usage of a domain model value by the associated generator. Finally, the purple node indicates usage of a fragment of code provided "as is" to the generator. The generator may modify the provided code (i.e extend or remove code from the fragment) or inject the fragment into the set of final project artifacts without modification.

2.2 Methodology

2.2.1 Background

To bring the benefits of CoGen to as many software engineering contexts as possible, we propose a low-friction iterative methodology to guide the application of CoGen to boilerplate generation in many varied contexts. This decision was motivated by two key observations. Firstly, during early experimentation with CoGen we noted that if applied too aggressively, and at too large of an initial scope, the complexity involved in successfully synthesizing large amounts of code simultaneously would introduce compounding overhead (thus mitigating the overhead reductions provided by boilerplate-related generation). Second, we noted a significant learning curve associated with the usage of code generation in the general manner proposed. We decided then that in order

to become proficient at leveraging code generation, a large degree of practice was required and thus encouraging a methodological approach would aid the user in becoming familiar with our technology and its application in a guided and incremental manner.

We began the formulation of our methodology by first constructing an abstract description of the overall process. This initial description was inspired by a number of experimental applications of the CoGen synthesis technology and two existing engineering methodologies, Stepwise Refinement and Refactoring. With abstract specification in hand, we began refining our methodology by applying it, alongside our synthesis technology, to three real-world case studies. During the execution of each case study, we enriched our methodology with a number of discovered principles.

2.2.2 Definition

We now provide a more formal description of our proposed methodology. CDMS or (Coding-Distillation-Modeling-Synthesis) is an iterative bottom-up methodology designed to facilitate the leveraging of program synthesis alongside any existing software engineering process.

To apply CDMS, one starts with an existing codebase, anything from a large legacy system to early development prototype, and applies each stage of CDMS to the codebase in an iterative manner. With each iteration, information pertinent to the problem domain is extracted from the codebase, and areas of boilerplate code are identified as potential candidates for synthetic generation. Next, identified boilerplate code is generated using the proposed synthesis technology and substituted into the original codebase in place of the original handwritten code. Alternatively, at this stage, handwritten code may be extracted into so-called fragments and interleaved with generated code "as is". After a sufficient number of CDMS iterations, the engineer converges on two key artifacts, the domain model and synthesis specification.

We will now discuss three important properties of CDMS which together allow CDMS to realize our goal of seamless boilerplate generation in varied contexts. Firstly CDMS allows for a highly variable degree of synthesis. When the user first begins using CDMS, zero lines of code are generated. After a few iterations, a small percentage of code may be generated. At this point, one could continue applying the methodology and generate an even larger percentage of code (up to and including 100%). Since the goal of CDMS is to assist in the generation of boilerplate code, however, we expect that many projects will converge on non 100% synthesis due to the presence of intricate business logic. The key takeaway is that our methodology places zero constraints on the amount of code one is required to generate.

Another key property of CDMS is its lack of interference with existing development processes. CDMS operates as an extension of the code-writing process alone and thus makes no assumptions about higher-level design or task-planning processes. For example, in the context of the Waterfall development methodology, CDMS could operate in parallel with the implementation phase with zero impact on analysis or design. In the context of Agile methodologies, integration is even tighter as insights from CDMS may feedback into design tasks, however, as before, CDMS will primarily influence only the development of source code during each Sprint.

The final property we would like to mention relates to the interaction between CDMS and an individual developer's workflow. CDMS and CoGen take an augmentative approach to code generation wherein synthesized code may be continuously regenerated at any point in time. This allows developers to directly reference generated code in their manually written code at all times thus ensuring generated code and handwritten code remain forever in sync. In addition, under our approach, the structure of code generation is guided by the engineer thus allowing for the generation of highly readable code. These two facts mean that at any point in time, the engineer may stop using CoGen and simply leverage the most recent code generated without any dependency on CoGen. This approach differs from many traditional synthesis technologies wherein code is generated as a pre-process directly before compilation.

We now begin a detailed description of CDMS.

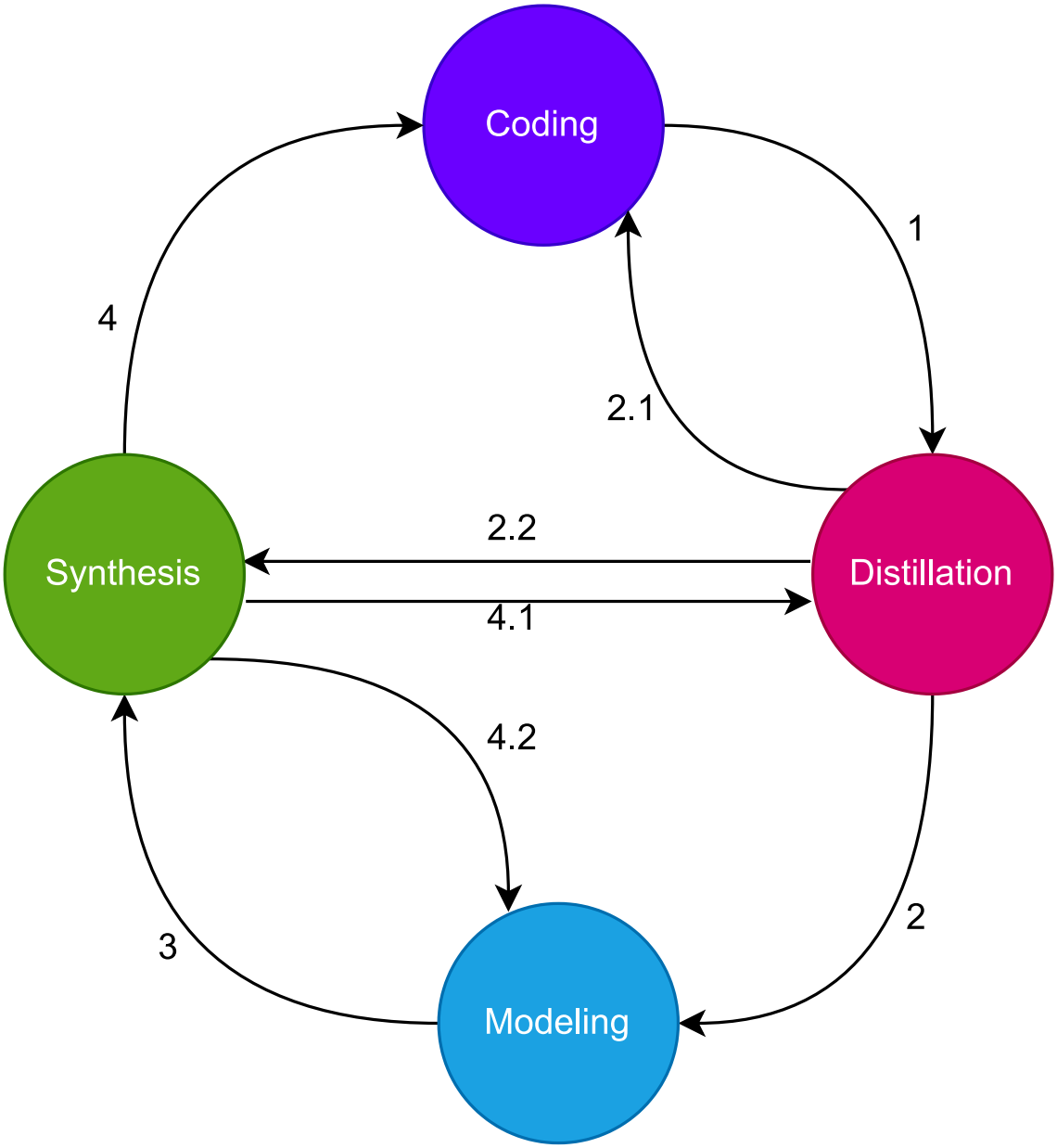

Figure 2.7: High-level illustration of the CDMS execution flow (execution begins in the "Coding" state)

As seen above, our methodology consists of four stages namely Coding, Distillation, Modeling and Synthesis. Stages are executed in the stated order however, Distillation may transition to Coding or Synthesis and Synthesis may transition to Distillation or Modeling. We will now discuss each methodology stage and describe how the stages interleave to facilitate convergence on a high-quality domain model and synthesis specification.

Coding

During the coding stage, source code is written by hand. Hand-written source code may interact with generated source code in this phase, otherwise, however, no additional code is generated.

From the coding phase the engineer may transition into the Distillation phase.

Distillation

While performing our case studies we found ourselves commonly making a number of specific types of observations. These observations proved extremely useful in identifying aspects of the codebase relevant to either the resultant domain model or synthesis specification.

Below is a list of encountered observation types. For each observation type, we provide a reference to an instance of the observation in a particular case study.

- The role of specific classes and resources in the codebase. "Rendering Case Study: Distillation - 1"
- The lifecycle of specific objects in the codebase. "Rendering Case Study: Distillation - 1"
- Methods specifically related to system lifecycle or the lifecycle of specific objects. Examples include methods responsible for initialization or cleanup. "Rendering Case Study: Distillation - 3"
- Fragments of code that consist almost entirely of business logic. Examples include specific algorithms, or business conditional logic. "Robotics Case Study: Distillation - 1"
- Variables, literal values or other parameters directly relevant to the business problem. Examples include names of business domain-related or GUI element names, filenames/resource paths, hardware configuration information. "GUI Case Study: Distillation - 1 - JavaFX"
- Code which is used purely to interact with the system environment i.e a library or API. "Rendering Case Study: Distillation - 3"

Given that observations tend to discard detail we decided to name the process of generating the list of stated observations, Distillation. It is important to note that the ability to successfully perform distillation is fundamentally a skill that is developed through practice and repeated encounters with specific patterns.

From the insights above we arrive at the following formal description of the Distillation phase.

During the distillation stage, the engineer analyzes existing source code and produces a set of observations. Observations should codify elements of the codebase that are directly related to the problem being solved in addition to high-level details involving the functionality and structure of source code.

Another aspect of distillation is that one should attempt to focus attention on one particular theme of the codebase at a time. This is based on the observation that further stages of CDMS often involve more work than distillation and thus a conservative approach should be taken to avoid oversaturation of work in later stages.

Examples of themes include:

- Models, Views or Controllers in the case of MVC GUI app
- Graphics API code, in the case of a rendering engine
- Robotic actuation or sensor handling

Once distillation is complete, the engineer may transition to either the modeling stage, synthesis stage, or coding stage. Transitioning to Modeling occurs when the current domain model must be updated to include additional domain-specific information or concepts. This transition is the common case. Transitioning to Synthesis occurs when no new domain-specific information has been discovered during distillation but new code has been identified for synthesis (See "Modeling - 1 - QT" in the GUI case study for an example of this transition type). Finally, a transition back to coding occurs when it is realized that modification to the existing code would make it more amenable to distillation.

Modeling

During case study evaluation, we found that attempting to create a partial domain model from sets of distillation observations proved to be a valuable and insightful process, thus we decided to dedicate an entire methodology stage to this aspect of development. Critically we found that building the domain model iteratively or "one small piece at a time" instead of attempting to define all aspects of the domain upfront paid incredible dividends with regard to adaptability with respect to unforeseen problems encountered during development. To encourage iterative model development we decided to provide an additional path in our methodology from synthesis back to modeling. We will discuss this new path in more detail in the synthesis section.

More formally, during the modeling stage, the engineer incorporates observations made either in the distillation phase (normal case) or the synthesis stage, into the domain model. Additionally, the engineer determines what code fragments should be synthesized entirely by CoGen and what code fragments should be "inlined/imported" explicitly during the synthesis phase.

As mentioned previously, a domain model is a set of classes designed to codify the concepts most relevant to the problem domain. During case study evaluation we identified a number of useful heuristics designed to aid in determining what information and structure should be codified by the domain model. Heuristics include:

- The domain model should describe WHAT the underlying system is (and does) not HOW it does it.
- The domain model should contain information one would expect to see in a configuration file namely information that parameterizes the functionality of the system but does not specify the actions associated with said functionality.

In addition to the heuristics above, we found ourselves consistently applying the following decision process during modeling.

- If a given observation is not directly relevant to the problem domain then continue to the next observation and consider the current observation during the upcoming synthesis phase.
- If a given observation is related to an implementation-specific class (for example, SQL connection) then continue to next observation. "GUI Case Study: Modeling - 1 - JavaFX"
- If alternatively the class is related to a domain-specific concern (for example represents an Inventory Item) then add a new concept to the domain model which mirrors the original class but excludes any implementation-specific data or behavior which may have been attached to the class, leaving only domain-specific information.
- If a given observation is related to a domain-specific data value then determine what existing concept in the domain model, the data value is most related to. If non exist, create a new concept to encapsulate the data value otherwise add data value to the most related class.
- If a given observation is related to the lifecycle of some domain-specific object then add a new concept to the domain which encapsulates the data required to influence aspects of the lifecycle. We recommend using the policy pattern to encapsulate such lifecycle-specific concerns. "Rendering Case Study: Modeling - 3"

Once modeling is complete, the engineer may transition to the synthesis stage to leverage gained insights and domain model information.

Synthesis

Returning to our case studies, we found that after acquiring a number of observations and an initial domain model, the next natural step was to begin developing synthesis specifications. We identify the process of constructing synthesis specifications as Synthesis.

After many hours experimenting with synthesis we found that an effective strategy for building synthesis specifications was to target generating existing handwritten code exactly as it occurred in the existing codebase, with certain values from the domain model substituted during generation. Once the original code had been generated in a 1-1 manner, only then would we begin integrating more information from the domain model into the generation process thus allowing for further extensibility and variation in code structure as a function of the domain model.

In summary: During the synthesis stage, the engineer uses the domain model and code structure insights derived from the previous distillation and modeling phases to build out a synthesis specification.

From the synthesis stage, we identified two natural transitions namely one to the coding stage and another back to the modeling stage. The transition to the coding stage allows one to continue modifying code either by adding new functionality as required by incoming functional requirements OR to leverage newly generated code directly. The second transition back to modeling occurs when the domain model is missing critical information required to synthesize a particular code fragment. For an example of this transition type see "Synthesis - 3" in the robotics case study.

2.2.3 Inspiration

Our methodology takes key inspiration from two fundamental ideas namely Stepwise Refinement and Refactoring. Stepwise refinement is a software engineering methodology wherein one starts with high-level set of specifications and iteratively and recursively refines/expands each specification until all that remains are detailed instructions that may be explicitly executed by a computer. At each refinement step, a number of design decisions are made. These design decisions may be influenced by any number of factors, for example, constraints of the underlying hardware or insights from the problem domain or newly discovered characteristics of the problem [3] The key insights we extract from stepwise refinement are that of high-frequency iteration and the steady increase in system detail at each iteration. Another key influence in the design of CDMS is a well known software engineering method called Refactoring. In refactoring small "behavior preserving" transformations are made to a software system iteratively such that the system may converge on a higher quality implementation while remaining functional at all times. [14]

CDMS turns stepwise refinement upside down and operates in a purely bottom-up fashion. One starts with an existing codebase and iteratively extracts from the codebase more general details until eventually arriving at a high-level specification encapsulating the necessary details of the original system but devoid of any particular details surrounding code structure. Taking inspiration from refactoring, each iteration of distillation is as small as possible thus facilitating smooth and methodical progression towards a robust representation.

2.3 Design Patterns

During the application of our approach, we encountered a number of interesting design problems. In this section, we describe each problem and discuss our devised solution. Note that we express each solution in the form of a design pattern as we believe each solution may be successfully applied in many different contexts.

2.3.1 Policy Pattern

In some cases, the structure of extremely specific code spread across an entire codebase may need to change as a function of some domain-level concern. This issue of cross-cutting concerns was encountered in the Robotics case study where initialization and teardown methods required different code depending on the communication policy of the node.

To solve this problem we suggest creating a new domain class to encapsulate the particular concern (communication policy in the case of ROS) then reference this policy in the synthesis specification during generation of multiple related code regions.

2.3.2 Type Projection Pattern

At times it is desirable to make accessible to generated code the contents of an entire domain model object instance. Such a scenario arose in the JavaFX case study wherein TextElement information encoded in the domain was required at runtime. A similar scenario also occurred when synthesizing ROS messages in the robotics case study.

To solve this problem we recommend the synthesis of a class in the target implementation which has a 1-1 mapping with the member variables of a specific domain class. The contents of the domain class may then be accessed at runtime by generated code (assuming correct object instantiation).

Chapter 3

Related Work

Using program synthesis to generate boilerplate code is not a new idea. Many researchers and practitioners alike have attempted to achieve exactly this goal. In the following discussion, we will examine existing approaches to code generation, both with respect to boilerplate generation and in general. We will then outline the limitations of current approaches and discuss why we believe our proposed solution overcomes the stated limitations.

As seen in the introduction, we will compare synthesis approaches with respect to the degree of code derived through deduction by the synthesis engine as compared to code specified completely or partially by the user. As before Figure 3.1 will guide our discussion.

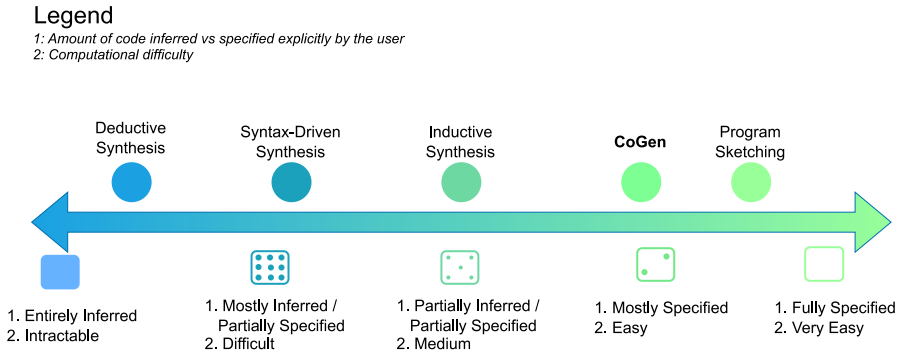

Figure 3.1: Span of synthesis approaches

3.1 Synthesis Paradigms

3.1.1 Deductive Synthesis

In our examination of synthesis paradigms we start with the most computationally intensive class of approach, deductive synthesis. Deductive synthesis is a method of program synthesis wherein source code is generated from a complete formal model of one or more aspects of a given domain [9]. This approach is beneficial in that it allows one to make extremely strong statements about the correctness of synthesized output, however this method struggles in pragmatic use cases due to the difficulty of model creation i.e issues of model rigidity and intractability.

3.1.2 Type-Driven Program Synthesis

Type-Driven Program Synthesis, developed by Nadia Polikarpova and her lab attempts to improve the scalability of deductive synthesis to larger real-world domains [13]. Type-Driven Program Synthesis improves on deductive synthesis approaches by leveraging type theory to both prune the search space of possible programs as a function of type compatibility and also allow for more flexible specifications through the usage of so-called refinement types. More specifically, "questions" in specifications are defined as types and the set of possible answers to each question must be within a set of valid type interactions. Such a restriction of allowed interactions greatly decreases the size of the program search space especially as compared to more unconstrained logical specifications seen in purely deductive approaches. Finally, Type-Driven Synthesis has the benefit of providing a greater degree of standardization of synthesis specifications, due to the fact that any strongly typed language may theoretically be used as the specification method. A number of real-world solutions have come from Type-Driven synthesis including synthesis of code to facilitate REST-based API implementations [12]

3.1.3 Inductive Synthesis

The next form of synthesis demonstrates a strong departure from deductive synthesis both in terms of computational intensity and amount of code specified vs generated. Inductive synthesis is a synthesis process that takes as input a partial code specification and other supporting artifacts such as expected program input-output examples. The generator then uses these artifacts to fill in facets of the specification left unspecified thus producing a final program. Unlike deductive synthesis, the resultant program may vary in correctness however the computational difficulty of generating the program is much reduced as compared to purely deductive approaches. Given these design decisions, inductive synthesis falls in the middle of our axis of comparison. Many implementations of inductive synthesis have been attempted. One prominent example is the PROSE framework developed by Microsoft Research. PROSE provides support for both inductive and deductive modalities of synthesis however the inductive facet of PROSE has seen success in a number of limited domains such as flash fill in Microsoft Excel [11]. One desirable aspect of PROSE is its ability to produce useful results in multiple well-defined domains, unfortunately however, the technology does not appear to scale well to the synthesis of larger complex software systems.

3.1.4 Counterexample-guided Inductive Synthesis

Counterexample-guided Inductive Synthesis (CEGIS) proposed by Solar-Lezama of MIT, enables scaling of inductive synthesis to more complex problems through iterative validation and improvement of synthesized candidate programs. The user first provides the CEGIS framework with partial programs and specifications. Within these specifications exist both fully specified and partially specified subsections where partially specified sections are called generators. The goal of CEGIS is to find the appropriate values for each partially specified section according to provided constraints. Next synthesis occurs in an iterative manner where the final program is de-ambiguated through consideration of failure counter examples produced by a post synthesis validation phase [10]. A key benefit of CEGIS is that it enables more rapid convergence to useful output programs while also providing the user with a higher degree of requirements flexibility and expressivity as compared to other deduction centric approaches. In a similar manner to PROSE however, CEGIS, in its original implementation appears restricted to a few relatively small and extremely well defined domains.

3.1.5 Program Sketching

One important contribution of CEGIS is the inspiration of a new paradigm of program synthesis called program sketching. Program sketching is a synthesis process wherein the user provides

the synthesizer with a partially complete program with certain "questions left unanswered". The synthesizer is then responsible for finding the best "answer to the question" while respecting certain correctness/compatibility guarantees. Program synthesis provides the greatest degree of specification flexibility while being the least computationally intensive in general. These facts place program sketching on the far right side of our axis of comparison.

3.1.6 Natural Language Driven Synthesis

More modern program synthesis paradigms include using machine learning systems such as LLMs (Large Language Models) to generate code from natural language descriptions [23]. There exist two major drawbacks to this approach. Firstly, there exists zero guarantee that the generation code produced by a machine learning model is correct or even valid syntactically. Additionally, due to the potential ambiguity of natural language-based descriptions, the code may be correct in the sense of solving a specific problem in a valid way, however, the actual program may not solve the same problem intended by the user. Another key issue with machine learning based approaches are ethics surrounding proper crediting of work. LLMs are often trained on large dataset consisting of code scraped from the internet. This implies that the generated output of an LLM is derived from the existing work of others and since current machine learning systems cannot assign credit to generated output, one could argue that such systems are plagiarizing existing work [35].

3.1.7 Diagram Driven Synthesis

Another common code synthesis approach is the generation of code from user specified diagrams [21] [28] [29]. Unfortunately, diagrammatic code generation may not always scale well to large projects due to the overhead of manipulating large diagrams on a 2D screen. Additionally, specific diagramming paradigms tend to be extremely domain-specific (specific to control systems in the case of Matlab) or restricted to a particular paradigm or way of thinking (UML in Rational Rose or Enterprise Architect).

3.2 Synthesis Systems

Now that we have discussed a number of code generation paradigms and approaches we will now focus our attention on a number of particular implementations of synthesis technology and especially those designed to facilitate boilerplate generation.

3.2.1 GUI Boilerplate Synthesis

One area where much work has been done to automate boilerplate maintenance and generation is that of GUI code generation. Systems such as the AndroidStudio Layout Editor [30] or JFormDesigner [31] provide the user with a WYSIWYG interactive editor wherein they may visually construct a GUI. Either system will then generate the code required to realize the user-specified user interface. Such systems can indeed accelerate development, however, they often produce code that is difficult to read and may even become desynchronized from the visual representation. Additionally, generated code can be difficult to debug due to its opaque nature.

3.2.2 Control Systems Synthesis

One common application of program synthesis is the generation of safety-critical code generally used to control mission-critical cyber-physical systems. One example of such a system is VeriPhy developed by Rose Bohrer et el. Veriphy is a complete end-to-end framework for synthesizing formally verifiable system controllers and monitors from one or more formal models formulated in differential dynamic logic [34]. This allows VeriPhy to model Hybrid Systems, systems involving

both continuous and discrete dynamics. Finally, VeriPhy overcomes some of the tractability issues of a purely deductive synthesis approach by validating the behavior of synthesized components at runtime, a technique called runtime validation [33].

3.2.3 Compiler Code Synthesis

Another key area that has seen boilerplate automation work is that of Compiler construction. When implementing a compiler for a particular programming language, many low-level aspects of language parsing remain similar across different language types. Yacc is a famous system that takes as input an abstract context-free grammar describing a target programming language. From this specification, Yacc then generates the required code (in the C programming language) required to implement the given grammar and thus foundational language parsing [32]. In general Yacc produces satisfactory results for simple languages. Unfortunately, the specification of more complex language can be verbose and difficult to understand. Additionally, code generated by Yacc may be difficult to integrate with existing compiler infrastructure.

3.2.4 API Code Synthesis

Google Protocol Buffers are a relatively popular example of synthesis in the communications protocol domain. The Protobuff synthesizer takes as input IDL or interface definition files and outputs source code consisting of data structures and corresponding serialization machinery. This generated code is responsible for acting as a bridge between the client runtime (which may be any number of programming languages) and the low-level protocol buffer transmission infrastructure. [16] Google protocol buffers provide a clear example of the benefits of code generation with respect to the reduction of non-essential complexity, in this case the complexity of transmitting structured data over an unstructured communications channel.

3.2.5 Multi-Domain Synthesis

There have been a few attempts at realizing code generation in a cross-domain manner. Two key technologies in this regard are the aforementioned Rational Rose and Enterprise Architect systems [28, 29] Unfortunately both of these systems restrict the user to a particular (opinionated) modeling paradigm. Additionally, both systems appear to emphasize top-down modeling alone and do not explicitly facilitate bottom-up modeling or more iterative methods of domain model formulation.

3.3 Comparison to CoGen

Let us now compare our chosen synthesis technology CoGen, with existing approaches. After an extensive review of the literature, we found a few key restrictions which we believe may limit the applicability of existing technologies to multi-domain synthesis of boilerplate. The first issue we identified was domain specificity. Many methods we examined were quite effective at synthesizing domain-specific code, for example communications protocol support code, GUI boilerplate, API wrappers, controls system code, however, these systems failed to provide sufficiently generic primitives to synthesize code across multiple domains (a requirement for solving the boilerplate problem in general). The second issue encountered was a lack of granular control over the structure of generated code. In order to realize an approach that works well with existing software engineering processes and supports synthesis in the context of highly complex systems we believe the engineer must be given complete control over the structure of generated code while still gleaning the generalization benefits of generating code as a function of an abstract model. The third and final issue is a lack of generality surrounding model specification. Many synthesis systems forced the user to specify their model using a domain-specific language or other restricted modeling formalism. For

our goals, we believe the user should be capable of codifying their model and specifications in a regular programming language thus facilitating maximum modeling flexibility.

Chapter 4

Case Study - GUI Domain

4.1 Overview

Here we demonstrate the application of CoGen and our synthesis methodology to the generation of GUI application code. Our goal is to synthesize the source code of a simple target application in two different application development frameworks (JavaFX and QT) and a 2d game engine (libGDX). The application we wish to synthesize consists of a single window containing a set of text labels arranged in a grid structure against a white background.

We generate the code for each target framework simultaneously as a function of a singular domain model. Our hope is to show how CoGen may reduce the workload associated with writing framework-specific boilerplate code for a single app which must be supported in multiple frameworks. Such a scenario arises commonly in software engineering when, for example, targeting different hardware platforms such as mobile and desktop. In this case study we start with the target example application implemented by hand in each target framework. We then incrementally synthesize aspects of the target codebases until all code is generated purely using CoGen.

Our goal for this case study is to act as a light introduction to synthesis using CoGen and CDMS. In short we are aiming for breadth over depth.

4.2 Domain Description

We characterize the gui domain in this case study as follows. Firstly we define the concept of a window. A window is a rectangular shape displayed on a screen. A window may contain 0-n elements where an element is some region of the screen with a specific visual appearance and functionality. Elements may be composed and thus are related to each other via a tree data structure. Additionally, elements may be positioned according to some layout. In this case study we consider a standard type of layout wherein elements are organized in a grid structure.

4.3 Target Description

We chose to generate code for JavaFX and QT and LibGDX as all three frameworks present a different architecture. QT and JavaFX are most similar in that they both explicitly present the concept of widgets, however both use different objects and methods to organize widgets. LibGDX on the other hand presents an entirely different paradigm for rendering of GUI elements. LibGDX is in fact a game engine and thus more granular code is required to render text on a screen. Our overall goal with the chosen targets was to demonstrate the variation capacity of CoGen within the same domain.

4.4 Application of Methodology

In accordance with the proposed synthesis methodology, we choose a bottom-up approach to the construction of both synthesis specifications and domain model. What follows is a step-by-step account of how we applied our methodology to the synthesis problem at hand.

4.4.1 Distillation - 1 - JavaFX

Upon initial examination of the JavaFX code example, the following observations were made.

- The main application class must inherit from a base JavaFX application class.
- The main application class must override three methods, one responsible for initialization, another for scene creation and another for cleanup.
- The user may create text labels which are a form of widget.
- Widgets must be attached to so-called scene
- Scenes can be attached to a so-called primary stage.
- The primary stage is used to set the window title
- The window title is a hardcoded literal string value.
- The primary stage is used to initial painting the GUI

4.4.2 Modeling - 1 - JavaFX

From the observations made above, we concluded that our domain model should include the concept of a Text Element and subsequently a more abstract Element from which Text element inherits. Next we introduced a class called Window containing a window title and a list of active widget elements. Finally, we created a base Domain class designed to contain a specific instantiation of our domain. For our first domain configuration, we instantiated a single text element and set the window title to "Basic JavaFX Application".

Overall, the guiding principle used in the introduction of these concepts was to codify both a high-level organization of system components and also information not specific to the JavaFX framework (in this case the primary stage or scene classes or the specifics of the JavaFX inheritance structure).

4.4.3 Synthesis - 1 - JavaFX

We begin synthesis by creating a new implementation provider for JavaFX. In the implementation provider, we first specify generators for the application class, and then generators for each key method present in the original codebase. Next we synthesize the actual code for each method one at a time. Note that for now we synthesize the code in the original target code baseline by line exactly as it occurs in the target.

The following code was the core focus of synthesis at this stage.

- Application class with inherited JavaFX Application base class.
- Init method with debug print and call to the base class init method
- stop method with debug print and call to base stop method

- start method which 1. instantiates a single text element (using domain model text element content as a constructor argument), 2. assigns the text element to a new scene object, 3. assigns the scene object to the primary stage and 4. sets the window title by passing the domain model scene title into the primary stage setTitle method as a string literal and finally 5. refreshes the page by calling update on the primary stage.

After synthesis of the basic example we now shift focus temporarily to our two additional targets.

4.4.4 Distillation - 2 - JavaFX

In our second distillation pass we now turned our attention to layout. Focusing on the layout component of the code we made the observation that JavaFX implements an explicit GridLayout class which facilitates the assignment of widgets at a specific x,y location in the grid.

4.4.5 Modeling - 2 - JavaFX

We decided that the concept of a layout as a whole was not specific to JavaFX and would be relevant to any GUI framework. For this reason, we decided to introduce two new classes to our domain model namely Layout and GridLayout, where GridLayout consists of a 2-d nxm array of widget elements. Finally in preparation for the next synthesis iteration we modified our domain instance to include an instance of our new GridLayout specifically a 3x3 grid of text elements.

4.4.6 Synthesis - 2 - JavaFX

In our next synthesis task we added a nested for-loop to the onstart method. Within the inner loop we instantiate a JavaFX text element and assign it to the corresponding JavaFX grid layout object. At this point we encountered an interesting problem. How, in the inner for loop, can we access the domain model-specific text element information at runtime such that we can instantiate a JavaFX text element with the proper content? We discussed various solutions to the problem, some of which we will demonstrate during synthesis of the QT and LibGDX targets. For JavaFX we settled on an approach we call type projection (see 2.3 for a more general description). The idea of type projection is to, for a given type in the domain model, synthesis a 1-1 equivalent of the domain model class in the target implementation and, at some point in the target code, populate sufficient instances of the target class type with the corresponding domain model information. This way, domain model information can be easily accessed during runtime via referencing the preinitialized class instances. In the case of JavaFX we synthesized a dedicated TextElement class containing a single string member variable. In the init function we created an array of TextElement classes and for each domain text element populated the corresponding Text Element with the contents of the domain model text element via assignment of a string literal. Finally with all text element information available at runtime, we modified the inner for loop such that Text element content was read from the array of Text element instances.

This concludes the JavaFX target.

4.4.7 Distillation - 1 - QT

We now turn our attention to the QT codebase. On our first consideration of the target codebase, we made the following observations.

- Overall codebase is very similar to JavaFX and leverages similar objects including, explicit GridLayout object. Additionally a widget composition architecture is used.
- method overrides are slightly different compared to JavaFX. No init or onstart just onclose method. Constructor used for initialization.

4.4.8 Modeling - 1 - QT

Given similarities to JavaFX, no additional modeling tasks were required to facilitate the QT target. This is a good sign as it implies our domain model is general enough to support multiple backend frameworks simultaneously (our original goal).

4.4.9 Synthesis - 1 - QT

The first difference between JavaFX and QT synthesis we implemented was to delegate initialization code to the main application class constructor. The second change involved handling of grid layout generation. Unlike the type projection approach taken in JavaFX, we decided to synthesize a string manipulation expression to produce the desired label context. The reason for doing this was to demonstrate how in some cases, domain information may be derived at runtime.

This concludes the QT target.

4.4.10 Distillation - 1- libGDX

The LibGDX target was fundamentally different from previous targets due to its game engine architecture. This major deviation from previous architectures called for a detailed distillation pass. Our distillation observations were as follows.

- LibGDX requires two primary classes, one to handle platform-specific runtime initialization and another class to implement the specific "game" / application logic.
- The platform initialization class must initialize a specific backend (in this case LWJGL) to facilitate access to lower-level window management and graphics rendering functionality.
- The platform initialization class sets various window parameters such as title, framerate and vsync mode.
- The application class class must inherit from a LibGDX application class.
- The application class overrides three lifecycle methods namely create, render and dispose
- Three key objects are used to facilitate rendering of text. The objects are a sprite batch object, font object and camera object.
- No explicit grid layout object or text element widgets are implemented by LibGDX. Instead, text is rendered to the screen using a specific font. Fonts are drawn at specific x,y coordinates using the sprite batch.

4.4.11 Modeling - 1 - libGDX

Even though the LibGDX implementation is quite different then JavaFX and QT, this did not necessitate mutation of the domain model.

4.4.12 Distillation - 1 - LibGDX

In a similar form to JavaFX and QT we first synthesized the application game class with its required parent class and lifecycle methods. We then synthesized the platform-specific class using the domain model window title. Returning to the game class, we then synthesized member variables for the font, camera and sprite batch classes. Next in the create method we synthesized code to instantiate the font, camera and sprite batch classes.

Next we turned our attention to the render method. First and foremost we synthesized code to clear the frame and configure the camera. Now it was time to determine how to render our text

elements on a grid layout. We decided to take a completely different approach then done with JavaFX and QT. Instead of using loop structures we decided to remove all iteration i.e flatten the code and generate the required code for each text element. We decided to generate flattened code as it demonstrates both the variability possible between different implementation providers with respect to code structure, but also how one can arbitrarily decide when to completely flatten code structure. This is a powerful capability as flattened code can often offer benefits both in terms of performance and readability. Naturally, the converse is also true namely that too much code structure denormalization can lead to quite unreadable code. We suggest there exists a balance between the two extremes that yields superior results.

We generated the flattened code as follows. For each text element in the domain we synthesized code to 1. calculate the correct x,y position of the font then draw the font with a particular literal string provided by the domain text element.

This concludes the LibGDX target.

4.5 Results

4.5.1 Artifacts

Resulting artifacts of this case study may be found in the following locations.

- Domain Model Location: Link
- Synthesis Specification Location:Link
- Target Codebase Locations:
  - JavaFx: Link
  - QT: Link
  - LibGDX: Link

4.5.2 Domain Model

Through application of our methodology, we converged on the following domain model. Although limited to Text elements and Grid layout, the model could be expanded to additional element types, layouts and perhaps even data binding related functionality.

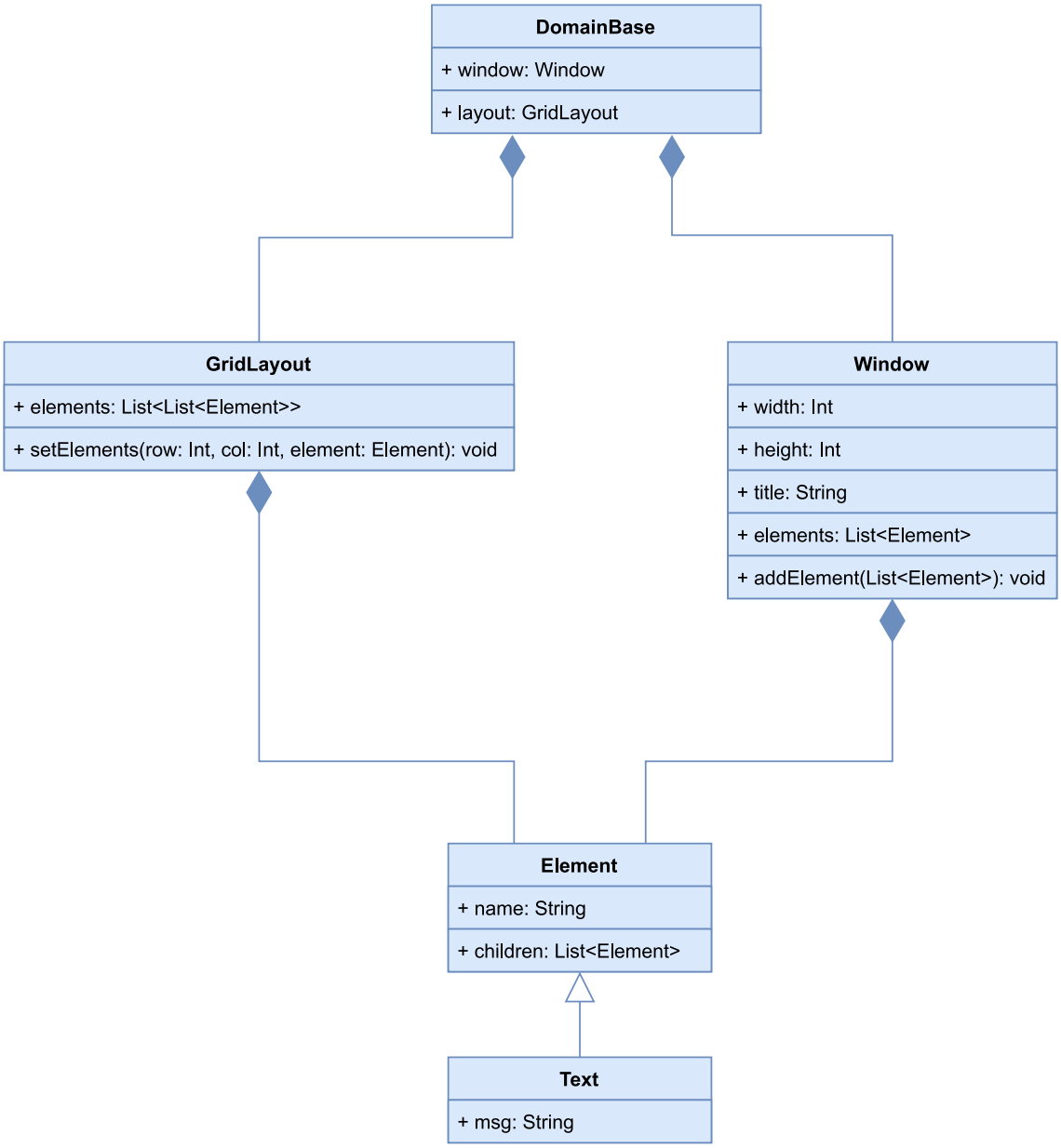

Figure 4.1: UML diagram of GUI domain model

In addition to the given domain model, we converged on the following per target synthesis specifications.

4.5.3 JavaFX

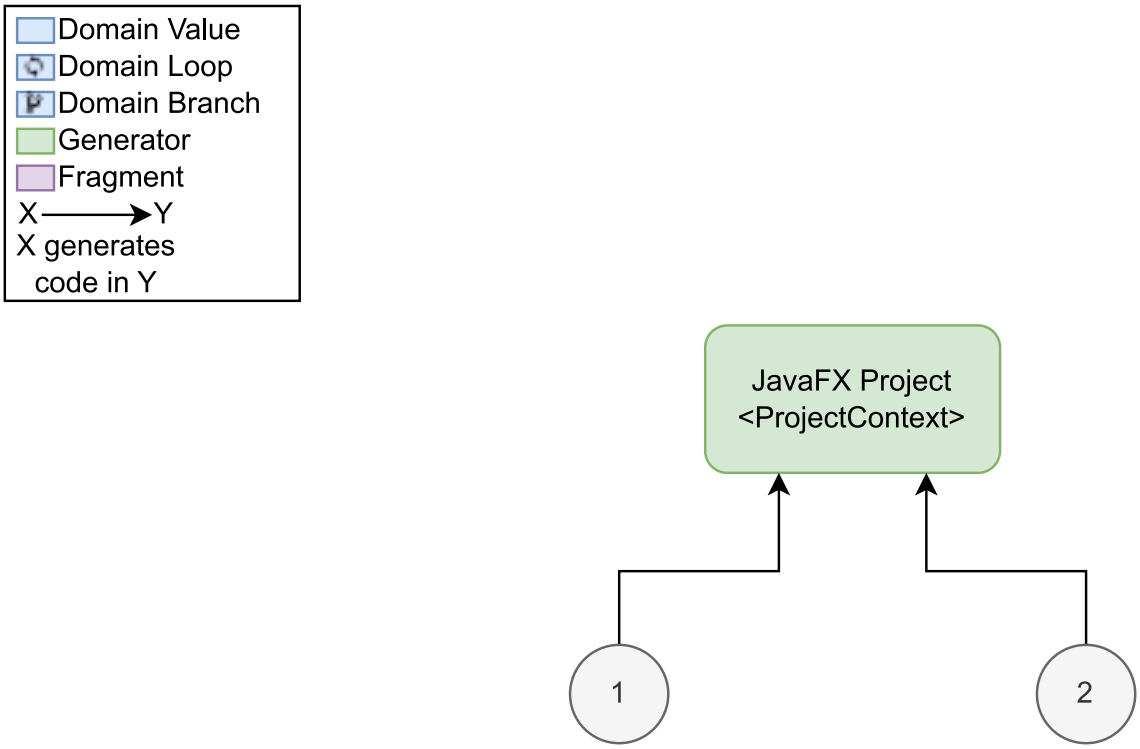

Figure 4.2: Synthesis tree for JavaFX target - 1

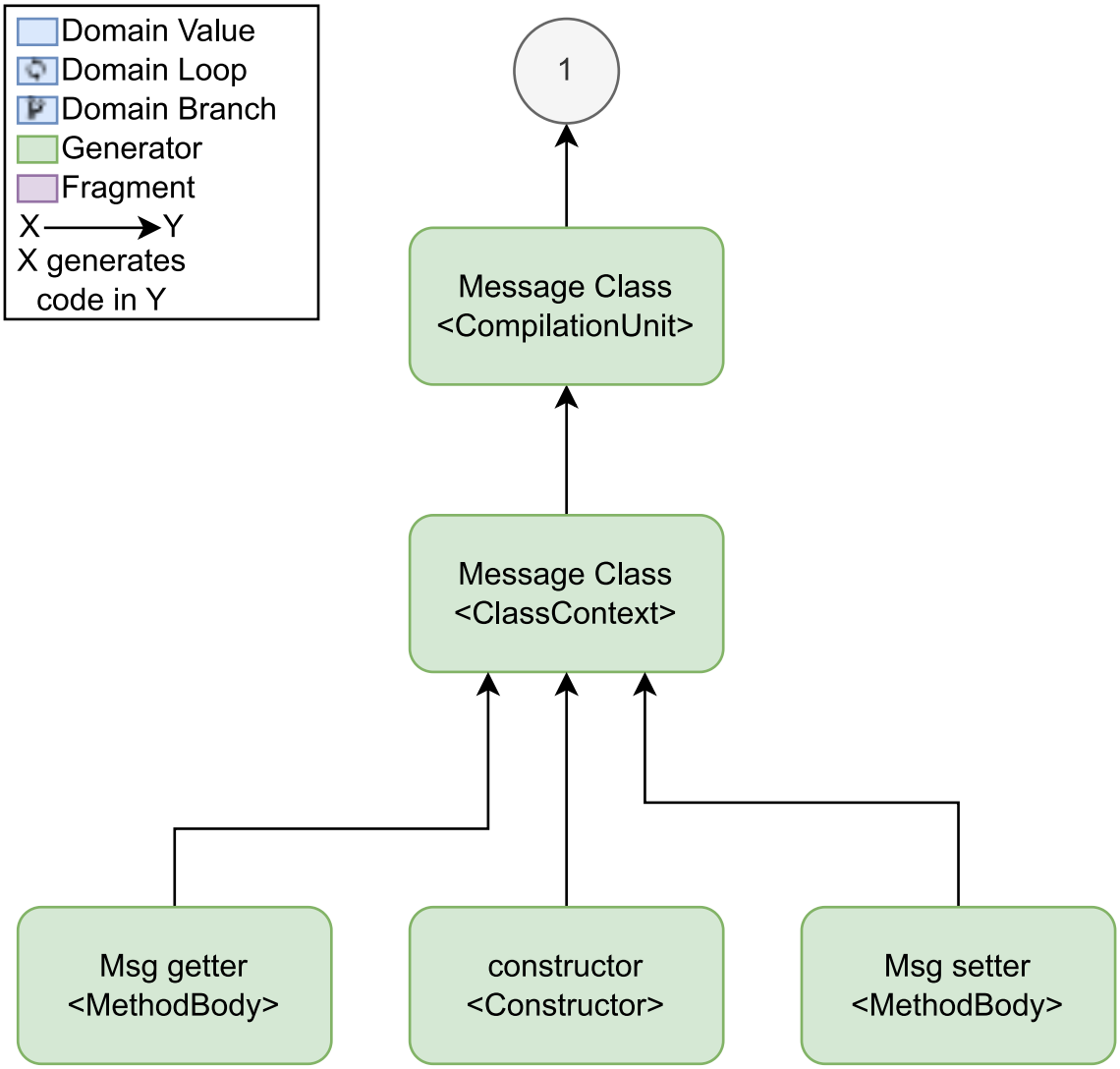

Figure 4.3: Synthesis tree for JavaFX target - 2

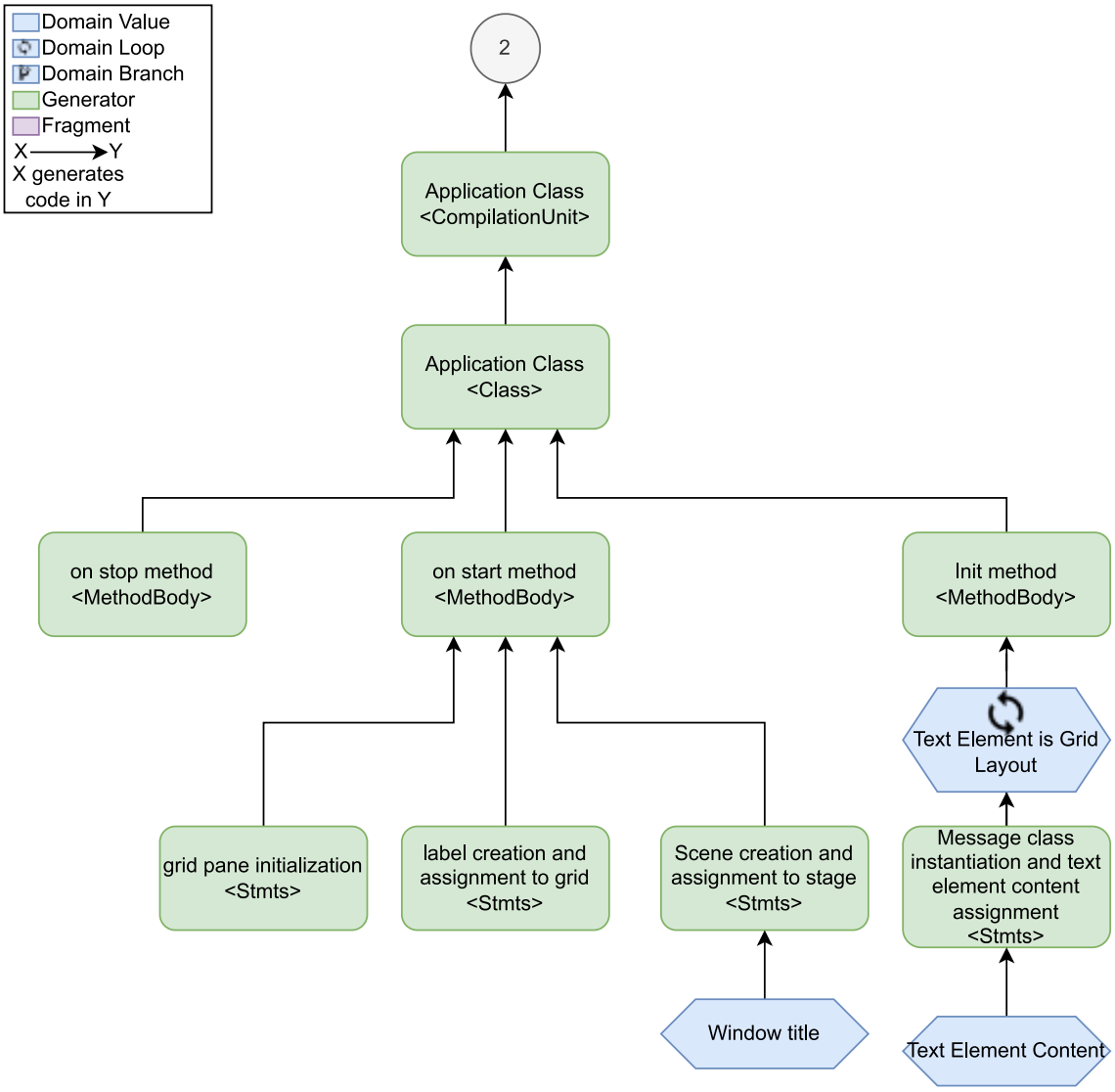

Figure 4.4: Synthesis tree for JavaFX target - 3

4.5.4 QT

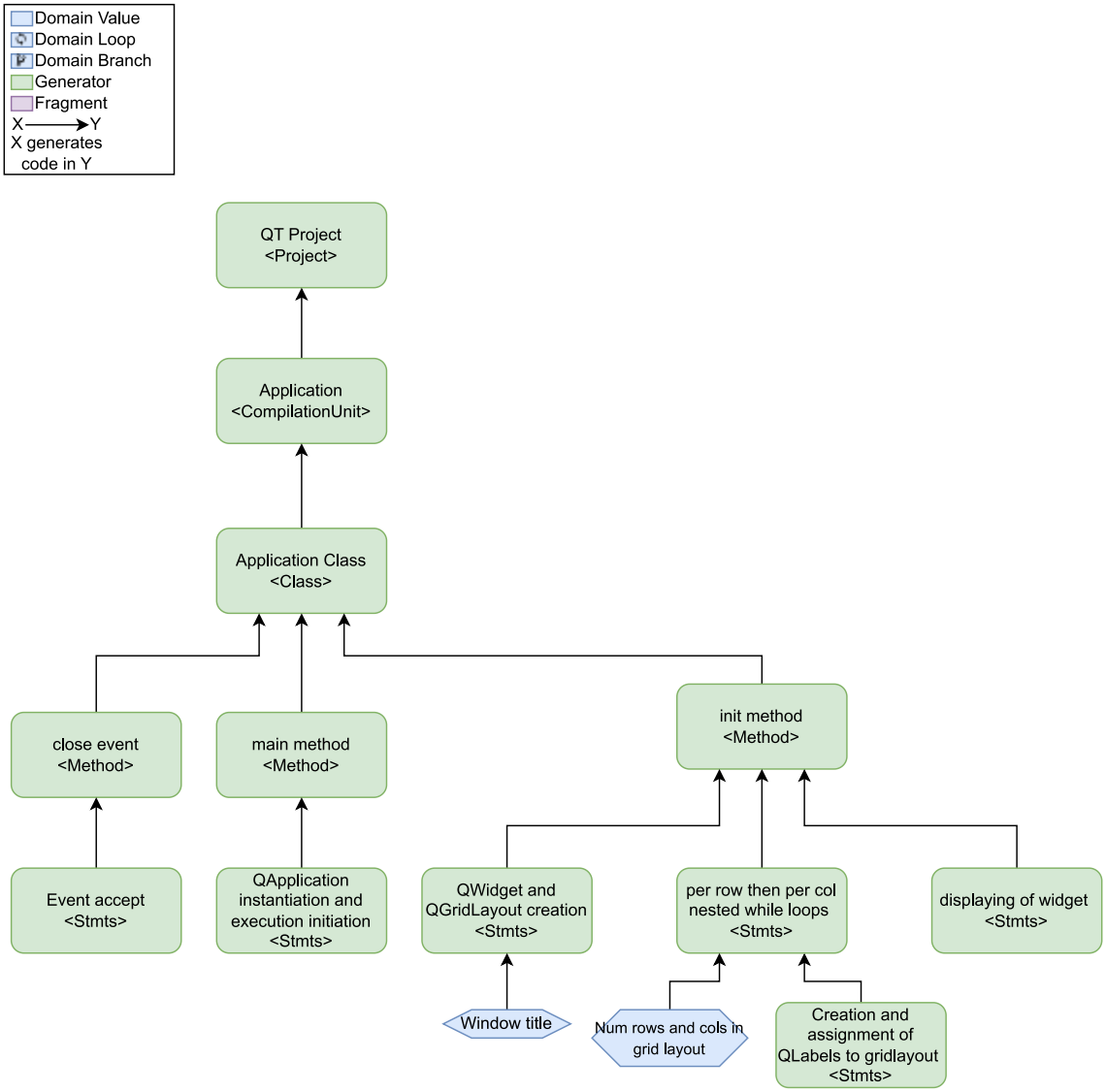

Figure 4.5: Synthesis tree for QT target

4.5.5 LibGDX

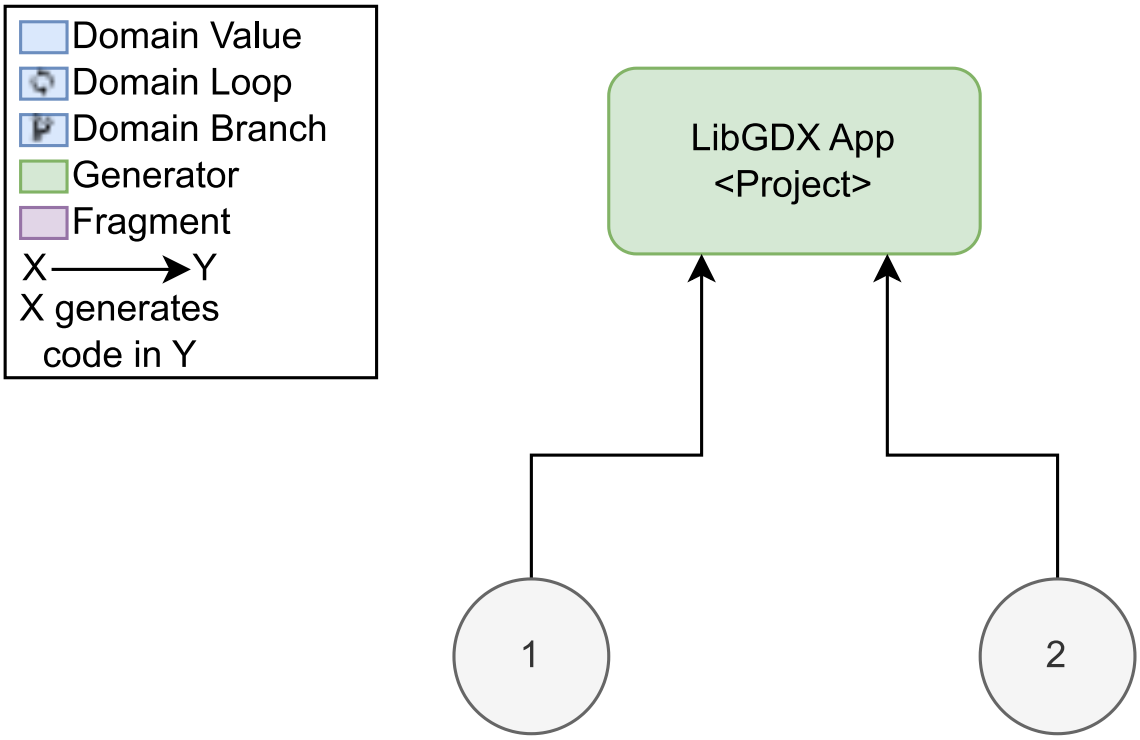

Figure 4.6: Synthesis tree for LibGDX - 1

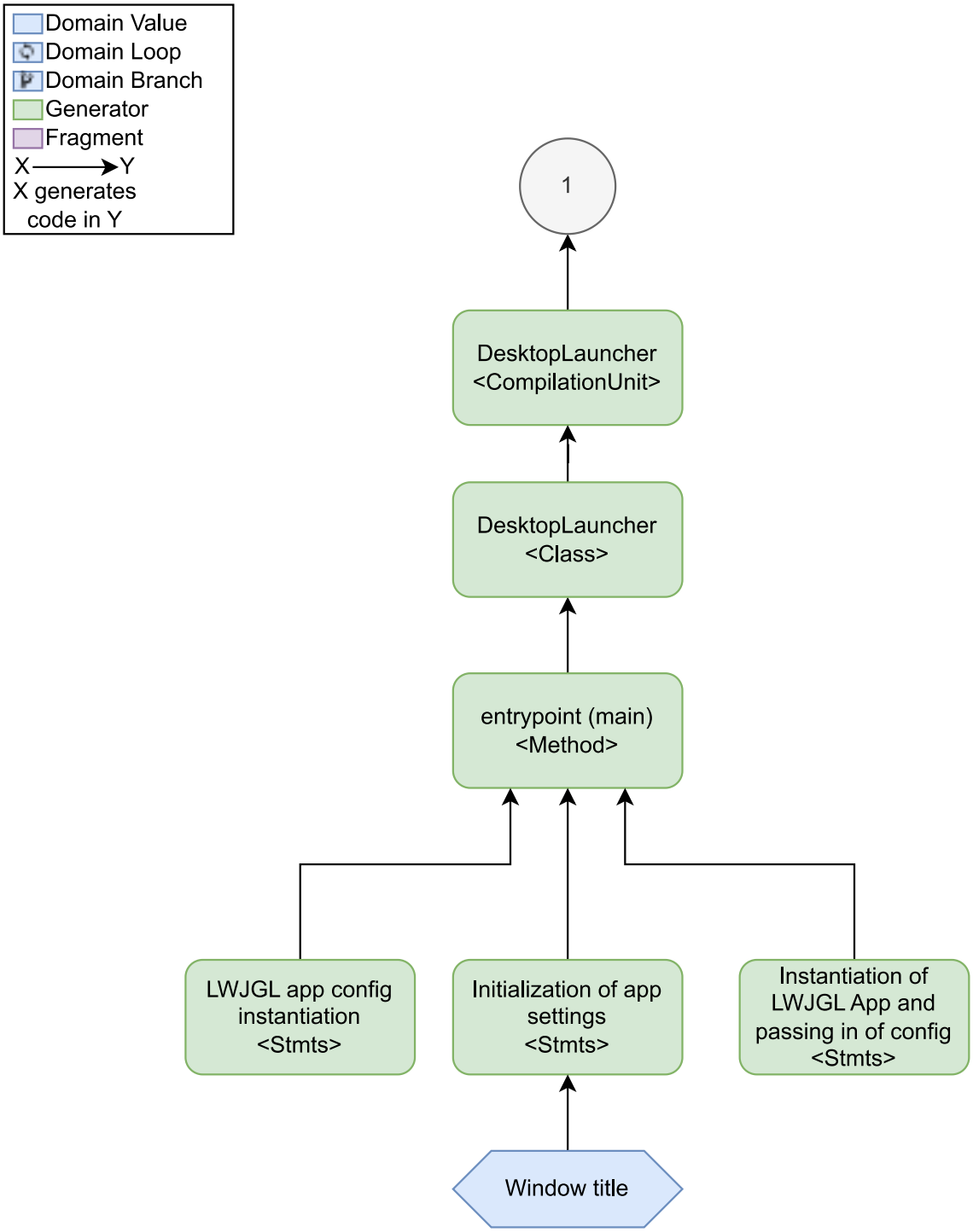

Figure 4.7: Synthesis tree for LibGDX - 2

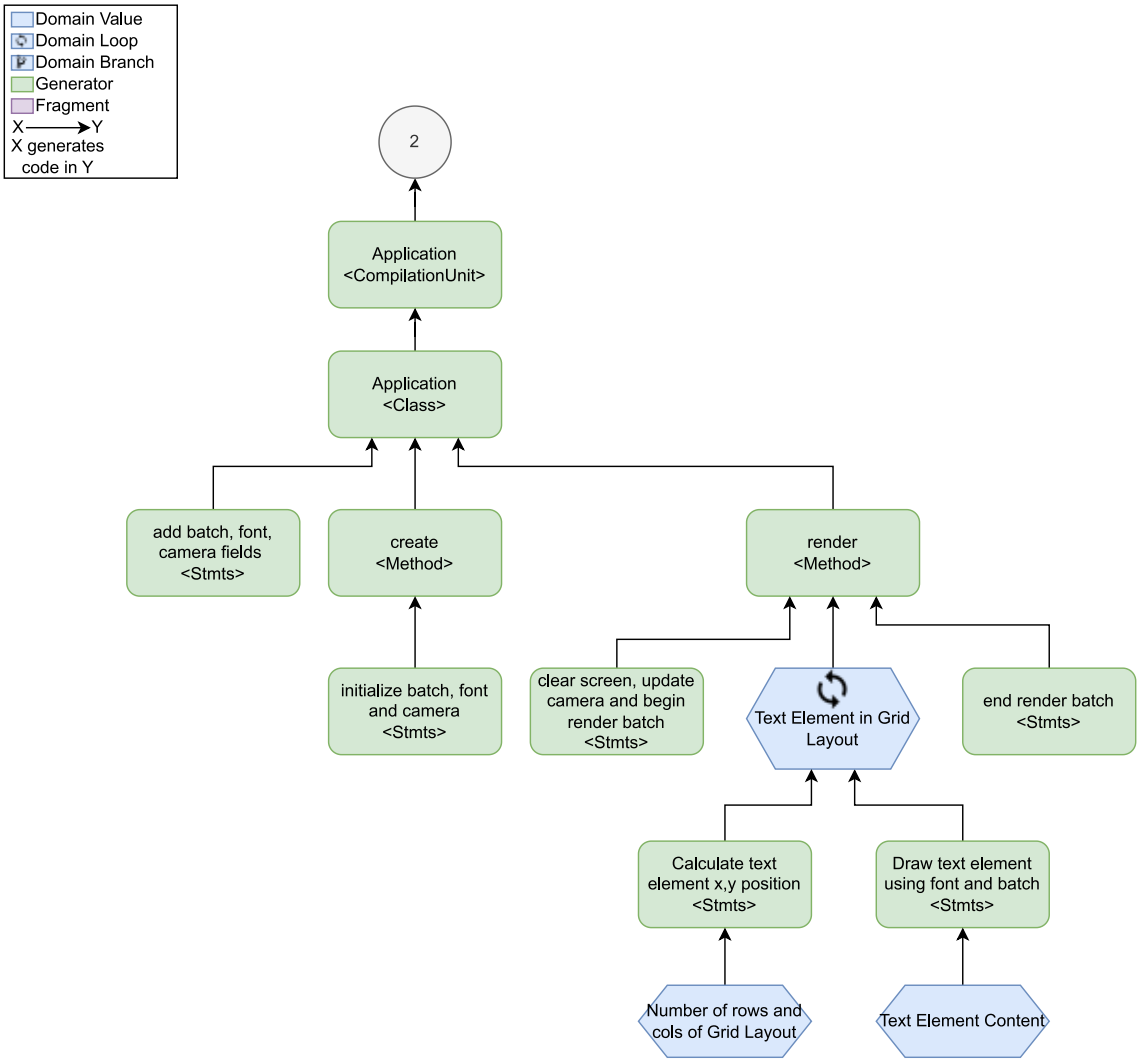

Figure 4.8: Synthesis tree for LibGDX - 3

4.6 Evaluation and Reflection

At the beginning of this case study we set out to synthesize the source code of a simple GUI application in three different GUI application frameworks simultaneously using a single supporting domain model. In each synthesized codebase we demonstrated the synthesis of a different implementation architecture, JavaFX using our novel type projection pattern, QT using runtime derivation of text element values, and LibGDX using flattened code structure.

We were able to synthesize 100% of application source code in all three frameworks and we feel the resulting domain model and synthesis specifications could easily be extended to further facets of the GUI domain and by extension target framework functionality. Overall we are quite happy with the results of this case study and feel it is an encouraging first look into the effectiveness of CoGen and CDMS.

Chapter 5

Case Study - Robotics Domain

5.1 Overview

In this case study we synthesize boilerplate code required to use the ROS robotics development framework. In the GUI domain we focused on breadth, synthesizing code to support a number of different targets in the same domain. The goal of this case study is to evaluate the depth of synthesis that is possible using CoGen and CDMS. We will show how CoGen can be used to synthesize code realizing nearly 100% of the feature set presented by the ROS framework. percent of boilerplate code usually found in a typical ROS project. Additionally, we demonstrate how the engineer may inject code fragments of their own design into a synthesized codebase thus allowing easy integration of existing non-synthesized business logic.

5.2 Target Description

Our synthesis target for this case study is a Java port of the ROS framework aptly named ROSJava. ROSJava was originally designed to facilitate usage of ROS under Android, however, the port also supports desktop operating systems. We based our initial codebase on the cited example. [18]

5.3 Domain Description

We chose in this case study to model a specific subset of core ROS functionality including ROS node lifecycle, basic node initialization and teardown, ROS message generation and handling and boilerplate related to the ROS node communication fabric (eg: Client/Server, Publisher/Subscriber).

5.4 Application of Methodology

5.4.1 Distillation - 1

For the first pass of distillation, we made the following observations:

- In the example code there exist a number of ROS node instances. Each instance takes on the role of either client, server, publisher or subscriber.
- Each ROS node has a string-based identifier.
- Each ROS node executes some form of business logic in a loop callback. Logic includes printing debug messages and in the case of one server node, adding two numbers.

5.4.2 Modeling - 1

From the distillation above we determined that at its core, our domain model should be centered around the ROS node. This idea aligns with the distributed systems architecture of ROS namely that a system should be broken up into nodes that ostensibly operate autonomously and interact via message passing.

To reflect the key observations made during distillation we added to our domain model a ROS class containing both an identifier string and the name of an optional code fragment to be executed during the periodic node callback.

5.4.3 Synthesis - 1

Our first synthesis specification generated a ROS java class for each ROS node instantiated in the domain model instance. To this class, the synthesis spec added a single string member variable to hold the node id, and also implemented the onstart method. Finally, the onstart method was populated with code to register a periodic loop callback (if a code fragment was specified in the domain model). The loop callback was realized as a functor and thus the synthesis specification was extended to synthesize an additional functor housing the user-provided code fragment.

5.4.4 Distillation - 2

In our next distillation pass, we focused on the concept of node roles and overall ros node communication. We framed our distillation around the fact that, as observed previously, each ROS node can contain additional code to facilitate communication with other nodes as either a Client, Server, Publisher or Subscriber. (Note: under ROS a single node can actually take on multiple roles simultaneously but for the sake of simplicity we will only consider one role per node instance)

Below were our observations in this context.

- Subscriber and Publisher nodes contain the name of a specific topic they publish messages to or consume messages from.
- Server and client nodes keep track of the name of the service they are providing or consuming.
- Client nodes implement a handler for responses returned by an invoked server
- Server nodes implement a handler for requests sent by clients
- Client and Subscriber nodes each have a supporting Listener class with methods to facilitate responding to either server responses in the case of client or publication of new messages in the case of subscriber.

5.4.5 Modeling - 2

From the observations above we saw that the role of a node significantly affected its structure and behavior while still leaving functional overlap between node types. Additionally, each role required slightly different role specific information (topic name vs service name). These conclusions led us to model node roles explicitly in the domain model and then pass a role into a domain model node instance during creation. the overall result of this decision was that we could add different variations of code to a ROS node during synthesis without interfering with more generic node components. Additionally as outlined in Distillation - 2, some roles require the creation of supporting listener classes.

Concretely we added to our domain model a role object for each role type. In each of these role objects we included parameters for role-specific information. To all role types (except server) we also added parameters for the specification of code fragments the user would like to run at

important points in the communication lifecycle. Such points include, upon receiving a response from a server, upon sending a request to a server, upon consuming a message published on a topic.

5.4.6 Synthesis - 2

Synthesis of the newly introduced communication functionality was time-consuming but straightforward. Firstly, for each node with role of Subscriber or Client we synthesized a supporting Listener class and appropriate handler methods. We then extended the onstart method of each ROS node class to register the listener class with the ROS runtime. We also extended onstart to create a ROS Client/Server/Publisher/Subscriber endpoint for our node and we stored the resulting handle as a field in our ROS node class. Finally, we loaded each domain model specified code fragment into the project as a class file. We then added calls to an expected static run method in each fragment to appropriate locations throughout the ROS code.

5.4.7 Distillation - 3

In our final distillation pass, we focused our attention on ROS Messages. Up until that point the codebase had relied upon predefined ROSJava messages. Upon examining the definitions of these predefined messages in the ROSJava codebase we made the following observations.

- Each message may contain any number of primitive member variables.
- Each message contains a string indicating the location of the package in the project package structure. (This is used to dynamically load the message at runtime)
- Each message contains a specially formatted string encoding the datatypes contained within the message.
- For each field in the message has a corresponding getter and setter.

5.4.8 Modeling - 3

In order to realize dynamic synthesis of messages we decided that from a domain model perspective, the user should be capable of specifying their messages as normal scala classes consisting of just the fields they want and requiring no additional specification of metadata.

5.4.9 Synthesis - 3

To synthesis ROS messages, we looped over every ros message type and then iterated over ever field in the message type (using reflection) and generated the corresponding java class including java equivalents of all fields, setters, getters and finally both type and location metadata strings. Additionally, in the case of Server messages, we also generated an additional message class required by ROS. This message class simply contains a concatenation of the Request and Response types used by the client and server respectively.

Now it was at this point that we encountered an interesting issue. Many ROS API calls are parameterized by the type of used during communication to or from an underlying node. In order to populate these types we decided the best course of action was to extend the domain model to support association of message types with roles.

5.4.10 Modeling - 4

Returning to the modeling phase we amended our domain model by adding to each role type, one or more parameters indicating the type of message communicated by the underlying ros node instance.

5.4.11 Synthesis - 4

Using the newly available domain model information we were now able to parameterize ROS API calls (such as message, request, response instantiation calls) with the correct message type.

5.5 Results

5.5.1 Artifacts

Resulting artifacts of this case study may be found in the following locations.

- Domain Model Location: Link
- Synthesis Specification Location: Link
- Target Codebase: Link

5.5.2 Domain Model

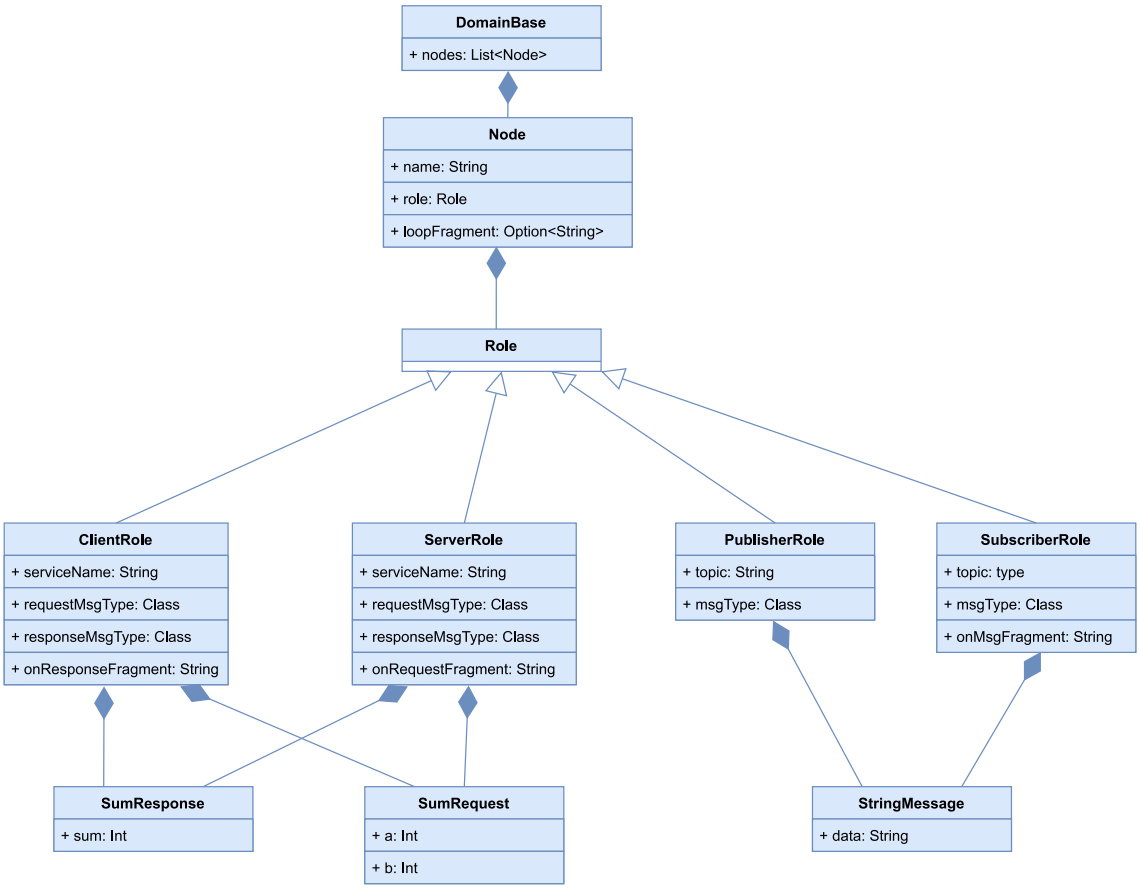

Figure 5.1: UML diagram of robotics domain model

5.5.3 ROS Node

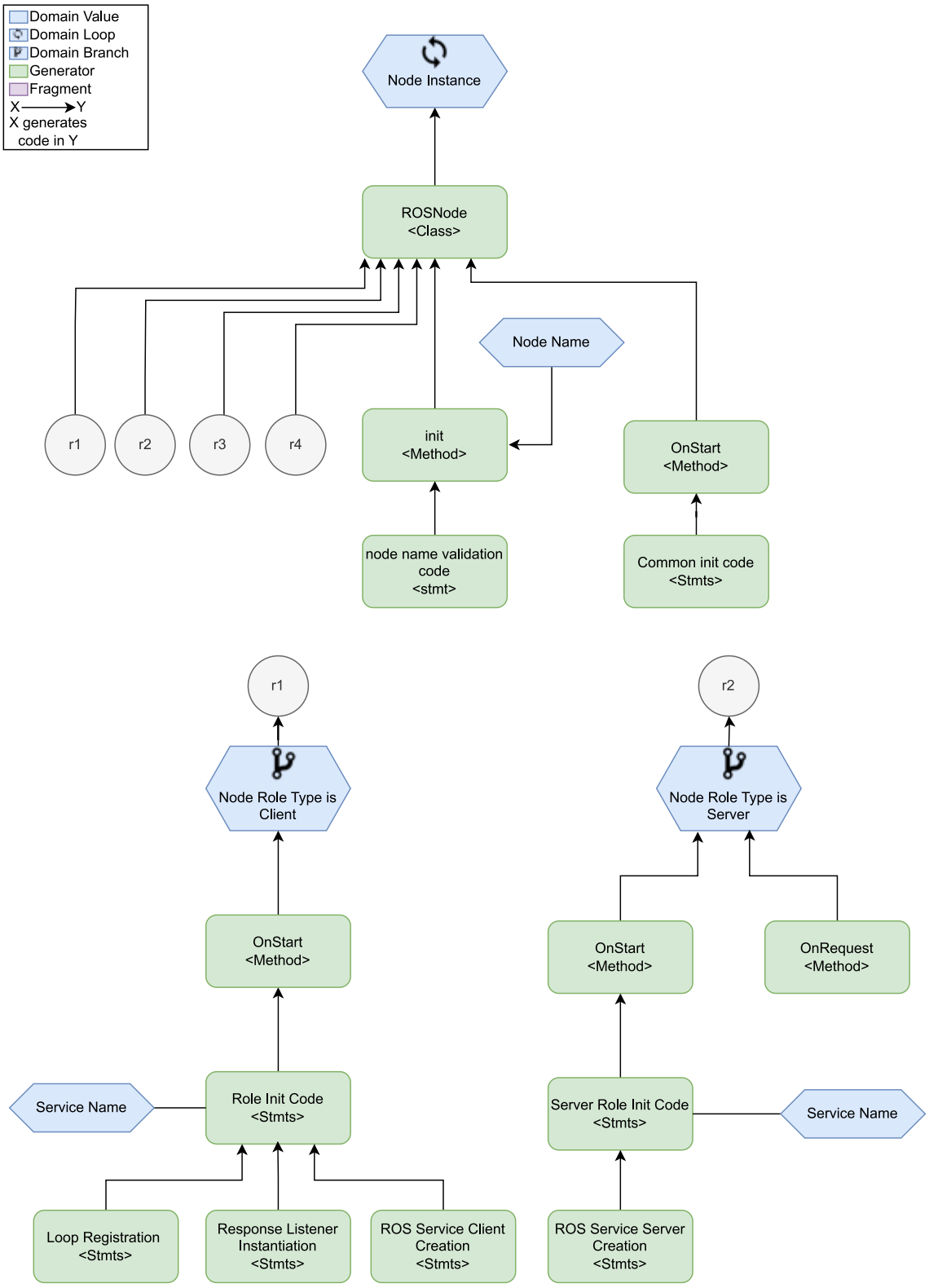

Figure 5.2: ROS Node synthesis tree - 1

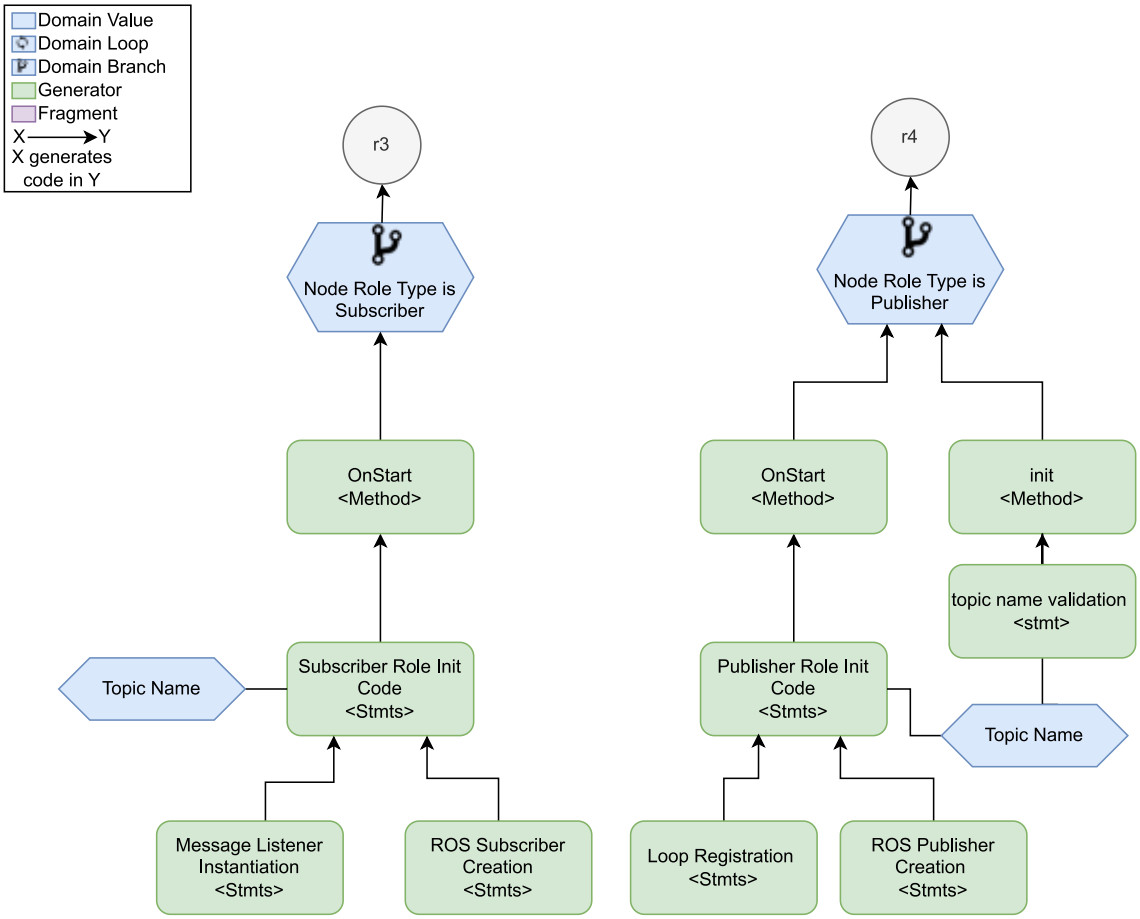

Figure 5.3: ROS Node synthesis tree - 2

5.5.4 ROS Node Listeners

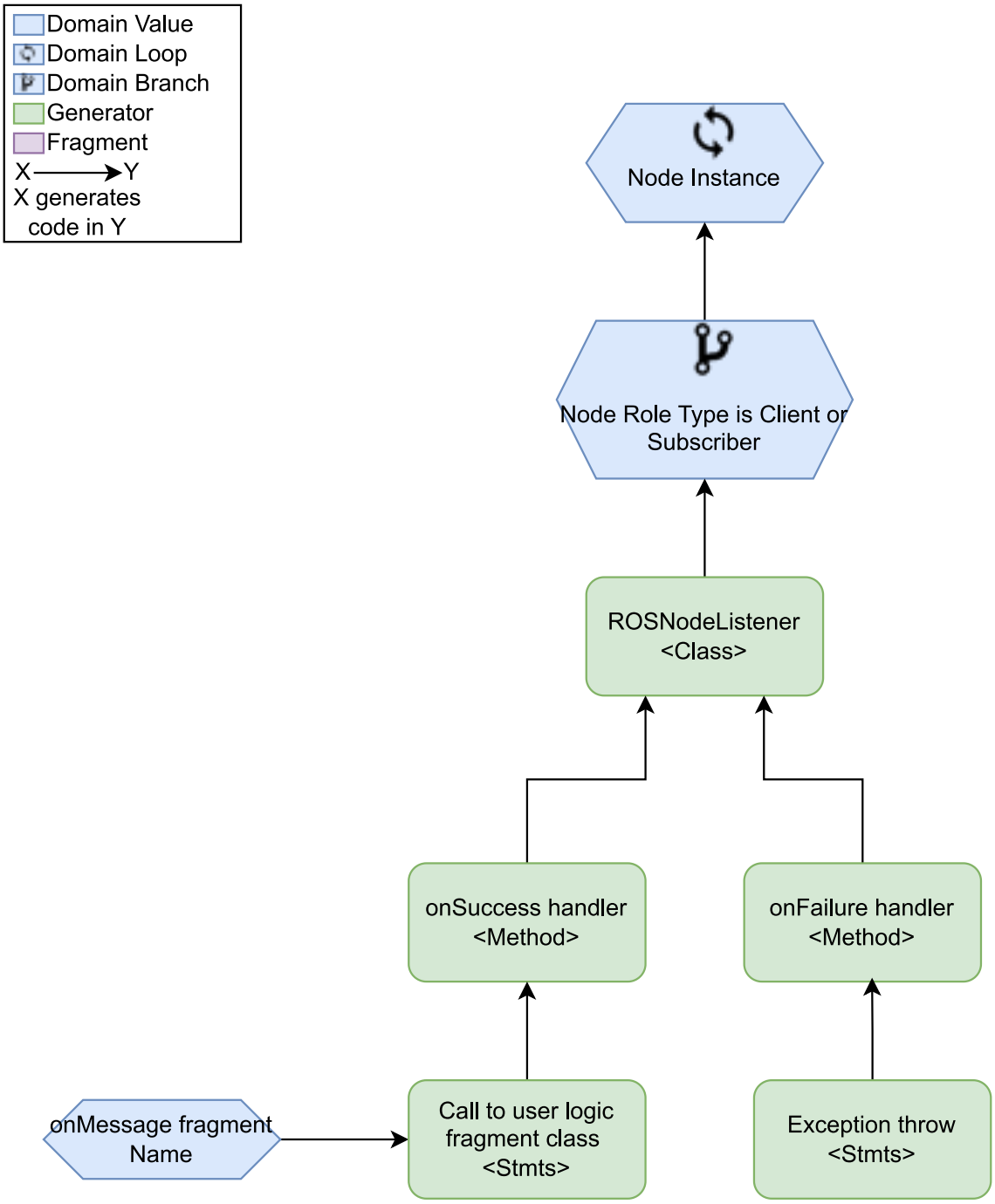

Figure 5.4: ROS Node Listener synthesis tree

5.5.5 User logic fragments

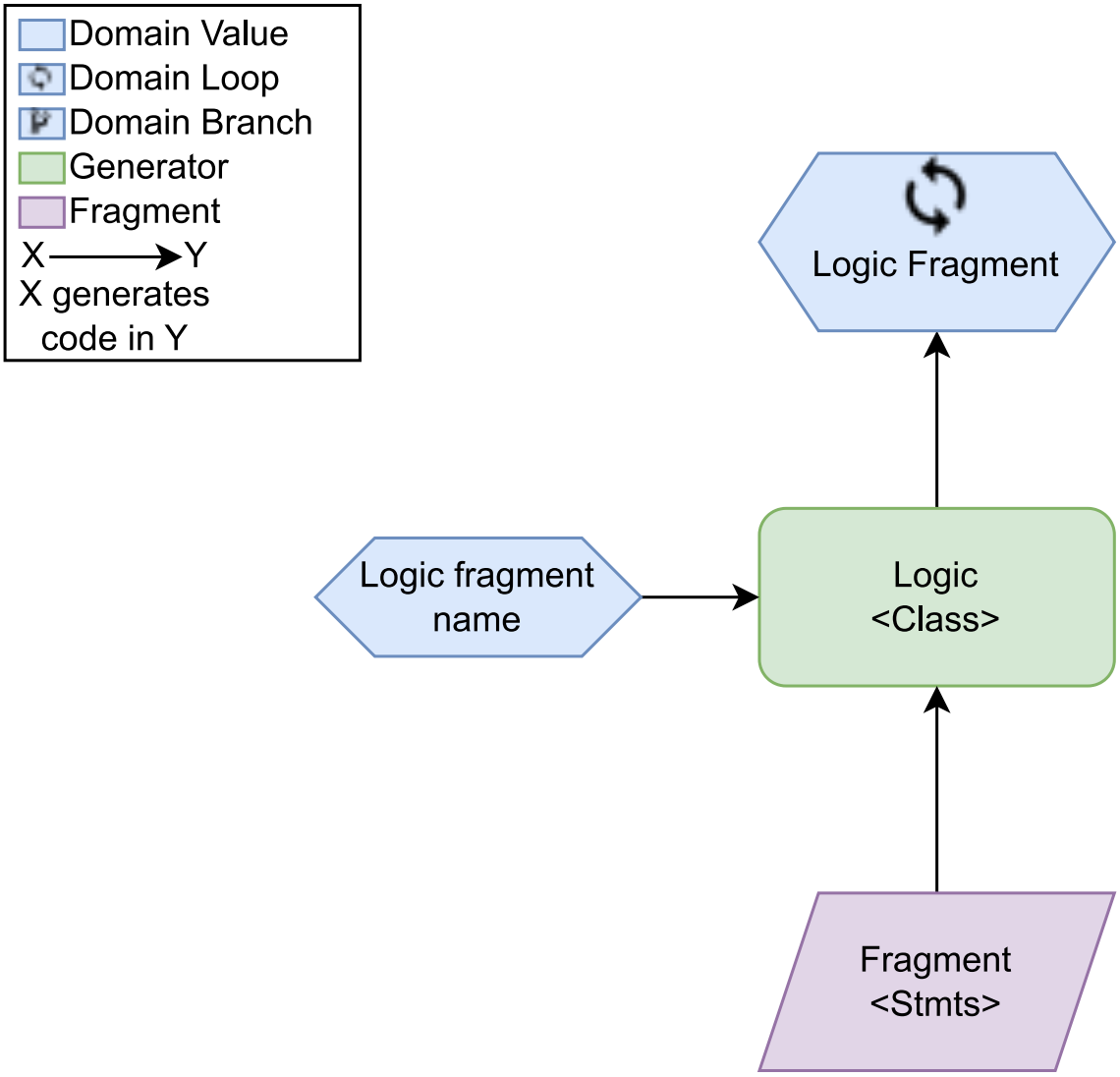

Figure 5.5: User logic fragments synthesis tree

5.5.6 ROS Messages

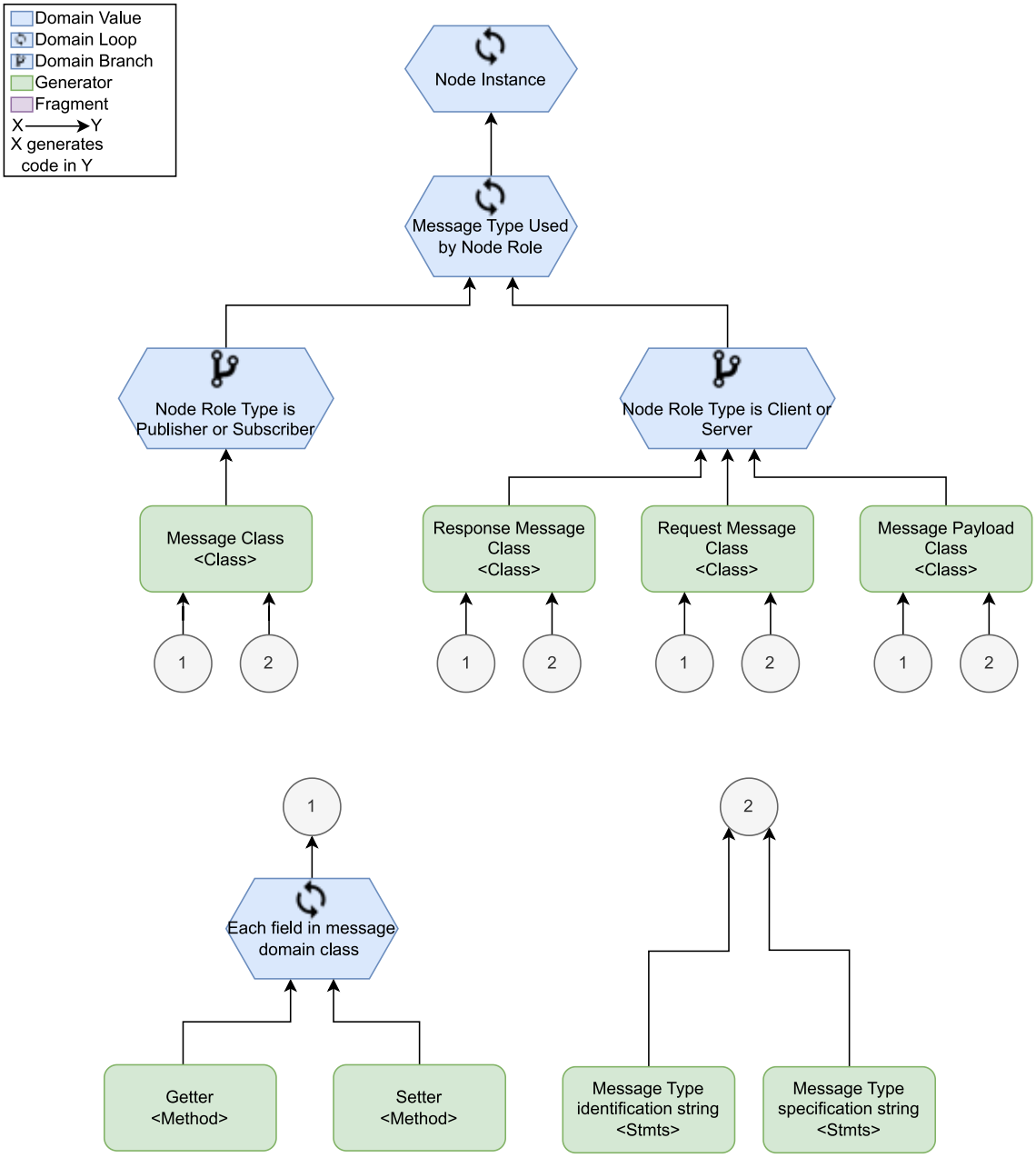

Figure 5.6: ROS Message synthesis tree

5.5.7 ROS Loop Class

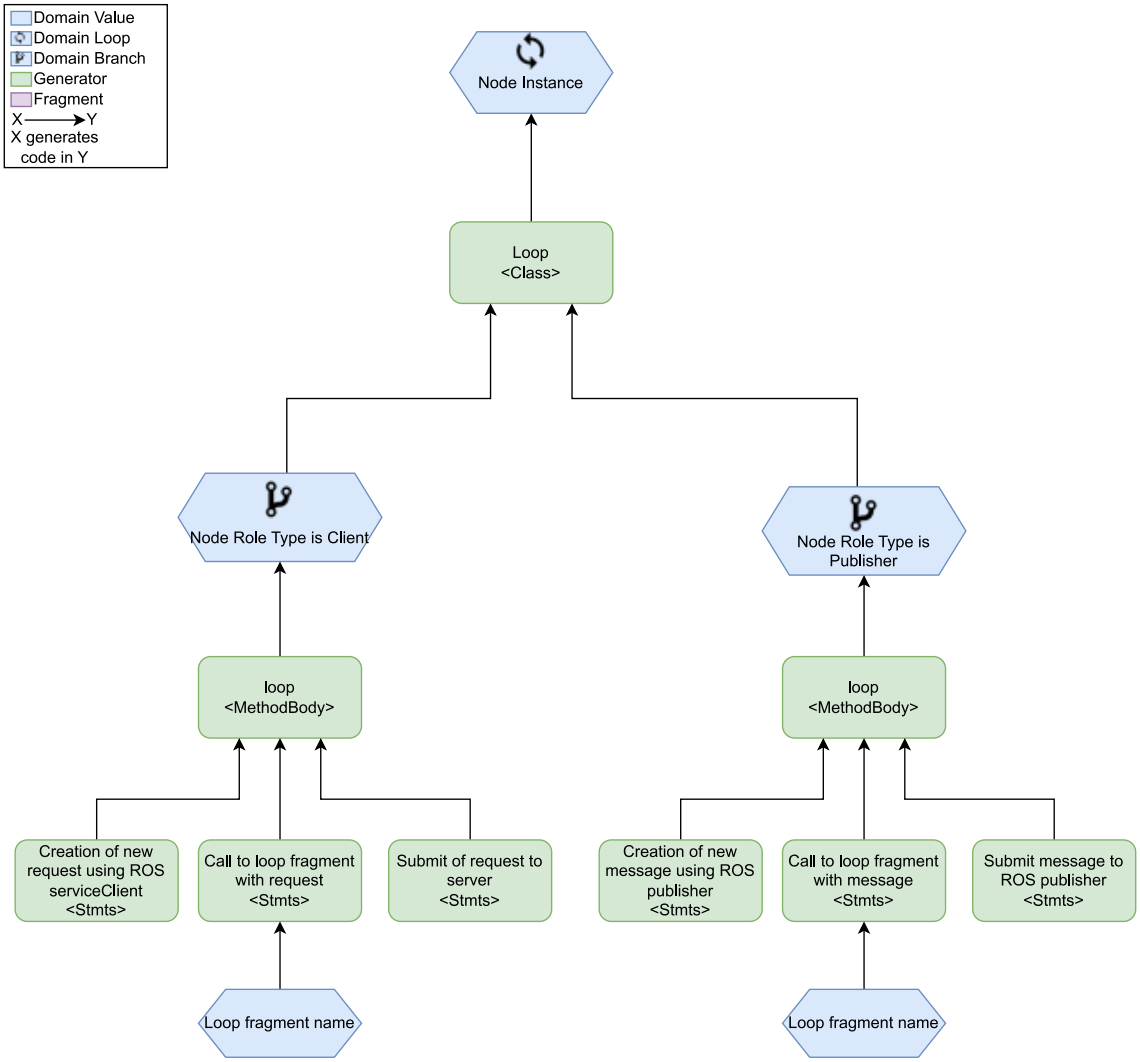

Figure 5.7: ROS Loop synthesis tree

5.6 Evaluation and Reflection

At the beginning of this case study we set out to synthesize as much ROS boilerplate code as possible. Our goal was to allow the user to supply their business logic have the generator produce most of the boilerplate required for node lifecycle management. In the end however we were able to completely synthesize ROS boilerplate related to node creation and communication with the constraint that each ROS node may only take on one communication role. We believe this constraint may be easily relaxed with a relatively small amount of additional development time. We are quite happy with the result and believe it demonstrates the depth with which one can apply CoGen and CDMS within a single domain.

Chapter 6

Case Study - 3D Rendering Domain

6.1 Overview

In this case study we consider the synthesis of boilerplate code in the domain of 3d rendering, specifically code required to make use of the Vulkan Rendering API. The high-level goal of this case study is to demonstrate the synthesis of code which may be injected into an existing large codebase. The primary benefit of this approach is that it gives the engineer familiar control over design and code structure while still allowing them to request the synthesis of boilerplate code in extremely targeted circumstances.

One key challenge encountered and addressed by this case study is the issue of defining a clear interface between synthesized and existing code. If the user requests to generate some code via CoGen, how do they then know how to interface with the generated code? In this case study we solve this problem by synthesizing methods that return opaque handles to resources and operate on predefined synthesized support objects.

Finally in this case study we focus purely on modeling of the target domain in order to demonstrate the usage of CDMS in purely an analytical mode.

6.2 Target Description

In this case study we start with a Java port of the classic Vulkan C++ tutorial "vulkantutorialjava". We chose Vulkan as the API is known for its high degree of boilerplate code. For example, in order to render a triangle on the screen around 1000 lines of boilerplate code are required. The positive tradeoff of this complexity is that Vulkan facilitates incredibly low-level and performant access to hardware resources.

6.3 Domain Description

In the 3d graphics rendering domain, a key concern is to accelerate the rendering of images on a screen by specifying computation in the form of highly concurrent programs called shaders. These shader programs are then run on some form of specialized accelerator hardware (usually a GPU). Hardware and software manufacturers have built many different APIs for running shaders on their respective hardware. Early APIs such as Directx 9 and OpenGL presented an abstract and highly synchronous interface to GPU hardware. This approach required that less code be written by the engineer but was limited in that it did not expose low-level hardware features and execution flow

control. More modern APIs such as Directx 12 and Vulkan provide extremely granular access to hardware functionality but at the cost of an increased degree of boilerplate code.

6.4 Application of Methodology

6.4.1 Distillation - 1

Upon initial examination of the target codebase we observed a number of resource types used in rendering.

- PhysicalDevice
- GraphicsQueue
- PresentQueue
- VertexBuffer
- IndexBuffer
- UniformBuffer
- RenderPass
- DescriptorPool
- DescriptorSet
- GraphicsPipeline
- CommandPool
- CommandBuffers
- Barrier
- Fence
- Graphics
- Shader
- Swapchain

For each of the objects above, the codebase contained a method to create or, in the case of uniform buffer, update the associated objects.

6.4.2 Modeling - 1

By applying existing knowledge of the 3d rendering domain, we were able to decide on a subset of observed objects to be codified explicitly in our domain model. Firstly we included Shader and GraphicsPipeline as explicit objects in our domain model. We made this decision as these two primitives encapsulate the primary concern of our domain namely "what we want the GPU to do". A gpu pipeline generally contains so called fix function stages such as Rasterization. To encapsulate configuration details of the rasterization stage we introduced a new class called RasterizationPolicy which is to be passed into the GraphicsPipeline object on instantiation.

Next, we turned our attention to resources that may be directly accessed by a shader. Such resources include any form of buffer or Image/Texture. For maxmimum flexibility we decided to

divide buffers up into two types, namely AdhocBuffer and BulkBuffer. Our objective was to have AdhocBuffer take on the role of Uniform Buffer in the original example and BulkBuffer the role of vertex and index buffer. This bifurcation of Bulk and Adhoc is useful as sending large sets of data (Bulk data) such as mesh vertices to the GPU usually involves different considerations and constraints when compared to the transfer of nonbulk data (Adhoc data). Additionally, when rendering, mesh data is often not updated frequently unlike adhoc data such as object transformations which are generally updated frequently (i.e every frame). This difference in role was also quite visible in the target codebase where entirely different code existed for the management of vertex/index buffers compared to uniform buffers.

The next object considered was Texture. We decided to represent Texture in our domain model with a class called Image. The purpose of the image class was to abstract away implementation details of a typical image data structure. For example, in our example code an image could be a framebuffer, render target, or texture. Our image object on the other just codifies the essence of these objects namely information related to a height, width, and format.

With our foundational set of objects in place, it was time to think about how to use them in our domain model. The high-level approach chosen was to allow direct instantiation of Shaders, GPUPipelines and associated Buffer/Image resources but with the assertion that Adhoc/Bulk buffer and Image instances represent types of objects that could be created at runtime and instances of GPUPipeline and Shader represent actual specific instance of a rendering resource i.e map 1-1 with their runtime equivalents.

6.4.3 Synthesis - 1

As seen in distillation 1, our example codebase involves the creation, updating and destruction of many different resource types. For this reason our first distillation task was the generation of a Catalog class for storing object instances and providing the user with handles to these objects as desired. A benefit of encapsulating object instances in a catalog is that it minimizes interference with existing code.

In order to further reduce interference we decided to build our synthesized code an an Opaque Handle based architecture similar to the Win32 API. Here we return opaque handles to created objects from various factory functions. This handle may then be passed into further synthesized helper functions to perform operations involving an object instance. Again the primary motivation behind this design decision was reduction of interactions with existing code.

With our synthesis architecture in place, we decided it would be best to directly synthesize factory methods for either specific Buffer and Image archetypes or 1-1 instances of Pipelines and Shaders.

6.4.4 Modeling - 1.1

During completion of Synthesis - 1 we noted that the user should be given some degree of control over how objects were created and associated object lifetimes. To accomplish this goal we added a new class to our model called CreationPolicy. Upon instantiation in the domain model, object types must be provided a CreationPolicy. This policy would then influence the code generation to create the target object instance.

6.4.5 Distillation - 2

With resource creation underway, we began a second past of code analysis this time with an eye towards to resource usage. In order to achieve the goal of "running code on the GPU" we need to be capable of binding both Shaders and shader resources to the GPU and also updating certain resources at runtime (for example updating the contents of a buffer). In our target codebase resources are created up front before rendering and later bound to the GPU upon rendering each

frame. This observation leads us to the idea of synthesizing a binding method and when applicable, update method for each resource type.

6.4.6 Modeling - 2

No further additions to our domain model were required at this step.

6.4.7 Synthesis - 2

In order to facilitate maximum flexibility, we built our binding and updating methods around a command list architecture. The idea with said architectures is that we construct a list of commands to be send to the GPU and then submit this list of commands using a special method call at a later point in time. First and foremost we introduced two new methods one to create a new instance of a command list and another to submit a command list. Next we introduced methods to enqueue resource binding, draw and update commands to a given command list.

6.4.8 Distillation - 3

To finalize our synthesis we turned our attention to higher-level rendering code namely initialization code, cleanup code and per-frame boilerplate. In our target codebase, cleanup and init tasks were handled by two respective methods. We decided to keep this design and simply synthesize both methods directly. We also observed code that was run before and after every frame. This code primarily involved synchronization and acquisition of the next image to render from the swapchain.

6.4.9 Modeling - 3

The per-frame code mentioned above involved interesting parameters such as number of frames allowed to be in flight at once. Additionally, it is possible that the user would like to run certain rendering processes in parallel (such as compute and graphics shaders). For this reason we extended our domain model with two new classes designed to encapsulate the aforementioned details. The classes in question are named ConcurrencyPolicy and SynchronizationPolicy.

6.4.10 Synthesis - 3

In the final synthesis pass we introduced four new methods, beginFrame, endFrame, initRendering and destroyRendering. beginFrame and endFrame implement boilerplate related to per-frame synchronization and other miscellaneous tasks. InitRendering and destroyRendering handle common Vulkan initialization and cleanup tasks.

6.5 Results

6.5.1 Artifacts

- Domain Model Location: Link

6.5.2 Domain Model

Below is a UML diagram of the domain model we converged upon.

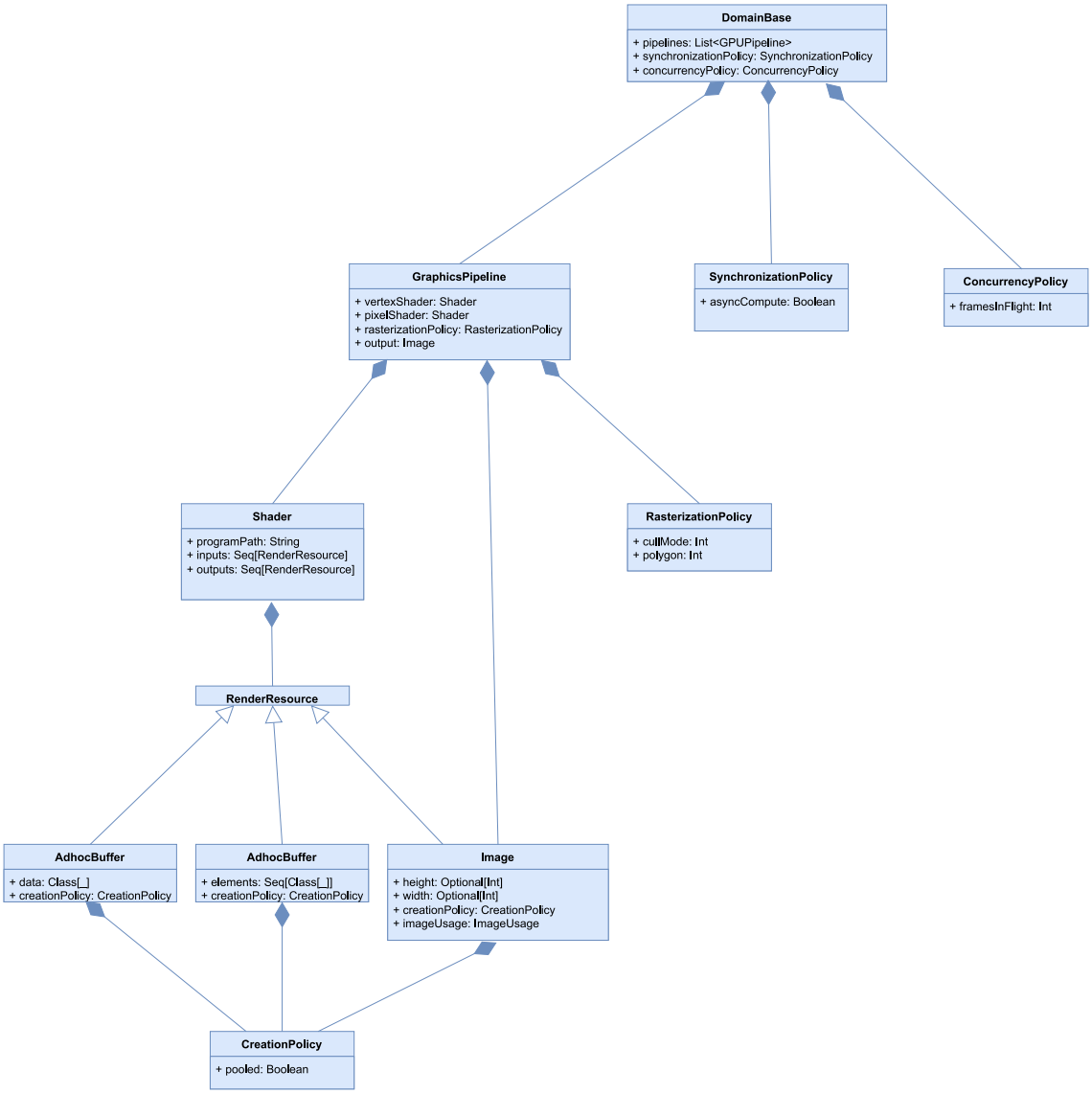

Figure 6.1: Rendering domain UML diagram

6.5.3 Vulkan App

Below is a sketch of a possible synthesis approach to achieving the goals of this case study.

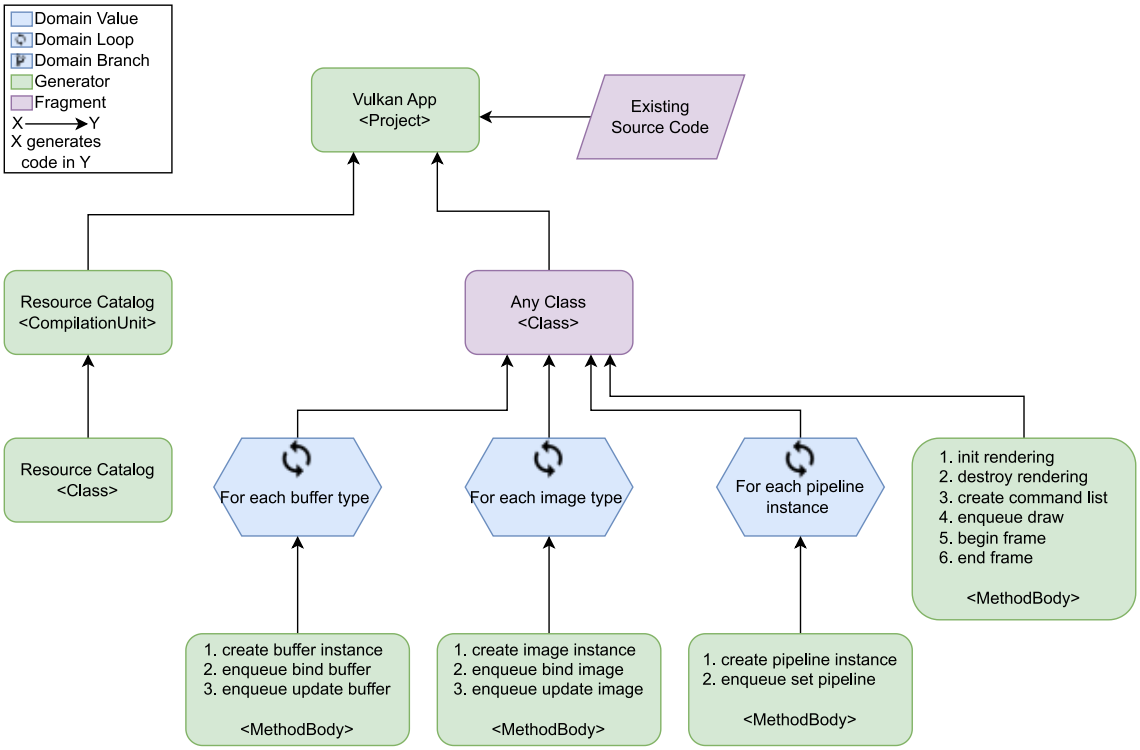

Figure 6.2: Vulkan synthesis tree

6.6 Evaluation and Reflection

At the start of this case study we set out to validate whether CDMS could be applied to a highly complex domain such as 3d rendering in Vulkan and in a manner that posing minimal interference with an existing system implementation. We believe we were able to achieve this goal. With further time investment, we believe it would be straightforward to implement a CoGen synthesis specification for the given domain model and synthesis tree. Additionally, we believe the approach could be modified to reflect a higher degree of object orientation without negatively affecting code interference.

Chapter 7

Evaluation

In order to gather evidence of the effectiveness of CoGen and CDMS we applied both technology and methodology to three varied case studies. During case study execution, results were documented in the form of a written development journal. In addition, metrics involving Lines of Code written were also gathered.

7.1 Quantitative Analysis

In order to analyze the results of our case studies quantitatively, we leverage a metric called LOC or lines of code. This metric is defined as the number of lines source code (not including comments and blank lines) which constitute a given set of source code files. Note that we will only consider the results from case studies 1 and 2 as case study 3 did not involve the implementation of synthesis specifications in CoGen.

Now it is important to note that the metric of LOC has many limitations and is not entirely useful in a general sense. In the case of this analysis, however, we simply present LOC as a way of grounding our work and acting as a basis for further discussion not as a metric to prove any particular claim about our results. With regard to this general discussion, we will use relative proportions of five different LOC statistics which we hope will provide some insight into the large body of coding related work backing this thesis.

Below is a table summarizing the five LOC metrics per case study.

Table 7.1: LOC Per Category

Case Study	**Original Codebase**	**Generated**	**Handwritten**	**Synthesis Specification**	**Domain Model**
GUI - JavaFX	47	68	0	577	30
GUI - QT	34	38	0	394	30
GUI - LibGDX	71	72	0	491	30
Robotics - ROS	375	295	49	1965	49

Below is a description of each data category.

- Original Codebase: LOC of the original case study codebase (before application of CDMS)
- Generated: LOC generated by CoGen.
- Handwritten: LOC of business logic written by hand.

- Synthesis Specification: LOC of the CoGen implementation providers which implement the case study synthesis specification.
- Domain Model: LOC of the final domain model consumed by the implementation provider.

The data above contains a number of interesting patterns. Firstly one may note that in the case of JavaFX, QT and LibGDX, the amount of code generated is greater than the original codebase size. In the case of JavaFX this is due to the synthesis of an extra class to encapsulate Text element information, and then additional code to instantiate instances of this class. In the case of QT the difference is due to the flattening of code structure. Finally, the difference in LibGDX is simply the result of minor code formatting differences. In the case of ROS we see that the amount of generated code is actually less than the original codebase size. This is again due to differences in code formatting.

We will now consider a number of more interesting observations. The first such observation is related to the drastically larger size of the synthesis specification as compared to generated output. The first cause of such a size difference lies in, ironically, boilerplate code required to interact with the CoGen framework. This boilerplate code accounts for around 171 LOC and does not vary with the amount of code generated or the domain. Boilerplate aside, the remaining bulk of synthesis specification code comes down to the generality of the synthesis process. With little modification, each synthesis specification could be changed to generate code in entirely different programming paradigms (such as functional, imperative etc) or programming language. This generality requires a slightly more verbose specification of code structure which we believe is a sufficient tradeoff. Another key driver of synthesis specification size is extensibility and support for multiple code structure permutations depending on the domain model. This is the primary driver of the large size of the ROS synthesis specification. In the generated example, three ROS nodes are generated. The current synthesis specification however is not limited to the generated of three nodes. The domain model may specify any number of nodes to generate. This example shows that as the domain model is leveraged to a greater and greater degree, eventually it is likely that the amount of code generated will surpass synthesis specification size.

7.2 Qualitative Analysis

In order to evaluate our results in a qualitative manner, we perform a reflective written analysis. Naturally, such an approach comes with major drawbacks including but not limited to a high degree of subjectivity and potential difficulties related to reproducibility of results. In the case of our results, however, we believe the process of executing each case study was documented thoroughly enough as to provide a clear line of support for each of the claims made below.

In the first case study, we achieved complete synthesis of lifecycle and environment configuration boilerplate required to utilize three GUI frameworks. In addition, we were able to synthesize boilerplate in all cases to support layout of text elements in a grid pattern. We believe these results show evidence of the ability of CoGen and CDMS to leverage a single domain model in the generation of boilerplate code for multiple contexts.

In the second case study, we generated all boilerplate required to facilitate the creation of ROS nodes, and communication between any number of ROS nodes via message passing in both pub/sub and client/server capacities. In addition, we generated all code required to initialize the ROS executor, instantiate desired ROS nodes and register them with the executor.

For the robotics and GUI case study, we succeeded in synthesizing well-formated, clear and compilable code, with the exception of two minor instances where CoGen had not yet implemented the ability to leverage two minor language features.

In the third case study, we focused purely on the modeling capabilities of CDMS. We believe that our resulting domain model and (diagram only) synthesis specification clearly demonstrated the

scalability of CoGen and CDMS to even extremely granular and customizable Vulkan boilerplate code.

We believe these case studies demonstrated a positive result with regard to the usefulness and applicability of our approach.

Each case study demonstrated the high degree of capabilities offered by CoGen including the generation of generics and other metaprogramming constructs, and the generation of complex embedded data formats in the case of ROS messages. We also demonstrated the ability of our approach to scale and remain usable across multiple varied domains. This concludes our evaluation.

Chapter 8

Conclusion and Future Work

In this thesis, we evaluated the effectiveness of generating boilerplate code in three real-world domains using the CoGen synthesis framework and CDMS methodology. Although the construction of synthesis specifications was time-consuming, we believe our results are a clear indicator that the foundational theory underlying the approach is valid and could be applied effectively to many more real-world domains. In the first case study, we achieved effective synthesis of boilerplate code across three target implementations from a single domain model. In the second case study we demonstrated synthesis of core boilerplate code required to deploy business logic in a ROS environment. In the final case study, we demonstrated an additive approach to synthesis wherein code was injected into an existing codebase in such a manner that the engineer could leverage generated code with zero interference with pre-existing code. We believe these three scenarios provide sufficient coverage of what is possible using CoGen and our synthesis methodology. We now turn our attention to a number of topics we would like to explore in future work.

8.1 Domain Model Composition

In each case study we developed a single domain model, however, in many real-world situations, combining domain models for the generation of a single project may be useful especially if the goal is to reuse code generation across multiple projects. We would like to explore this line of reasoning further as it would be highly useful in the context of open source and scaling our approach.

8.2 Domain Model and Synthesis Specification Reuse

Under the circumstances that our approach is adopted by a larger community, the ability to efficiently reuse and share domain models and synthesis specifications amongst individuals and organizations in a scalable way will become of utmost importance. Additional research and development will be required to find the best way of facilitating such growth. Potential areas of research could include, creating searchable public repositories of domain models and synthesis specifications and additional IDE tooling to facilitate easy access and usage of public artifacts within development workflows.

8.3 Generating Game Engine Components

Inspired by the 3D rendering domain, we would like to explore the synthesis of various game engine components and subsystems including platform-specific renderer functionality, and entity/-world representation mechanisms such as ECS. This feat would explore the usage of CoGen in a

more top-down/modeling-centric manner and also test CoGen's practical ability to generate high-performance code (a strict requirement of most game engines). Finally, the synthesis of specific game engine concerns would better test the scalability of synthesis specifications to the generation of larger codebases. We suspect that by synthesizing a codebase on the scale of a game engine, we will discover new and powerful design patterns and methods of structuring synthesis specifications that scale better to the greater size.

8.4 IDE Tooling

Our approach presents ample opportunity for IDE-supported enhancements to both CoGen and CDMS workflow. Some ideas for future IDE tooling include:

- Inline identification of generated code vs non-generated. Allow for instant regeneration with one click.
- Automatic creation of synthesis specifications corresponding to highlighted code.
- Automatic integration of generated code into existing project structures.

Bibliography

[1] Brooks, Frederick P. (1986). "No Silver Bullet—Essence and Accident in Software Engineering". Proceedings of the IFIP Tenth World Computing Conference, pp. 1069–1076.

[2] CoGen. [Online]. Available: https://github.com/combinators/expression-problem.

[3] Wirth, Niklaus. (1971). Program development by stepwise refinement. Commun. ACM, 14(4), 221–227. https://doi.org/10.1145/362575.362577.

[4] Lämmel, Ralf and Peyton Jones, Simon. (2003). Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, 26-37. https://doi.org/10.1145/604174.604179.

[5] Ghazarian, Arbi. (2015). A theory of software complexity. Proceedings of the Fourth SEMAT Workshop on General Theory of Software Engineering (GTSE '15). IEEE Press, pp. 29–32.

[6] McCabe, T. J. (1976). "A Complexity Measure". IEEE Transactions on Software Engineering, SE-2(4), 308-320. https://doi.org/10.1109/TSE.1976.233837.

[7] Heineman, G. T., Bessai, J., Düdder, B. (2022). EpCoGen Journal. ms, WPI, Worcester, MA USA; Technische Universität Dortmund, Germany; University of Copenhagen, Denmark. (IN PROGRESS).

[8] Gulwani, Sumit; Polozov, Oleksandr; Singh, Rishabh. (2017). "Program Synthesis". Foundations and Trends® in Programming Languages, 4(1-2), 1-119. https://dx.doi.org/10.1561/2500000010.

[9] Rich, C.; Waters, R.C. (1993). Approaches to automatic programming. In M.C. Yovits (Ed.), Advances in Computers (Vol. 37, pp. 1–57). https://doi.org/10.1016/S0065-2458(08)60402-7.

[10] Solar-Lezama, Armando. (2013). "Program Sketching". International Journal on Software Tools for Technology Transfer, 15(5–6), 475–495.

[11] Prose framework. Microsoft Research. (2022, July 22). [Online]. Available: https://www.microsoft.com/en-us/research/project/prose-framework/.

[12] Guo, Z.; Cao, D.; Tjong, D.; Yang, J.; Schlesinger, C.; Polikarpova, N. (2022). Type-Directed Program Synthesis for RESTful APIs. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).

[13] Polikarpova, Nadia; Kuraj, Ivan; Solar-Lezama, Armando. (2016). Program synthesis from polymorphic refinement types. SIGPLAN Not., 51(6), 522–538. https://doi.org/10.1145/2980983.2908093.

[14] (1999). Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., USA.

[15] Sutton, Richard S.; Barto, Andrew G. (2018). Reinforcement Learning: An Introduction. A Bradford Book, Cambridge, MA, USA.

[16] Google. (n.d.). API reference — protocol buffers. [Online]. Available: https://developers.google.com/protocol-buffers/docs/reference/overview.

[17] MDAOG. [Online]. Available: https://mdaog.sourceforge.net/.

[18] SpyrosKou. (n.d.). Plain-ros-java-system-example. [Online]. Available: https://github.com/SpyrosKou/Plain-ROS-Java-System-Example/tree/main/src/main/java/eu.spyros.koukas.ros.examples.

[19] Overvoorde, A. (n.d.). Introduction - Vulkan Tutorial. [Online]. Available: https://vulkan-tutorial.com/.

[20] Naitsirc98. (n.d.). Vulkan-tutorial-java. [Online]. Available: https://github.com/Naitsirc98/Vulkan-Tutorial-Java.

[21] Simulink Coder. MATLAB. [Online]. Available: https://www.mathworks.com/products/simulink-coder.html.

[22] ChatGPT. [Online]. Available: https://chat.openai.com/?model=gpt-4.

[23] GitHub copilot. [Online]. Available: https://github.com/features/copilot.

[24] Evans, E.; Fowler, M. (2019). Domain-driven design: tackling complexity in the heart of software. Addison-Wesley.

[25] Evans, E. (2015). Domain-driven design reference: Definitions and patterns summaries. AbeBooks.

[26] ParaLock. (n.d.). Fork of CoGen. [Online]. Available: https://github.com/ParaLock/CoGen.

[27] ParaLock. (n.d.). Case Study Repo. [Online]. Available: https://github.com/ParaLock/WPI-Masters-Thesis-Workspace.

[28] IBM. (n.d.). Rational rose model. [Online]. Available: https://www.ibm.com/docs/en/rational-soft-arch/9.5?topic=migration-rational-rose-model.

[29] Sparx Systems. (n.d.). Enterprise architect. [Online]. Available: https://sparxsystems.com/products/ea/.

[30] Android Developers. (n.d.). Android Layout Editor. [Online]. Available: https://developer.android.com/studio/write/layout-editor.

[31] FormDev Software. (n.d.). JFormDesigner - Java/Swing Gui Designer. [Online]. Available: https://www.formdev.com/jformdesigner/.

[32] Yacc. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9699919799/utilities/yacc.html.

[33] Kim, Moonjoo; Viswanathan, Mahesh; Ben-Abdallah, Hanêne; Kannan, Sampath; Lee, Insup; Sokolsky, Oleg. (1970). MaC: A Framework for Run-Time Correctness Assurance of Real-Time Systems.

[34] Bohrer, Rose; Tan, Yong Kiam; Mitsch, Stefan; Myreen, Magnus O.; Platzer, André. (2018). VeriPhy: verified controller executables from verified cyber-physical system models. SIGPLAN Not., 53(4), 617–630. https://doi.org/10.1145/3296979.3192406.

[35] University of Oxford. (n.d.). Tackling the ethical dilemma of responsibility in large language models. [Online]. Available: https://www.ox.ac.uk/news/2023-05-05-tackling-ethical-dilemma-responsibility-large-language-models.

[36] Learn scala. Scala Documentation. (n.d.). https://docs.scala-lang.org/#