CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Realizing Privacy-Aware Data Flow Diagrams In Java

Master's thesis in Computer science and engineering

Theodor Angergård
Tobias Karlsson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Realizing Privacy-Aware Data Flow Diagrams In Java

Theodor Angergård
Tobias Karlsson

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Realizing Privacy-Aware Data Flow Diagrams In Java

Theodor Angergård
Tobias Karlsson

Supervisor: Gerardo Schneider, Department of Computer Science and Engineering
Supervisor: Sandro Stucki, Department of Computer Science and Engineering
Examiner: Magnus Myreen, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

Abstract

Privacy by Design is an approach to designing systems at every step of the way to respect people's personal data. Alshareef et al. defined rules for taking a Data Flow Diagram, which can be a good tool for designing functional aspects of systems, and introduced transformation rules that add new nodes and edges that bring the non-functional aspect of privacy. The result of transforming a Data Flow Diagram is a Privacy-Aware Data Flow Diagram that would force the developer to design with privacy in mind.

However, the Privacy-Aware Data Flow Diagram was only sketched up in theory and never put to the test, which is what this thesis changes. We did this by first designing an algorithm called Ray that generates code from a Data Flow Diagram. Then extending this functionality with another algorithm, called Holt, to support most of the ideas with the PA-DFD. These two algorithms, along with a new data structure we call Holt Privacy-Aware Data Flow Diagrams, are our contributions to one possible solution of realizing Privacy-Aware Data Flow Diagrams into runnable code. The code gets generated with the help of annotation processing in Java. We evaluate this solution at the end of the thesis with a runnable case study.

Keywords: Data Flow Diagrams, Privacy by design, Code generation, GDPR

Acknowledgements

We want to thank our supervisors, Gerardo Schneider and Sandro Stucki, for the great project proposal, the rewarding meetings, and for their expertise and continued support throughout the project.

We also want to thank our opponents Alexander Selmanovic and Camilla Sönderlund.

Theodor Angergård & Tobias Karlsson, Gothenburg, June 2022

Contents

List of Figures

1

Introduction

Since May 2018, all organizations need to be compliant with the General Data Protection Regulation (GDPR) if they want to continue to process the personal data of people within the European Economic Area [1]. The GDPR gave clear rights to the data subject, such as the right to be forgotten and the right to restrict data processing. The regulation increased the number of privacy features that software engineers had to implement for a GDPR-compliant system.

Privacy by Design (PbD) is an approach that software engineers can follow to ensure that they are compliant with the GDPR. PbD forces developer to think about these privacy requirements at every step of the design and development and not try to fulfill them at the end of a project [2][3]. The GDPR very explicitly mentions that developers must practice "Data protection by design and by default"; namely, Article 25 in the GDPR includes that the software designers have to put thought into their system and their designs to protect the rights of data subjects [1].

There are many ways of designing software architecture. One of them, the Data Flow Diagram (DFD), is a way of defining requirements with a graph consisting of nodes, called activators, and edges, called flows, that can represent a software system [4]. A DFD has three activator types: processes, databases, and external entities. Processes are a function that takes some input and computes an output, databases can save data between runs through the DFD, and external entities are entry points for actors outside the system to interact with the DFD. The edges are called data flows and connect the different activators. Figure 1.1 shows an example DFD of a system that saves order history from a store in a database.

One issue with DFD is that it does not consider privacy. No checks exist to ensure that the data getting processed by the activator are allowed to do so by the data subject. There is also no way to see if the subject agrees to retain their data indefinitely. The DFD does not consider the data subject's rights from the GDPR. Nevertheless, it has an excellent foundation for adding privacy checks, which is what Antignac et al. worked to accomplish in 2016 and 2018 with the formulation of Privacy-Aware Data Flow Diagram (PA-DFD)[5][6].

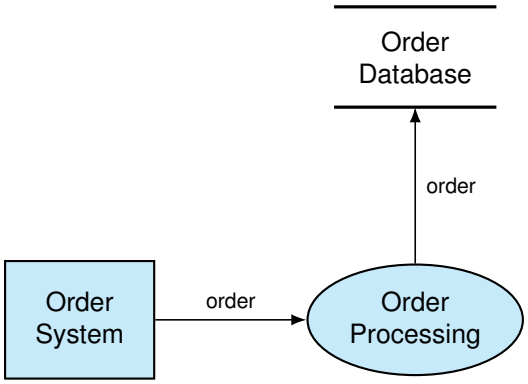


Figure 1.1: Simple DFD storing order history in a database.

A PA-DFD gets generated by a transformation from a DFD. It does this by following a set of rules first described by Antignac et al. and then further evolved in Alshareef et al. [5][7]. The defined rules describe, for example, that before some data gets read from a database to a process, there needs to be a check that makes sure that the data subject for that piece of data has agreed to the specific processing.

In this thesis, we will design two frameworks called Ray and Holt that use a DFD as input and generate code that developers can use to incorporate the strength of PA-DFDs into their systems. We split this issue into two parts. First the generation of code from a DFD by designing a framework, then using that framework to realize PA-DFD.

1.1 Problem statement

The papers we base our work on left many aspects of PA-DFD abstract or undefined [5][6][7]. There is a direction towards the goal of PA-DFD, but not how things should work in practice, which is the main problem we want to solve. That, together with the fact that we want to ask ourselves if Ray and Holt can help developers, was the motivation behind our research question:

RQ 1. How can we bridge the gap between PA-DFD and code?

To help answer this question, we have three sub-questions:

- *RQ 2. What sub-algorithms does our algorithm need?*

  Several steps will need to be taken to go from a PA-DFD to generated code. We need to clearly define these steps so that others can understand and maybe implement their own version. There is two specific technical questions that we want to answer that has to do with these algorithms:

  - *RQ 2.1. How can we represent policies and then enforce them?*
  - *RQ 2.2. How do we make sure that expired data is never processed?*

- *RQ 3. How do we ensure the correctness of our algorithm?*

  - *RQ 3.1. How can we test the implementation of our algorithm?*
  - *RQ 3.2. What requirements do we need to put on developers using our algorithm?*

- *RQ 4. How do we ensure the implementation of our algorithm is usable?*

1.2 Goals and challenges

Throughout this section, we will state three contributions that we want the reader to consider when going through our thesis. These are our primary research contributions.

Before being able to support a PA-DFD in any matter, we needed a DFD framework to work off. We could not find a tool to help us generate code from a DFD, so we indirectly added the goal to design and create that before starting to realize the PA-DFD solution. A vital requirement of the design was to separate the different stages so that our PA-DFD solution could intercept and modify to its liking. For example, creating new activators or moving flows. This description leads us to Contribution 1:

> **Contribution 1: We have designed a flexible and functional DFD framework, called Ray.**

Contribution 1 came with many challenges, such as trying to mitigate the technical debt accumulated throughout the project due to its size, many functions, and the drive to keep things modular. The reader will experience the completeness of the DFD solution through the rest of the thesis due to it being the backbone of everything.

Contribution 2 is a general contribution that we believe we have designed a solution that can help with privacy problems. However, it is essential to separate the work done before us and the work we contribute. Section 2.2 describes the previous work done, which was crucial for enabling us to make our contribution.

> **Contribution 2: Holt, together with Ray, can help developers to design with privacy in mind.**

Regarding Contribution 2, we have created a functional solution to realize the idea of PA-DFD into runnable code. We think we have made exemplary contributions in this regard, which Chapter 4 will go through more thoroughly. A challenge was that we did not think of the solution every step of the way with the PA-DFD when designing Ray. We sometimes went back to Ray to modify and add functionality that would help Holt, but it was a challenge to keep the two separated all the time.

We believe with Contribution 3 that our PA-DFD solution is flexible enough to let developers use their policy representation. Chapter 4 will have some examples of policy representation. Section 5.3 talks about other policy representations and how they could theoretically be used.

> **Contribution 3: The flexible design of Holt lets developers use their representation of policies to enforce privacy.**

One overall goal was to give concrete and understandable examples throughout the thesis to drive home the advantages of Ray and Holt while still stating the potential issues that could arise when developing. Many aspects from previous papers were left abstract to let the reader fill in the blanks with their ideas, including what a policy representation could be. It was a challenge to make these examples good to let the reader also imagine their problems and how they could use their business domain to represent solutions.

When discussing usability, we had the challenge when thinking about the contributions about how we could design something as little intrusive as possible. For both designs, Ray and Holt, we wanted to make sure that the developer did not have to give up too much flexibility to use them. Good usability became another overall goal for this thesis.

The code, along with some examples from this thesis, can be found in [8].

1.3 Methodology

The research questions will be undertaken using the Design Research methodology that was put forth by Hevner et al. [9].

When following this methodology, our work was divided into two phases; research and writing code, which were repeated many times. During the research phase, we researched, discussed different scenarios, experimented, and, in the end, formulated a plan for how we should execute the next coding phase. The second phase consisted of putting the research into code and continuously writing examples to verify that the newly added functionality worked as intended.

Our work consisted of three iterations, where each iteration used the Design Research methodology [9]. *Correctness* and *usability* has to be prioritized when evaluating the solution for each iteration. If ever needed to, we prioritized *correctness* over *usability*. After all iterations, we created a case study that helped us create an evaluation of our algorithms and implementations.

The three iterations were roughly the following:

1. Design of Ray.
2. Realization of the PA-DFD using Ray.

3. Refined both Ray and Holt by developing examples.

2
Background

The first two sections in this chapter provide a more in-depth description of the DFD and the PA-DFD. The goal of these descriptions is to be general. For example, the description for DFDs will be more general here, with more explicit constraints set when used in Chapter 3.

2.1 Data Flow Diagram

For a software system to be maintainable, the designers can use Structured Analysis. One part of Structured Analysis is the DFD [4]. The DFD is a helpful graphical tool to visualize and design the way data flows within a software system [4]. They consist of two general building blocks: *activators* and *flows*. The activators represent some form of data processing, while the flows are displayed as arrows and are used to visualize data flow between the activators. There are different categories for the activators that are:

- *Processes* - Shown as a circle and represents a transformation of some input to some output.
- *Database* - Shown as two horizontal parallel lines and represents the function to store data.
- *External entities* - Shown as a rectangle and represents something that is not part of the software system and is the start of all flows within the system.

When multiple flows create a use case, we have chosen to call it a traverse. A traverse has a single start activator, which has to be an external entity, and one of more end activator, which can either be external entities or databases. Figure 2.1 shows three flows that get combined into a single traverse and represent the use case of retrieving a user from a database.

Figure 2.1 shows an example DFD which contains all the elements that a DFD can have. The `User` rectangle is an external entity type and represents a collection of actions that external users can take. This activator is also what starts and ends this traverse. The `UserFormatter` circle is a processor that queries the `UserDB` activator using the data it received from the `User` activator and forwards the result back to the `User`. Finally, the arrows represent the data flow within the system, and together they create the *GetUser* traverse.

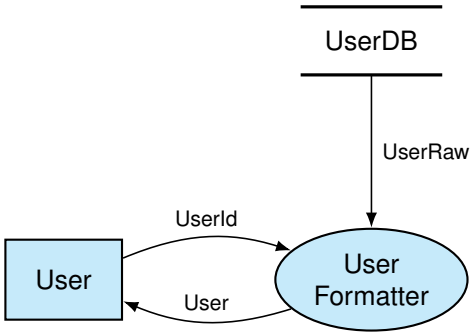
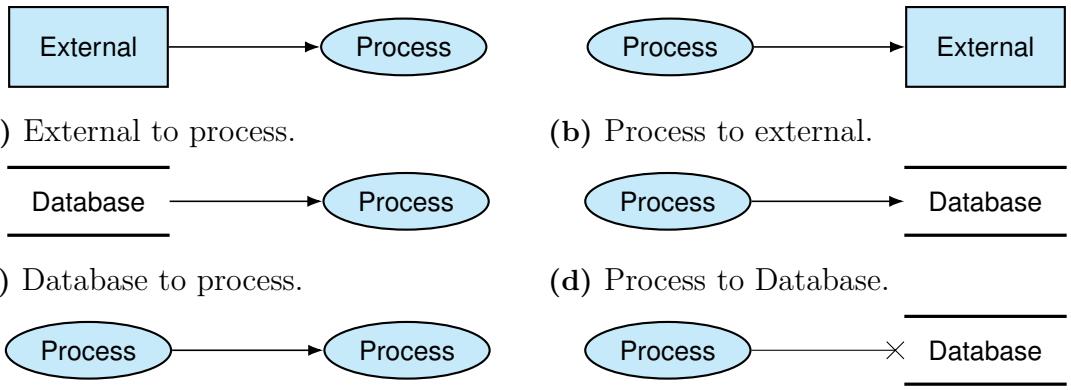


Figure 2.1: Example of an external entity used to recieve information about a user by providing its ID.

2.1.1 DFD rules

We will use the exact definition as Alshareef et al. defined [7]. Between what types of activators data can flow between can be seen in Figure 2.2 and will be further defined in sub-section 3.2.1.



(a) External to process.

(b) Process to external.

(c) Database to process.

(d) Process to Database.

(e) Process to process.

(f) Process to Database, delete.

Figure 2.2: Rules for DFD.

2.2 Privacy-Aware Data Flow Diagram

PA-DFD is an extension of DFD where the functionality remains, but new specific privacy checks get added. These new privacy checks focus on achieving non-functional properties such as "[...]purpose limitation and retention time, as well as to ensure accountability and policy management." [7]. The idea is to design a DFD, transform it into a PA-DFD, inspect it, and then generate a template of code where developers can add their business logic.

Antignac et al. defined rules for transforming a DFD into a PA-DFD for each flow between two activators, where new flows and activators are needed [5]. These rules were later refined by Alshareef et al. [7]. Figure 2.3 shows these rules, and the

algorithm for transforming is, in short, to go through each flow in the DFD and introduce the new activators and flows between those two original activators. Using these transformation rules, the DFD in Figure 2.1 would result in the PA-DFD seen in Figure 2.4.

In a regular DFD, the flows between activators only contained data, but now it can also contain policies that dictate processing limitations for the corresponding data. For example, a policy could limit an email not to getting used for a marketing purpose but allow it to send a one-time password. Another example of information that a policy could contain is how long the data subject has approved the storage of the data.

The key to achieving these non-functional requirements is to have a way to retrieve the relevant policies, decide whether the coming activator has the right to process them, and log the result of that decision. The new activators *Limit*, *Reason*, *Request*, *Log*, and *Clean*, are central to achieve this. There are loose ideas about the tasks for these activators, but each system is unique and has unique circumstances and requirements, which means that these PA-DFD-specific activators need to be flexible. The PA-DFD has the primary purpose of forcing developers to design their systems with privacy in mind at every step.

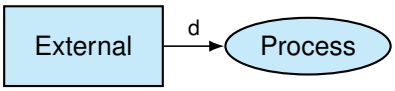
For each process activator, there is a related reason activator added. This activator gets policies as inputs and must decide its policy output depending on the process activator's behavior.

The request activator has to format the policies so that the limit activator can use them to make an active decision regarding the data. It receives policies from a policy database, external entity, or reason activator. For example, when querying from a database to a process activator, a query is also done from the corresponding policy database to a request activator.

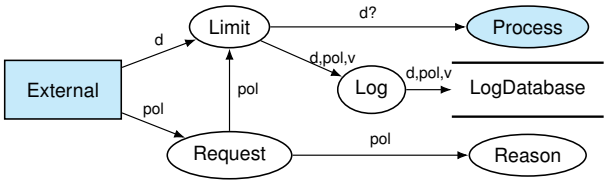
Accountability is an essential aspect of the GDPR, and it gets achieved by logging each limit decision with the log, and log database activators [1]. The log activator will receive the data, the policy connected to the data, and the limit activator's verdict. Since there is a log activator for each limit activator, we can distinguish them and log what and where things went wrong; or as expected.

The last activator is the clean activator, responsible for deleting data from databases whenever its related policy no longer allows it to be stored. An example could be if the purpose of the storage is for a particular event that has passed, which would mean that there is no longer any reason to store the data. The clean activator is only used in the `type:store` transformation seen in Figure 2.3h.

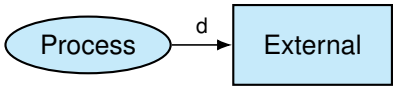
Something worth mentioning, and that the Figure 2.3 displays, is the fact that each data flow, in a sense, has a corresponding policy flow. These flows collide for the limit activator to decide if the data can continue flow through the graph.



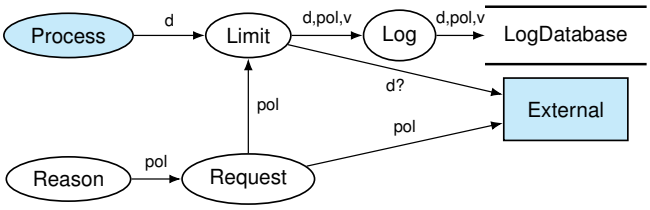
(a) DFD - External to process. `type:in`



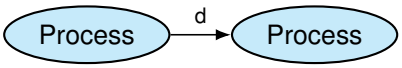
(b) PA-DFD - External to process.



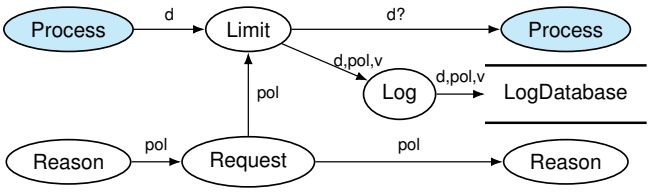
(c) DFD - Process to external. `type:out`



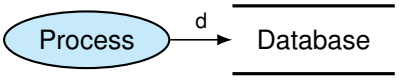
(d) PA-DFD - Process to external.



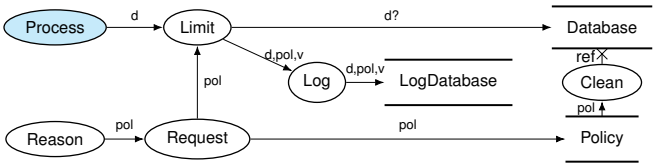
(e) DFD - Process to process. `type:comp`



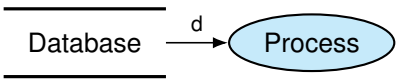
(f) PA-DFD - Process to process.



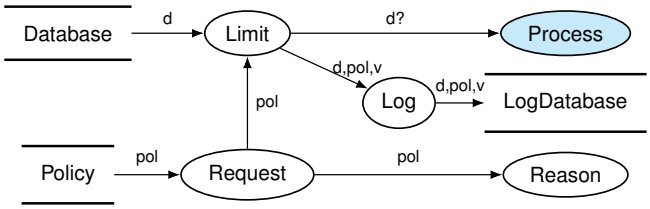
(g) DFD - Process to Database. `type:store`



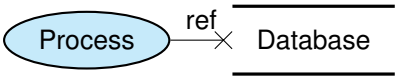
(h) PA-DFD - Process to Database.



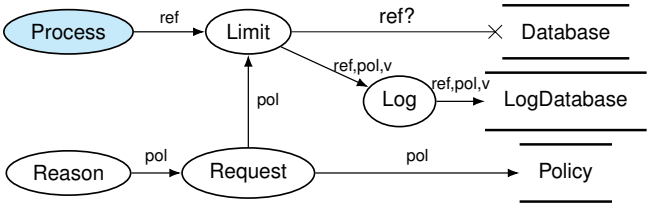
(i) DFD - Database to process. `type:read`



(j) PA-DFD - Database to process.



(k) DFD - Process to delete from Database. `type:delete`



(l) PA-DFD - Process to delete from Database.

Figure 2.3: Transformation rules from DFD to PA-DFD. Each DFD rule on the left side transforms into its respective PA-DFD rule on the right side.

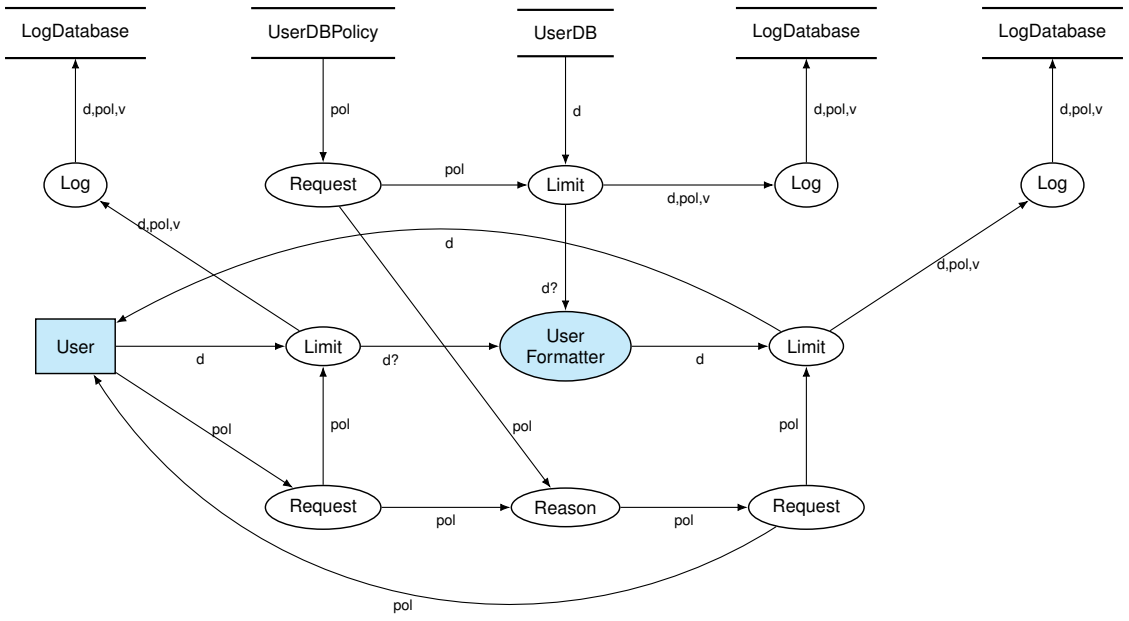


Figure 2.4: Example after transformation from Figure 2.1.

An important insight to have when reading this thesis is that the policies represent what the data subject approves and what the system can do with the connected data. Take, for example, a company that stores text files. One data subject could say that for a set of text files, they allow the system to use it to teach their neural network to find grammar errors better, while another data subject strictly says no to that.

Alshareef et al. were careful not to define what a policy could or could not be, which means that the developer still has to define their policy structure independently. This thesis will have examples where policies list predefined strings of the types of actions that the data subject has approved and the data's expiry date. We believe we can easier explain our algorithm with these concrete examples.

2.3 Tools

We use compile-time annotation processing from Java to generate code for the developer[10]. The developer needs to add our project as an annotation processing dependency and then start using our custom annotation to let Ray start loading in the diagrams and generating code inside the build directory of the developer's codebase. Since we get so tightly coupled with the developer's codebase via this processing, we can use the developer's own defined classes to generate code that fits within their ecosystem.

We also used the build automation tool Gradle to set up our codebase, and to generate correct Java code we used a library called JavaPoet [11][12].

3

Generating code from Data Flow Diagrams

We use annotation processing in compile time to generate runnable code from a DFD in Java. If a given DFD gets modified, change will be reflected in the generated code at recompile. Section 3.1 provides a step-by-step workflow of how we go from a DFD to executable code using our algorithm, *Ray*, making it easier to digest Section 3.2, which describes the DFD and algorithm specification we have chosen and why. Section 3.3 expands Section 3.1 with further examples. Finally, Section 3.4 reasons around the design decisions around the overall algorithm.

3.1 Workflow when using Ray

This section will go through an example of how a developer goes from a DFD to executable code. It is positioned early in this chapter to help put everything else in context, to explain how and why easier. We will base this example on the *Marketing* traverse in the DFD defined in Figure 3.1 and add Java annotations to complete the setup.

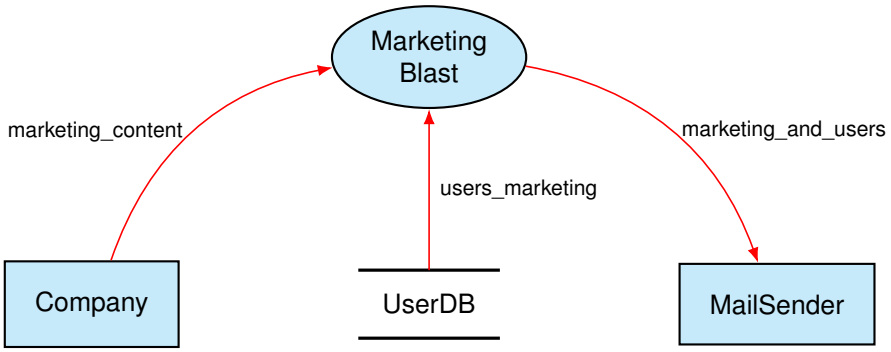


Figure 3.1: DFD with one traverse named *Marketing*, shortened `M`. It starts from the `Company` external entity and ends in `MailSender`.

Ray will execute based on specific annotations during compilation time to generate code. We will, throughout this overview, go through each type of the annotation and what their purposes are. Figure 3.2 shows the data model that will be used throughout this overview.

```java
public record Email(String email) { }
```

```java
public record EmailAndContent(Email email, EmailContent content) { }
```

```java
public record EmailContent(String content){ }
```

```java
public record User(Email email) { }
```

Figure 3.2: Definition of different classes that are used in the workflow overview.

The first step of the developer's workflow when using Ray is to design the DFD. This is done via the open source software *diagrams.net* [13]. The custom diagrams.net library which has DFD elements by Michael Henriksen is also used [14]. When the DFD design is done, it is exported without compression to an XML file. Figure 3.3 shows how Figure 3.1 looks when drawn in diagrams.net.

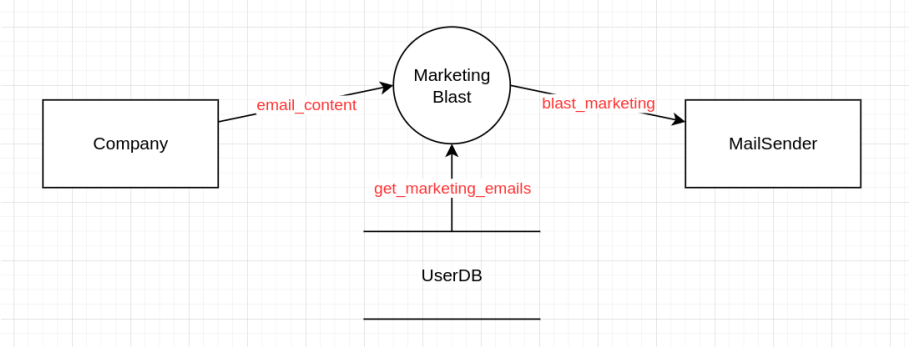


Figure 3.3: DFD drawn in diagrams.net.

Next the developer has to use the `@DFD` annotation and supply two values; the first value must be a unique name of the DFD, and the second is the path of the XML file from diagrams.net. This annotation can be put on any class but should, for clarity, be on the same class that the `public static void main()` method resides. Figure 3.4 shows the usage of `@DFD`.

```java
@DFD(name = "mail", xml = "mail-dfd.xml")
public class Main {
    public static void main(String[] args) {}
}
```

Figure 3.4: Usage of `@DFD`. `xml` references to the DFD specified in Figure 3.1 in an XML format.

After a recompilation, Ray will generate a Java class for each activator in the given DFD. That class is an interface for processes and databases and an abstract class for external entities. These generated classes will later contain the requirements needed to go through the traverses in the DFD. Figure 3.5 shows the generated code from the recompile as of right now.

```
@Generated
public abstract class AbstractCompany {
    public AbstractCompany() { }
}
```

```
@Generated
public abstract class AbstractMailSender {
    public AbstractMailSender() { }
}
```

```
@Generated
public interface MarketingBlastRequirements { }
```

```
@Generated
public interface UserDBRequirements { }
```

Figure 3.5: Generated abstract class and interfaces.

The next step is to create a class for each generated class, inherit it, and then annotate the created class with the annotation `@Activator`. Figure 3.6 has these new classes.

```
@Activator
public class Company extends AbstractCompany { }
```

```
@Activator
public class MailSender extends AbstractMailSender { }
```

```
@Activator
public class MarketingBlast implements MarketingBlastRequirement {}
```

```
@Activator
public class UserDB implements UserDBRequirements { }
```

Figure 3.6: Developer created classes.

With the classes generated, we will now focus on the class `Company` that is an external entity where we will specify the two traverses that start from this entity with the `@Traverse` annotation. Figure 3.7 adds two `@Traverse` annotations that specify the traverse's name and the execution order of the flows for that traverse. Figure 3.1 has two different traverses, `M` and `RP`. The flow IDs are named when designing the DFD, as seen in Figure 3.1. For example, the ID `marketing_content` is the flow between `Company` and `MarketingBlast`.

```
1  @Traverse(
2      name = "M",
3      order = {"marketing_content", "users_marketing",
       ↪  "marketing_and_users"}
4  )
5  @Activator
6  public class Company extends AbstractCompany {
7      public Company() { }
8
9  }
```

Figure 3.7: Usage of `@Traverse`. Note that the order of which the flows are specified is the order of execution. The traverses name is declared as a `public static final` to be accessible for other classes.

After a recompilation, several things will happen with the developer's codebase. First, in `AbstractCompany`, a traverse is generated, seen in Figure 3.8. The developer can now create `Company`'s methods that then can start this traverse, seen in Figure 3.9. `Company` needs to supply the references for `UserDB`, `MarketingBlast`, and `MailSender` via `super(...)` in its constructor. How traverses work is explained later in sub-section 3.2.4.

Second, in the generated interface `MarketingBlastRequirement` and two methods are generated, those and the developer-created class `MarketingBlast` can be seen in Figure 3.10. `M` takes the input from `Company` and the query result from `UserDB`. In `M` the input gets combined and forwards the result to `MailSender`. Figure 3.11 shows the code for `MailSender`. The query gets defined by `queryUserDBM` which returns a new interface seen in Figure 3.12.

By default, `Object` is used for all inputs and outputs since it is the root class of the Java class hierarchy, which leads to the issue of having to use `instanceof` and cast to the actual classes that the developer is sending between the activators. If the developer instead wants to use custom types, they can use `@Traverse`'s parameter `startTypes` for external entity and `@FlowThrough` for processes to change the signature of the generated methods. These parameters take an annotation `@Output` which has a parameter called `collection` that should be used if the developer want to use a collection of the specified type as output. `@Traverse`'s `startTypes` sets the

```java
@Generated("holt.JavaFileGenerator")
public abstract class AbstractCompany {
  /* ... */

  public AbstractCompany(UserDB userDB, MarketingBlast
   ↪  marketingBlast, MailSender mailSender) {
    /* ... */
  }

  protected final void M(EmailContent input0) {
    final var qd1 = this.marketingBlastRef.queryUserDBM(input0);
    final var v0 = this.marketingBlastRef.M(input0,
     ↪  qd1.createQuery(this.userDBRef));
    this.mailSenderRef.M(v0);
  }
}
```

Figure 3.8: `AbstractCompany` with generated traverse.

```java
/* ... */
@Activator
public class Company extends AbstractCompany {
    public Company(UserDB userDB, MarketingBlast marketingBlast,
     ↪  MailSender mailSender) {
        super(userDB, marketingBlast, mailSender);
    }

    public void sendMarketing(EmailContent content) {
        super.M(content);
    }
}
```

Figure 3.9: Code that use the generated traverse.

start type for the traverse, seen in Figure 3.13a, and `@FlowThrough` can set the output type for each method as well as what potential queries are to return, examples of both are seen in Figure 3.13b.

After specifying the annotations and recompilation, the requirements will get regenerated into what is seen in Figure 3.14. The developer can now add their business logic without having to handle `instanceof` and the potential that anything other than the expected can be the input inside their created activator classes, shown in Figure 3.15. After that, the developer can create a new instance of `Company` to call the wanted function, seen in Figure 3.16.

```java
@Generated("holt.JavaFileGenerator")
public interface MarketingBlastRequirements {
    Object M(Object input0, Object dbInput1);

    UserDBToMarketingBlastMQuery queryUserDBM(Object input0);
}
```

```java
@Activator
public class MarketingBlast implements MarketingBlastRequirements {
    @Override
    public Object M(Object input0, Object dbInput1) {
        if (dbInput1 instanceof Collection collection && input0
        ↪  instanceof EmailContent ec) {
            /* return ... */
        } else {
            throw new IllegalArgumentException();
        }
    }

    @Override
    public UserDBToMarketingBlastMQuery queryUserDBM(Object input0)
    ↪  {
        return UserDB::getAllUsers;
    }
}
```

Figure 3.10: `MarketingBlast` with the generated requirements class.

```java
@Generated("holt.JavaFileGenerator")
public abstract class AbstractMailSender {
  public AbstractMailSender() { }
  public abstract void M(Object input0);
}
```

```java
@Activator
public class MailSender extends AbstractMailSender {
    @Override
    public void M(Object input0) {
        if (input0 instanceof Collection collection) {
            /* ... */
        } else {
            throw new IllegalArgumentException();
        }
    }
}
```

Figure 3.11: `MailSender` with the generated abstract class.

```java
@Generated("holt.JavaFileGenerator")
public interface UserDBToMarketingBlastMQuery {
    Object createQuery(UserDB db);
}
```

Figure 3.12: Interface related to query from `UserDB`.

```
@Traverse(
        name = "M",
        startTypes = {@Output(type = EmailContent.class)},
        order = {"marketing_content", "users_marketing",
        ↪  "marketing_and_users"}
)
@Activator
public class Company extends AbstractCompany {
    /* ... */
}
```

(a) Setting output type in `@Traverse`.

```
@FlowThrough(
        traverse = "M",
        output = @Output(type = EmailAndContent.class, collection =
        ↪  true),
        queries = {
                @Query(
                        db = UserDB.class,
                        output = @Output(type = User.class,
                        ↪  collection = true)
                )
        }
)
@Activator
public class MarketingBlast implements MarketingBlastRequirements {
    /* ... */
}
```

(b) Setting output type in `@FlowThrough`.

Figure 3.13: Through annotations, class types are set to get rid of `Object`.

```
@Generated("holt.JavaFileGenerator")
public interface MarketingBlastRequirements {
  Collection<EmailAndContent> M(EmailContent input0,
  ↪  Collection<User> dbInput1);

  UserDBToMarketingBlastMQuery queryUserDBM(EmailContent input0);
}
```

Figure 3.14: New requirements after annotations that specify the class types.

```java
/* UserDB.java */
private final Map<Email, User> userMap = new HashMap<>();

public User getUser(Email email) {
    return userMap.get(email);
}

public List<User> getUsers() {
    return this.userMap.values().stream().toList();
}

/* MarketingBlast.java */
@Override
public Collection<EmailAndContent> M(EmailContent content,
 ↪  Collection<User> users) {
    return users.stream().map(user -> new
     ↪  EmailAndContent(user.email(), content)).toList();
}

@Override
public UserDBToMarketingBlastMQuery queryUserDBM(EmailContent
 ↪  input0) {
    return UserDB::getUsers;
}
```

Figure 3.15: Business logic without having to use `instanceof`.

```java
@DFD(name = "mail", xml = "mail-dfd.xml")
public class Main {
    public static void main(String[] args) {
        Company company = new Company(new UserDB(), new
         ↪  MarketingBlast(), new MailSender());
        company.M(new EmailContent("Marketing!"));
    }
}
```

Figure 3.16: Usage of `Company` external entity.

3.2 Algorithm specifications

With the understanding of how to use Ray from Section 3.1, we can now move towards a more formal specification. When choosing these specifications, we always kept Contribution 1 in mind.

3.2.1 DFD specifications

In order to limit the complexity of Ray, we made some decisions when it came to the DFD rules, some more demanding than others. These decisions will be discussed further in Section 3.4. The rules are:

- A DFD has to consist of at least one external entity and one processes.
- Only one data type and data value per flow is permitted.
  - Processes and databases are only allowed to output one data type and data value per traverse.
- A traverse has to start in an external entity and end in one or more external entity or database.
- A traverse can only pass though each activator once.
- The process that gets queries a database does not necessarily provide the input for that query.
- Flow IDs have to be unique and use *snake_case* as their naming scheme
- All names for activators and flows must be unique, even across multiple DFDs.
- All flows within a traverse are run sequentially.
- All traverses have to be continuously connected.
- Query definitions only receive inputs from where they are defined, and they do not have any database input.

3.2.2 Definitions for annotations

This subsection will go through the annotations we provide to generate code.

- `@DFD`: It can be used multiple times throughout the project but each DFD has to have a unique name. Used to generate the base activators. Definition seen in the Appendix Figure A.1.
- `@Activator`: The developer annotates classes that inherit any requirement with this annotation, which lets the preprocessing pick up the annotated class to generate code properly. Definition seen in the Appendix Figure A.2.
- `@Traverse`: Lets the developer specify the execution order for a given set of flows. Definition seen in the Appendix Figure A.3.
- `@FlowThrough`: Customize the types and names for a given method inside a process and its potential queries. Definition seen in the Appendix Figure A.6.
- `@Output`: Used in `@Traverse` and `@FlowThrough` to specify the data type for each flow within the DFD. It also has a field called `collection`, which lets the developer specify if the flow should be a collection of the type or a single value. Definition seen in the Appendix Figure A.7.
- `@Query`: Used to specify what data type a query result has and what database the data comes from. Definition seen in the Appendix Figure A.6.

- `@QueriesFor`: This annotation is used when the process that receives data for a query and the process that receives data from a query are not the same. Example of its usage can be seen in sub-section 3.3.3. Definition seen in the Appendix Figure A.4.
- `@QueryDefinition`: Replaces the query definition in `@FlowTrough` between one database and another process. Example of its usage can be seen in sub-section 3.3.4. Definition seen in the Appendix in Figure A.5.

3.2.3 Sub-algorithms

There are three different stages that a developer can find themselves in.

1. Adding `@DFD`, along with a recompilation, will generate a requirements class for each activator in the given DFD.
2. The developer then has to create classes for each activator, annotating them with `@Activator` and inheriting the corresponding abstraction. To generate the methods for the activators, the developer has to specify the traverses and their flow order with the `@Traverse` annotation. Those annotations go on top of a relevant external entity that starts the given traverse. After recompilation, the methods get added to the requirements abstractions.
3. The last stage is when the developer can start to specify the data types for the flows by annotating processes with `@FlowThrough` and adding `startTypes` to `@Traverse`. After recompilation, the types will change from `Object` to the specified types.

Ray has five different sub-algorithms that will run from scratch every time after recompilation if there is any change to the developer's source code. However, if there is no change in either a DFD or any annotation, the generated classes would stay the same since the algorithm is deterministic. Ray's domain increases in detail during stages one to four until stage five, where the code is generated using Ray's domain. The sub-algorithms are the following:

1. Classes annotated with `@DFD` and `@Activator` are used to help set up the domain. `@DFD` links to an XML representation of a DFD that is first read. Then, for each activator in the DFD, an aggregate is instantiated that represents the activator. An activator aggregate consists of, among others things, what Java class it is connected to and what activator it represents. The aggregates also contain activator-specific data, such as all flows that pass through a process activator.
2. In the second round of searching, classes annotated with `@Traverse` and `@FlowThrough` are used to enhance the domain further. If a traverse has a flow order specified, we can start connecting the aggregates. With the types from both `@Traverse` and `@FlowThrough`, we also add types to the aggregates. After this, the aggregates are not enhanced further.
3. For each activator aggregate, we generate the requirements Java code. The barebone class gets made if the developer is only at stage 1. If, however, the developer is at stage 2 or 3, methods get added to the class.
4. If the developer is at stage 2 or 3, we can generate the traverses inside the

external entities' abstraction for each specified traverse. Note that there are no checks here; Ray trusts the input completely.
5. After generation, we take the code and create actual Java files.

After sub-algorithm five, the generated Java files get compiled with the possibility of compile errors.

3.2.4 Representation of domain

Before we start to read the metadata from annotations such as `@FlowThrough`, we convert the graph structure we receive from a DFD to what we call aggregates. There are three different aggregate types for each type of activator: `ProcessActivatorAggregate`, `DatabaseActivatorAggregate`, and `ExternalEntityActivatorAggregate`. These aggregates, along with the traverses, are the domain that we modify until we send it in its entirety to the Java file generator.

These aggregates get connected through `Connector`s, which is what we use when setting up the traverses. For example, if we want to connect a flow from a processor to a database, we will create a `Connector` that represents the output for the processor for the given traverse, and then use the same instance of `Connector` as the input for the database store operation. The great thing we get out of using the same instance is that when we receive information from `@FlowThrough` that this processor should change its output, then the database will have the same information that one of their inputs will be this new type.

These connectors are not saved directly upon the aggregates but instead on `FlowThroughAggregate`. For each traverse that goes through a `ProcessActivatorAggregate` we have one `FlowThroughAggregate` that has the output `Connector` and the input `Connector`s. This `FlowThroughAggregate` also contains information such as the function name.

`FlowThroughAggregate` also contains the information needed for queries. Queries are stored differently from `Connectors` because we want the possibility to split up the definition of a query and the actual data that comes in to a method. Ray lets the developer move the query definition, which is the lambda where the user specifies what data to query, because sometimes it makes more sense to move it, as described in sub-section 3.3.4. One strength of moving the query definition is the possibility to use the inputs for that processor instead of the inputs that were going to go to the activator that will receive the query results. Section 3.3.4 has an example of how this works in practice. Something vital with query definitions is that they must not have any side-effect, such as saving something inside the definition.

This domain representation is then what gets used when generating the code. We generate the code per activator aggregate, and all possible outside information will get retrieved by `FlowThroughAggregate` and things such as their `Connectors`. The traverse runtime generated for each traverse in the abstract external entity goes

through the specified execution order step by step. The execution order is the order in which a method inside the different activators should execute.

To know what input goes where, we create a map that maps from the instance of a `Connector` to a variable name. We then go through the execution order, and since we only have one output per activator, we save the return value of the activator to that variable. If that `Connector` is later used by the database to store something, the `Connector` will be an input for that database, and we can then retrieve the variable name that has the value. Suppose the developer specifies the traverse order incorrectly, then there will be a compilation error since an activator wants the value from a connector whose variable has not been defined yet.

The execution order gets calculated from the flow order specified by the developer. For each specified flow, we take the *from* activator and add that to the order of execution if it is activator is a process. If the *to* activator is something where the traverse can end, such as a database or external entity, we add that to the execution order. If, however, that database or external entity has already been added, we move the first instance to the back of the execution order since we need to make sure that all the activators that that output activator needs has been able to execute. We then end up with a list of activators for how a traverse should execute which Figure 3.8 shows.

3.3 Other functionalities

This section gives further examples of functionalities that Section 3.1 did not cover.

3.3.1 Changing function name

The name that gets used for all requirements files will by default be the traverse name, but this can make the developer-created classes hard to read. `@FlowThrough` gives the developer the option to change the name for the given process activator with the `functionName` field. Figure 3.17 shows a method name for the traverse `M` which changed from `M` to `marketing`. The name change does not affect anything outside of this method. The traverse generation will adapt and use the new function name `marketing` instead of `M`.

Changing the function's name can help the developer create process activators that are easy to read without having to know about Ray. Without the ability to change the function name, the developer would get forced to use the non-descriptive traverse names that do not help with explaining what the function does. This functionality contributes to Ray's flexibility.

```
@FlowThrough(
        traverse = "M",
        output = @Output(type = EmailAndContent.class),
        functionName = "marketing"
)
@Activator
public class MarketingBlast implements MarketingBlastRequirements {
    @Override
    public EmailAndContent marketing(...) {
        /* return ... */
    }
}
```

Figure 3.17: Example of changing function name for a process.

3.3.2 Reflection

The starting external entity for each traverse is responsible for providing references for all activators along the traverses. To help the developer with some of these, we added the option to instantiate the activators using Java Reflection. By using reflection, the developer does not need to worry about instantiating all the activators but, in turn, loses the flexibility of the instantiation. The developer can use this option by specifying `instantiateWithReflection = true` in `@Activator`. Figure 3.18 shows an example of this.

3.3.3 @QueriesFor

When designing a system, it might not necessarily be that the process that needs data from a query is the one providing the query with the input data.

For this scenario, we created the `@QuerierFor` annotation. Figure 3.19 shows an example of how they get used. This means that we can hide the ability to save data directly to UserDB inside the query definition by accident. The developer only has access to methods used for querying.

A thing to remember is that UserDB in this scenario will now have to implement another method called `getQuerierInstance()` that is used to get a new instance for `UserDBQuerier`.

3.3.4 @QueryDefintion

By default, the query definition gets placed on the processor that will receive the data, but sometimes it might make more sense to move this definition. For example, if there is a system used to send the latest sale information to a user's preferred way of contact. Figure 3.20a shows how one could model that DFD.

```java
@Activator(instantiateWithReflection = true)
public class MarketingBlast implements MarketingBlastRequirements {
    /* ... */
}
```

```java
@Generated("holt.JavaFileGenerator")
public abstract class AbstractCompany {
  /* ... */
  public AbstractCompany() {
      this.userDBRef = reflect(UserDB.class);
      this.marketingBlastRef = reflect(MarketingBlast.class);
      this.mailSenderRef = reflect(MailSender.class);
  }

  private <T> T reflect(Class<?> t) {
      try {
        return (T) t.getDeclaredConstructor().newInstance();
      } catch (Exception e) {
        e.printStackTrace();
        throw new IllegalStateException("Could not
        ↪  instantiate...");
      }
  }
  /* ... */
}
```

Figure 3.18: Usage of `instantiateWithReflection`.

The `PreferredContactSelector` process checks the user object and sends forward the preferred way of contact. It is more interesting for `MessageConstructor` to have the query definition since they are the ones that are going to use the contact information. Figure 3.20b shows how the `@QueryDefinition` gets used by `MessageConstructor` and how they now have the query definition in their class. Note that the inputs for `MessageConstructor` will now get used instead of the input that `PreferredContactSelector` would have recieved (which would be none in this example).

3.3.5 Modifying the DFD

Software projects are constantly changing, and therefore Ray needs to handle that. When changing the DFD, the designer can either add or remove activators, and this section describes how Ray handles that and what the developer needs to do.

When the developer starts with the DFD seen in Figure 3.21, a system for com-

```java
/* ... */
public class EmailCreator implements EmailCreatorRequirements {
    /* ... */
    @Override
    public UserDBToEmailCreatorCreateQuery
     ↪  queryUserDBCreate(EmailContent input1) {
        return UserDBQuerier::getAUser;
    }
}
```

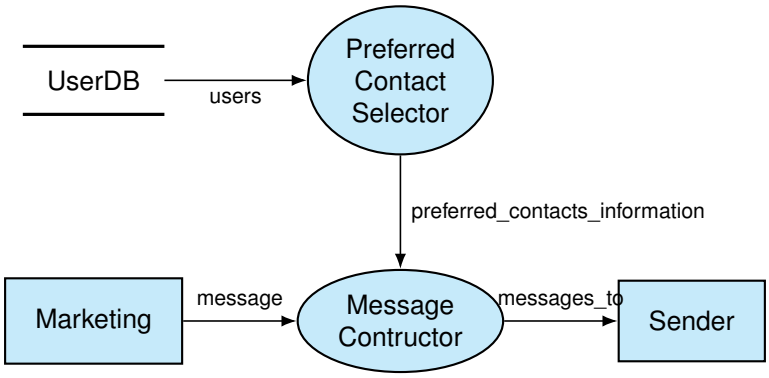
```java
@QueriesFor(UserDB.class)
public class UserDBQuerier {
    /* ... */
    public User getAUser() {
        return userDB.getAUser();
    }
}
```

Figure 3.19: Usage of `@QueryDefinition`.

paring different packing methods, they would also create something similar to what Figure 3.22.

If the developer notices that `StrategyOne` never creates better packing, it is a straightforward process to remove it. After first removing the activator from the XML file, compile errors appear until the developer has removed all code related to that activator and flows that go in or out of it. For example, the `FindBestStrategyRequirements` would change from having two parameters in the `findBestStrategy` method to only having one. Figure 3.23 shows the change from Figure 3.22.

If the developer instead discovers a need for a third packing strategy, the process is similar. The first step is to modify the DFD by adding the new activator and flows, resulting in the DFD seen in Figure 3.24. When adding an activator in the DFD, Ray tells the developer that there are flows in the XML file that are not yet used in the codebase and that the new activator requirements interface has not been implemented. After the developer corrects the order and implements the interface, the code compiles and works as expected. Figure 3.25 shows the updated files. An essential aspect of these processes is that the developer does not need to change more than is necessary, as only the affected parts of the DFD need to be changed in the corresponding code.



(a) DFD for sending messages for `@QueryDefinition` example.

```
@FlowThrough(
    traverse = "M",
    output = @Output(type = MessageWithContactInformation.class,
    ↪  collection = true),
    overrideQueries = {
        @QueryDefinition(
            db = UserDB.class,
            process = PreferredContactSelector.class,
            output = @Output(type = ContactInformation.class,
            ↪  collection = true)
        )
    }
)
@Activator(instantiateWithReflection = true)
public class MessageConstructor implements
↪  MessageConstructorRequirements {
    @Override
    public Collection<MessageWithContactInformation> create(Message
    ↪  message, Collection<ContactInformation> contactInformations)
    ↪  {
        /** return ... */
    }

    @Override
    public UserDBToMessageConstructorMQuery
    ↪  queryUserDBCreate(Message input0) {
        return UserDB::getUsers;
    }
}
```

(b) `MessageConstructor` that uses `@QueryDefinition`

Figure 3.20: Usage of `@QueryDefinition`.

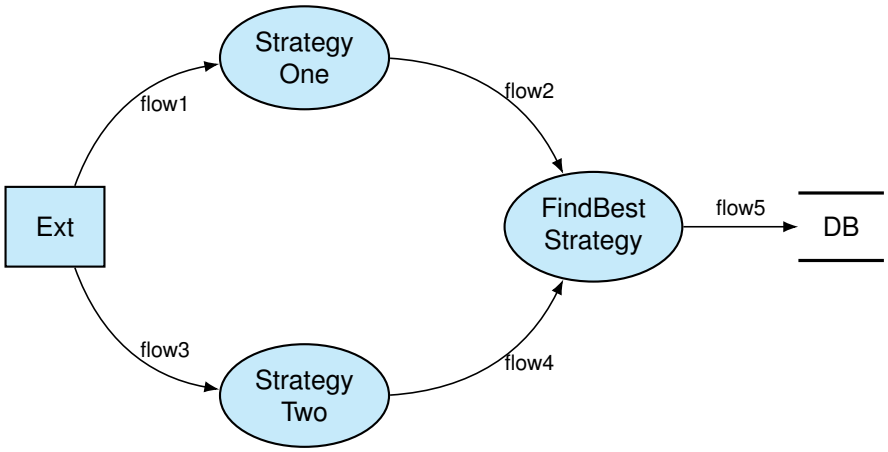


Figure 3.21: Starting DFD.

```
@Traverse(
        name = "P",
        order = {"flow1", "flow2", "flow3", "flow4", "flow5"},
        startTypes = {@Output(
                type = Item.class,
                collection = true)}
)
@Activator
public class Ext extends AbstractExt {
    public void package(Collection<Item> items){
        super.package(items);
    }
}
```

```
/* FindBestStrategy.java */
public Container findBestStrategy(Container input0, Container
 ↪  input1) {
    /* ... */
}
```

Figure 3.22: Examples of `Ext` and `FindBestStrategy` with input type specified in `StrategyOne` and `StrategyTwo`.

3.3.6 Implementing multiple requirements

Sometimes there might be a need to combine multiple activators into a single Java class. If we continue the example of the different packing strategies, it might be helpful to combine them into a single class instead of separating them into two or more classes. To use this feature, the developer needs to implement both of the required interfaces related to the activators that should be combined. In the `@FlowThrough`

```
@Traverse(
        name = "P",
        order = {"flow3", "flow4", "flow5"},
        startTypes = {@Output(
                type = Item.class,
                collection = true)}
)
@Activator
public class Ext extends AbstractExt {
    /* ... */
}
```

(a) `Ext`.

```
/* FindBestStrategy.java */
public Container findBestStrategy(Container strategyTwo) {
    return strategyTwo;
}
```

(b) `FindBestStrategy` after removing `StrategyOne`.

Figure 3.23: Examples of `Ext` and `FindBestStrategy`.

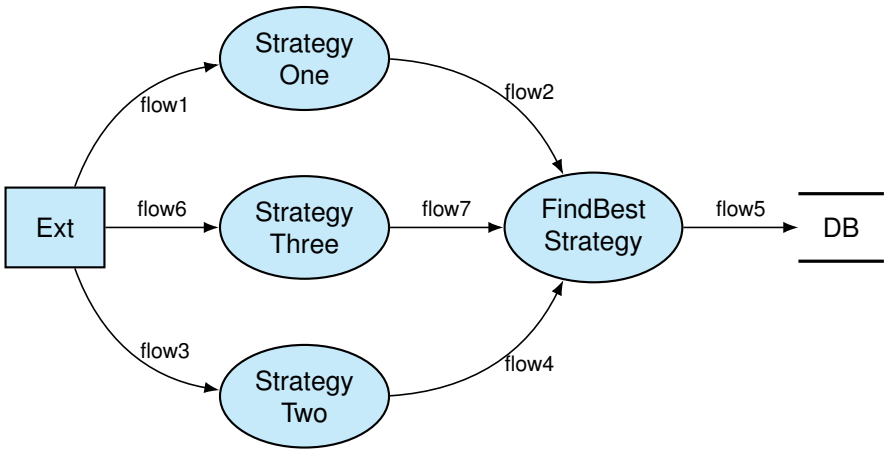


Figure 3.24: Adding StrategyThree.

for this class, the developer also needs to specify for what activator the different `@FlowThrough` applies to via the `forActivator` parameter. Figure 3.26 shows how this scenario could be done.

```
@Traverse(
        name = "P",
        order = {"flow1", "flow2", "flow3", "flow4", "flow6",
        ↪  "flow7", "flow5"},
        startTypes = {
            @Output(
                type = Item.class,
                collection = true
                )
        }
)
@Activator
public class Ext extends AbstractExt {
    public void package(Collection<Item> items){
        super.package(items);
    }
}
```

(a) `Ext`.

```
@Activator(instantiateWithReflection = true)
public class FindBestStrategy implements
↪  FindBestStrategyRequirements {
    @Override
    public Container findBestStrategy(Container input0, Container
    ↪  input1, Container input2) {
        /* ... */
    }
}
```

(b) `FindBestStrategy` after adding a third strategy.

Figure 3.25: Examples of `Ext` and `FindBestStrategy`.

3.3.7 Returning data to the starting external entity

We have only shown examples where we send data from an external entity to databases or other external entities. However, there is also the functionality of returning data to the starting external entity. If this is the case, the generated traverse method inside the abstract external entity would return the specified value instead of nothing. Figure 3.27 shows an example of this where the input is the salary before tax, and it returns a calculated taxed salary after running through a few activators.

```
@FlowThrough(
        traverse = "P",
        output = @Output(type = Container.class),
        functionName = "strategyOne",
        forActivator = "StrategyOne"
)
@FlowThrough(
        traverse = "P",
        output = @Output(type = Container.class),
        functionName = "strategyTwo",
        forActivator = "StrategyTwo"
)
@Activator(instantiateWithReflection = true)
public class Strategies implements StrategyOneRequirements,
 ↪  StrategyTwoRequirements {

    @Override
    public Container strategyOne(Collection<Item> items) {
        /* ... */
    }

    @Override
    public Container strategyTwo(Collection<Item> items) {
        /* ... */
    }
}
```

Figure 3.26: Example of combining multiple activators into a single class.

```
@Generated("holt.JavaFileGenerator")
public abstract class AbstractTaxCalculator {
    /* ... */
    public TaxedSalary CT(Salary salary) {
        /* ... */
        return v5;
    }
    /* ... */
}
```

Figure 3.27: Example from AbstractTaxCalculator where the traverse Calculate Tax (CT) returns the salary after tax.

3.4 Design decisions

We finish this chapter by covering several design aspects regarding DFD code generation and our reasoning surrounding why we think we have designed a good solution.

3.4.1 DFD specifications

We put much thought into our DFD specification to make a good and usable algorithm for the developers.

Section 3.2.1 listed our specifications, the first one being that a DFD has to consist of at least one external entity and one process. The reason for this requirement is that all traverses have to start in an external entity, and for the flow to end, it has to have passed through a process activator.

The second specification states that a data flow can only carry one data type with one data value and that processes and databases only can output one data type and data value per traverse. The reasoning behind this is that we wanted to not increase cohesion and if we allowed more than one output, the developer would need to specify what input each activator needed. Note that the one data value can also be a collection.

Another specification is that the activator that gets data from a database does not necessarily provide the input for that query. It essentially means that any process can start a query flow as long as it can provide the parameters for the query. This specification was essential for the extension to PA-DFD since we wanted to use Ray for the PA-DFD later in the project; explained in Chapter 4. It also increases code cohesion, as described in Section 3.4.3.

The last specification was that query definitions only receive inputs from where they are defined and do not have any database input. This specification decreases coupling within the developer's project, and it also ensures that no processor ever has direct access to the database it queries from.

3.4.2 Annotation processors to generate code

Ray executes during compile-time to help tighten the integration from the DFD to the developer's codebase and make it more resilient to future refactorings. If, for example, a process is added to a DFD, then a new interface with requirements will be generated at recompile. The only other activator that would be affected would be those connected to the new activator and all others would be unaffected. This strategy for code generation is, in our opinion, superior to instead generating everything, exporting it, and then importing it into the developer's codebase for each change made to the DFD or execution order in the traverses, which is what previous work suggested [7].

A mismatch between the generated code from DFD and the developer's codebase would result in a compilation error with our strategy, which is good. Compilation errors require human interaction to resolve them. Often the errors are, at least in our case, straightforward with the issue at hand. They will always happen when the developer adds new metadata to the activator through annotations. For example, the default name for a function is the traverse's name. If the developer instead wants to change the name to something more descriptive, say `createEmailAndContent`, then the activator requirements abstraction will be generated with this metadata, and there will be a compile error for the class that inherits the abstraction. The developer will need to change the name of the method from the traverse's name to the new name of `createEmailAndContent`. Figure 3.28 shows the before, with new annotation value in Figure 3.28b, and after fixing the compilation error in Figure 3.28c.

Another essential factor with preprocessors is that we generate code that has high cohesion with the developer's code, but not the other way around. Suppose the developer does not want to use our code generation anymore. In that case, they can still use the modular processes, databases, and external entities to connect them once again to run their business logic. The developer would have to remove all the implemented requirements interfaces and abstract external entity class. Still, they would not lose any functionality after removing them other than the generated traverses. The requirement interfaces are there to help the project stay in line according to the designed DFD.

3.4.3 Specifying queries in processes instead of database activators

We decided to put the processes' query methods inside the processes to achieve higher code cohesion since each query is only relevant to the traverse it belongs to. If there are two traverses where there is a query from a database to a process, then those two queries are separated and have nothing in common. If those queries existed as a method for the database requirements interface, it would not be as easy to glance over what kind of data a given method inside a process queries when looking at the process activator.

Another route for achieving high cohesion could be to give access to the database specified directly to the method in need, but this would instead lead to high coupling. The whole point of having a DFD to base the design on is to separate different stages of a given traverse, and it would break the single responsibility principle. Thus, having a separate method for each query helps separate the different flows.

However, these queries return a query interface instead of directly returning the result, and the main reason behind this decision is to guide the developer that access to the database is only temporary. We do not want the developer to use the database reference however they like because the same issue described in the last paragraph would return. Nothing stops the user from saving the reference outside,

```java
@Activator
public class MarketingBlast implements MarketingBlastRequirements {
    @Override
    public Object M(Object input0) {
        /* ... */
    }
}
```

(a) Before adding annotation.

```java
@FlowThrough(
    traverse = "M",
    functionName = "createEmailAndContent"
)
@Activator
public class MarketingBlast implements MarketingBlastRequirements {
    @Override
    public Object M(Object input0) {
        /* ... */
    }
}
```

(b) The method name is now set, and the developer have recompiled. The highlighted lines shows the current compilation error since in `MarketingBlastRequirements`, the method name is `createEmailAndContent` and not `M` which it was named previously.

```java
@FlowThrough(
    traverse = "M",
    functionName = "createEmailAndContent"
)
@Activator
public class MarketingBlast implements MarketingBlastRequirements {
    @Override
    public Object createEmailAndContent(Object input0) {
        /* ... */
    }
}
```

(c) No more compilation errors after changing the method name.

Figure 3.28: Example of the process for the developer to change the method name from the default one, which is the name of the traverse, to something custom which in this case is `createEmailAndContent`.

but we are at least not encouraging it by our design.

3.4.4 Option of having multiple DFDs in one project

DFDs and other similar graphical representations of system design tend to get complicated and hard to understand as they grow in size. The strength of DFDs are their simplicity and that they do not have to represent the entire system but rather a slice of the system. Context diagrams are one option to combat the complexity of DFDs, but we chose the route instead of giving the developer the ability to have more than one DFD that generates code. To take advantage of multiple DFDs, the developer needs to use multiple `@DFD` in their project.

However, there are a few things to keep in mind with the opportunity to have multiple DFDs. Names for external entities, processes, and databases defined in the DFDs must be unique to not clash during code generation. All activators will be unique for the given DFD and not shared. Unique activators promote low coupling, primarily if the developer properly designs each DFD as its separate slice. There can be shared business logic that multiple activators use outside the DFDs, but this could come back to haunt the developer if that makes it harder to separate the responsibility of each DFD. Nevertheless, having, for example, multiple database activators does not mean that there exists one actual database per activator but rather that there is a clear separation of responsibilities over the actual database.

3.4.5 Everything compiled before runtime

A significant strength of Ray is that it depends on the developer's codebase more than the other way around. There are annotations that the developer uses to generate the requirements file correctly; however, those annotations can get removed along with the requirements file, and the business logic would still be there. If the developer removes Ray, what would be missing is the traverse runtime that gets generated automatically.

The generated traverse runtime is easy to evaluate for the developer and testing. There are no libraries needed; it is ready to be tested and run after Ray has generated the code. No extra runtime is needed in any matter, which is a great strength for Ray.

Imagine the alternative, if the traverse runtime was set up at the program's start instead. This scenario could lead to unexpected runtime errors such as object casting errors. By generating before runtime, that pitfall is avoided. It is one of the ways that Ray brings stability.

3.4.6 Using Ray as an entry point

Our solution of generating code and forcing the developer to follow the designed DFD line can help create an entry point for different use cases for a given system. For example, given there is an existing system with many services to handle an online clothing company. These services could be:

- `ShippingService`: Handles ordering shipments for postal service.
- `InventoryService`: Keep track of items in inventory.
- `CustomerService`: Manage customer information such as payment information and shipping address.
- `PaymentService`: Contacts external service to make payments by credit card.

To fulfill the requirement of letting Alice order a T-Shirt, we would need to use all services. However, this can be unclear when planning first in terms of layout and design. Here, Ray could be a great addition. Figure 3.29 shows a DFD for our suggestion to solve this problem. The order of the traverse to buy a T-Shirt would be:

1. Send the order request to `InvoiceCreator`.
2. `InvoiceCreator` queries customer information containing Alice's shipping address and payment information.
3. `InvoiceCreator` would also query and reserve the T-Shirt that the request wants to buy and send the cost with it. Note that it is a query with a side effect: the reservation.
4. `InvoiceCreator` would, with the queried data, create an invoice and then send that to:
   (a) `PaymentService` to try to charge Alice with the price for shipping and the T-Shirt. If there is any problem, an unchecked exception would get thrown, which would stop the traverse further execution.
   (b) `ShippingService` would also call an external service to create an order to the shipping department to send the T-Shirt to Alice.

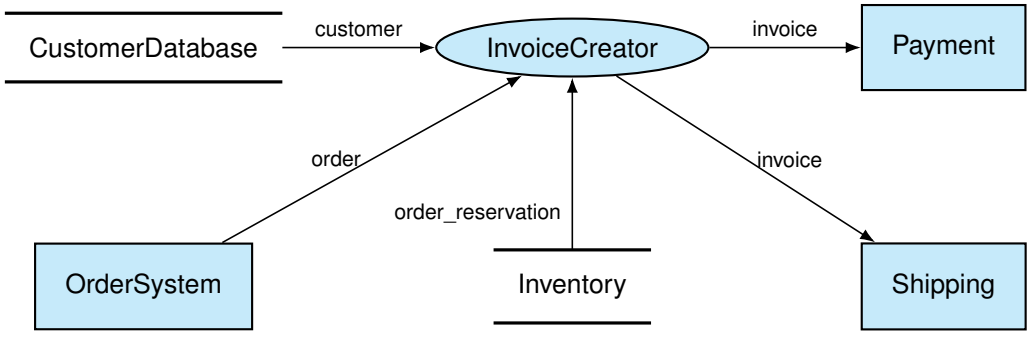


Figure 3.29: DFD that represents a order system.

The developer would then create the activators and call the correct services. Another solution could be to implement the requirements interface directly onto the service classes and then implement the method there, thus turning them into activators.

4

Realizing Privacy-Aware activators and their runtime

This chapter will explain how we realize PA-DFDs into runnable Java code. It will tackle Contributions 2 and 3 specified in Section 1.2, which has to do with providing a workflow for Privacy by Design (PbD) and being able to represent privacy concerns with policies. We start this chapter by introducing the problems we aim to solve and why it is exciting and essential to solving them.

Whenever a system has to process personal data in any matter, a sense of responsibility and duty must get introduced at the same time as designing the system, and PbD is a way of thinking that aims to help developers not forget the integrity and privacy of personal data. PbD does not have any concrete ideas or design patterns that developers can pick up; it gets viewed more as a philosophy where specific questions get asked by the developer from the start of the design until production and beyond.

Antignac et al. aimed to take the design one can do with DFD and introduce checks before each process and accountability for each limit activator [5]. This was to make sure all decisions made were made with intention. They introduced the PA-DFD. It brings PbD to create a workflow where developers are forced to not forget about the critical privacy concerns that need answers.

Today, even without our thesis and its solutions, a developer can use the transformation rules that they introduced and create a system that follows the principles of PbD and be successful. The key to why that is so is that much of Antignac et al.'s paper often decide not to take a stance, if possible, on how policies are structured or precisely what the request and reason activators have for tasks, for example.

However, the problem of trying to realize a PA-DFD directly in the developer's codebase is that there are many steps that can be automated to make development more manageable and faster, along with the many decisions that have to be made, but more importantly, continuously made, to ensure consistency in the developer's codebase. Many decisions that have to be made cannot be taken lightly. Problems such as handling the logging or ensuring that all limit activators are properly implemented but, more importantly, called adequately at the correct time before the next activator.

These extra steps, between a designed PA-DFD and a clear, structured codebase

with runnable code, are where we introduce two names: HPA-DFD and Holt. Note that Holt is just a code word we have chosen to make it easier for the reader to differentiate between PA-DFD and HPA-DFD, for example.

HPA-DFD, which is an acronym for Holt Privacy-Aware Data Flow Diagram, is modifying the definition of PA-DFD to work within the constraints of the designed Ray specified in the previous chapter. In short, we build Holt on top of Ray. Later in this chapter, we will go through each difference between PA-DFD and HPA-DFD, but for now, remember that HPA-DFD is how we can achieve the wanted behaviors of PA-DFD. Do note that HPA-DFD is, at its core, just a DFD, but that has some generated code within itself that Holt generates.

Holt is the algorithm that takes the given DFD from the developer and transforms it into an HPA-DFD by adding new activators, new flows, and moving queries, amongst other things. While Ray on its own works well if the developer wants to generate code from a designed DFD, Holt gets tightly integrated between specific steps of Ray. Holt is highly dependent on Ray, but Ray is only dependent on Holt because it will call Holt when needed.

As previously mentioned, we built our HPA-DFD solution on top of Ray specified in the last chapter, which has one major limitation: there can only be one output per process for a given traverse. Using Ray has its pros and cons, but we believe the pros outweigh the cons, even the previously mentioned limitation of outputs.

The HPA-DFD aims to mimic the PA-DFD as closely as possible in the sense that the developer should feel like they are interacting with a PA-DFD and not the extended HPA-DFD. HPA-DFD has, for example, additional activators that the developer never gets to see but which are essential so that the developer can focus on the continuous work of PbD.

The following section will go through a workflow similar to what got done in Chapter 3. However, this time we will activate the Privacy-Aware functionality and thus activate the usage of Holt.

4.1 Workflow when using our algorithm

This section will work like Section 3.1, but this time with a PA-DFD, to show how a developer goes from a DFD to executable code with privacy-aware functionalities. This example will start from nothing, just as Section 3.1 did. We will only go through one traverse to keep the example simple. The executable code will not be a 1:1 map of the graph; there are some differences that we will go through in greater detail later. We will base this example on the PA-DFD defined in Figure 4.1. There will be more classes to create and more annotations to add, even though we only have one traverse.

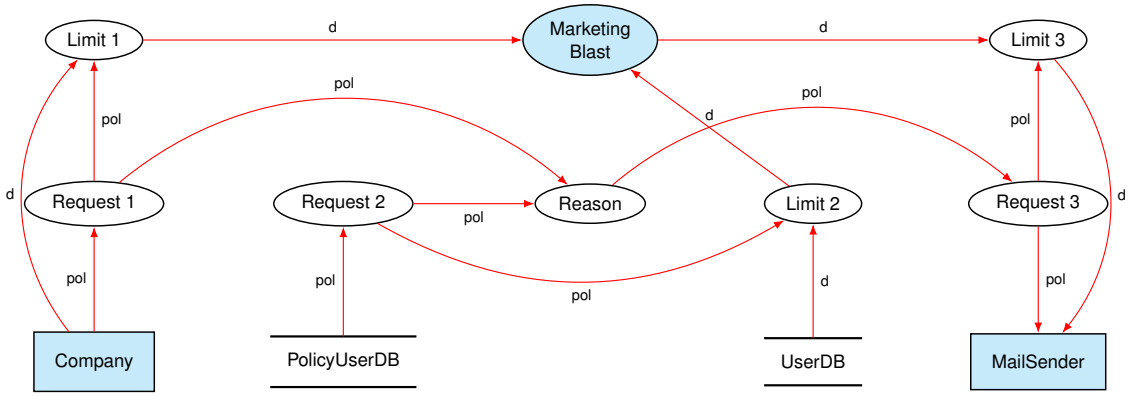


Figure 4.1: PA-DFD with a traverse named *Marketing*, shortened `M` that starts in the `Company` external entity and end in `MailSender`.

It is worth noting that, before getting into our example, how we are implementing policies and privacy is not the rule of law; our algorithm does have an opinion on that. The only opinion is the PA-DFD structure itself, with the custom activators and new flows.

What before was a DFD to send marketing emails to users without any privacy checks is now a PA-DFD where we check if the user has agreed to receive marketing emails before sending them. To represent this, we use the classes defined in Figure 4.2.

```
1  public interface Agreement extends Policy { }
2
3  public record UserPolicy(List<Agreement> agreements) implements
   ↪  Policy { }
4
5  public enum AccessUserReason implements Policy, Agreement {
6      MARKETING, RESET_PASSWORD, DELETE, STORE
7  }
8
9  public enum MarketingType implements Policy {
10     PRODUCT_MARKETING
11 }
12
13 public record ContentAndUserPolicy(MarketingType marketingType,
   ↪  UserPolicy userPolicy) { }
```

Figure 4.2: Records used as policies.

This workflow example will omit log and log databases to keep this example smaller and easier to digest. Figure B.1 in the Appendix shows a PA-DFD including those

activators for two traverses. If we did include them in this example, we would add one log activator next to each limit, along with one log database after each log activator. As previously mentioned, the log activator's goal is to fulfill the accountability requirement by logging the data, the corresponding policy, and the decision from the limit activator.

Before going through the code, we want to take a second to go through the traverse step by step and help the reader visualize how data flows. Inside our UserDB, we have Alice and Bob. Alice has agreed to let the company send marketing to her, while Bob has not. These agreements exist in the PolicyUserDB.

The company now wants to send a marketing blast for their new pair of red shoes, and they are going to use the external entity `Company` to do so. Two outputs from the entity will get sent, one that contains the content of the email and one purpose, which in this case is that we want to send a marketing blast with the type of product.

Request 1 will pass forward the purpose object and does not need to do any further processing.

Limit 1 will make sure that the type of product that will be marketing is of type product since this is the only type of product that Alice and Bob could have agreed on currently. If we were to add a new type, say holiday sale, then that Limit check would stop execution and force the developer to acknowledge this possible issue.

Before running the method in MarketingBlast that generates a ready-to-send email, we need to retrieve Alice' and Bob's emails. Holt has moved the query definition from Limit 2 to Marketing Blast since it is where the result will get used. However, the emails still need to pass through the Limit 2 check for MarketingBlast to receive the emails.

After the query has run, another query gets done from the policy database, which will retrieve the policies related to the emails retrieved from UserDB. Request 2 will construct a map that links a given email to the policy that represents it.

With the policy map constructed in Request 2, Limit 2 can then check for each email retrieved from the first query if they can get used as a recipient of a marketing blast, which is for new red shoes. Alice has approved this, while Bob has not, so Alice's email will get sent to the MarketingBlast process, and Bob's email will not go any further.

The method for generating the emails can now run, and MarketingBlast will receive the email content from Limit 2 and Alice's email address from the user database. It generates the ready-to-send email and sends it down the line to MailSender.

Limit 3, Reason, and Request 2 will not do anything special now that we have a ready-to-send email that has the approval of Alice. If we wanted to design things

differently, we could have split the policy into asking Alice and Bob if they approve the construction of any marketing email using their emails and then have another field where they have to approve the specific kind of marketing email. The second check could be something that Limit 3 can implement. Nevertheless, for now, we check both those things with Limit 2.

The marketing email for the new red shoes with the address to Alice now reaches MailSender and can get sent via an email server outside of this traverse.

With that, we can continue with going through how we can code this logic. As we did in Section 3.1, we will add the annotation `@DFD` to a given class to generate all the requirements, but this time we also add the option `privacyAware = true` to generate the privacy-aware requirements. These new privacy-aware requirements are all interfaces since Limit, Reason, and Request are processes, and policy database is a database. Ray does not differentiate between a DFD and a PA-DFD. PA-DFD, in this sense, is just a modification of the original DFD shown in Figure 3.1. The behind-the-scenes modifications will be shown in depth later in this chapter, but for now, imagine Figure 4.1 as a DFD. Usage of `@DFD` is shown in Figure 4.3. The same four requirements as from Section 3.1 get generated: `AbstractCompany`, `MarketingBlastRequirements`, `UserRBRequirements`, and `AbstractMailSender`.

```
1 @DFD(name = "mail", xml = "mail-dfd.xml", privacyAware = true)
2 public class Main {
3     public static void main(String[] args) {}
4 }
```

Figure 4.3: `@DFD` with `privacyAware = true`.

Next, we will create `Company` to extend the generated class `AbstractCompany`, so we can annotate it with `@Traverse`. Figure 4.4 shows `Company` after this has happened. Some classes generated after recompilation are:

- `CompanyToMarketingBlastMLimitRequirements`
- `CompanyToMarketingBlastMRequestRequirements`
- `UserDBPolicyRequirements`
- `UserDBToMarketingBlastMLimitRequirements`
- `UserDBToMarketingBlastMRequestRequirements`
- `MarketingBlastReasonRequirements`
- `MarketingBlastToMailSenderMLimitRequirements`
- `MarketingBlastToMailSenderMRequestRequirements`

The list above covers each new privacy-aware process or database from Figure 4.1 and is unique to each traverse. If there were, for example, another traverse that contained a flow between `Company` and `MarketingBlast`, then there would be another Limit and Request activator generated between them.

```java
@Traverse(
    name = "M",
    order = {"marketing_content", "users_marketing",
    ↪  "marketing_and_users"
    },
    // MarketingType represents what type of marketing is planned
    ↪  to be sent out.
    startTypes = {@Output(type = EmailContent.class), @Output(type =
    ↪  MarketingType.class)}
)
@Activator(instantiateWithReflection = true)
public class Company extends AbstractCompany {
    public Company(UserPolicyDB userPolicyDB, UserDB userDB,
    ↪  MailSender mailSender) {
        super(userPolicyDB, userDB, mailSender);
    }
}
```

Figure 4.4: Updated `Company` when using PA-DFD.

The three following sub-sections will define the code for each of the flows from the original DFD. After them, a sub-section will summarize the workflow.

4.1.1 User to UserFormatter

The first external entity, where the traverse starts, is a bit special in that there can be more than one output which `@Traverse` and `startOutputs` field define. For PA-DFDs, we can use this functionality to set just two start outputs: one for data and one for policy. For our example, the data will be of type `EmailContent` and the policy will be of type `MarketingType`. Our algorithm needs to be able to differentiate these two, and it does it by requiring that `MarketingType` implements `Policy`. With that information, only the data goes to `Limit`, and `Request` only receives the policy.

Figure 4.5 shows how we only forward the policy received from the entity, which can feel pointless, but sometimes there is nothing for the request activator to do.

By the same logic as Figure 4.5, Figure 4.6 sets up a predicate to check if we can continue with the traverse. The check returns true if the given `MarketingType` is of type `PRODUCT_MARKETING`, and will stop the flow for any other type of marketing. The next sub-section will go through the limit activator more thoroughly.

```java
@FlowThrough(
    traverse = "M",
    output = @Output(type = MarketingType.class)
)
@Activator(instantiateWithReflection = true)
public static class CompanyToMarketingBlastMRequest implements
    CompanyToMarketingBlastMRequestRequirements {
    @Override
    public Map<EmailContent, MarketingType> M(MarketingType input0)
        {
        return Map.of(null, input0);
    }
}
```

Figure 4.5: `CompanyToMarketingBlastMRequest`.

```java
@Activator(instantiateWithReflection = true)
public static class CompanyToMarketingBlastMLimit implements
    CompanyToMarketingBlastMLimitRequirements {
    @Override
    public Predicate<EmailContent> M(Map<EmailContent,
        MarketingType> input0) {
        return emailContent ->
            input0.get(null).equals(MarketingType.PRODUCT_MARKETING);
    }
}
```

Figure 4.6: `CompanyToMarketingBlastMLimit`.

4.1.2 UserDB to MarketingBlast

The developer also has to specify another query definition to get the relevant policies of the given user, and they can do it with the result from their query from the regular database. Figure 4.7 demonstrates how this new query definition is done in the request activator and how that result is then used in the function M. The M function needs to format the policy as a map is order to be able to connect the user to its policy.

The limit activator now has to make sure that the previously specified rules are followed to protect the privacy of the queried user. Figure 4.8 shows the limit activator and the definition of the predicate lambda, which checks that the user has accepted the relevant user agreement. Please note that the limit activator does not receive any data but just the policy map from the request activator; they need to handle the decision given any data. This means that the logic to handle this could be reused elsewhere or even kept in one place to quickly have an overview of the

```
@FlowThrough(
            traverse = "M",
            output = @Output(type = UserPolicy.class),
            functionName = "M",
            queries = {
                    @Query(
                            output = @Output(type =
                            ↪  UserPolicy.class, collection =
                            ↪  true),
                            db = UserPolicyDB.class
                    )
            }
    )
@Activator(instantiateWithReflection = true)
public static class UserDBToMarketingBlastMRequest implements
↪  UserDBToMarketingBlastMRequestRequirements {
    @Override
    public Map<User, UserPolicy> M(Collection<User> input0,
    ↪  Collection<UserPolicy> dbInput1) {
        Map<User, UserPolicy> result = new HashMap<>();
        Iterator<User> iterator0 = input0.iterator();
        Iterator<UserPolicy> iterator1 = dbInput1.iterator();
        while (iterator0.hasNext() && iterator1.hasNext()) {
            result.put(iterator0.next(), iterator1.next());
        }
        return result;
    }

    @Override
    public UserPolicyDBToUserDBToMarketingBlastMRequestMQuery
    ↪  queryUserPolicyDBM(Collection<User> input0) {
        return db -> db.getPolicies(input0);
    }
}
```

Figure 4.7: `UserDBToMarketingBlastMRequest`.

different limit checks used in the system.

4.1.3 MarketingBlast to MailSender

Now that we have the data from the database, we can combine the data into something that the external entity wants. Figure 3.15, from Chapter 3, shows the combination of email and email content in action in the MarketingBlast class.

```
@Activator(instantiateWithReflection = true)
public static class UserDBToMarketingBlastMLimit implements
↪ UserDBToMarketingBlastMLimitRequirements {
    @Override
    public Predicate<User> M(Map<User, UserPolicy> input0) {
        return user -> input0.get(user).agreements()
            .contains(AccessUserReason.MARKETING);
    }
}
```

Figure 4.8: `UserDBToMarketingBlastMLimit`.

The reason activator will now receive policy data from the two request activators. In this case, we want to forward the combination of what type of marketing it is along with the user's policies since that is what is related to the data returned from `MarketingBlast`. Figure 4.9 displays the reason class we use. Note that if more flows were going through the MarketingBlast class, like other types of marketing, there would be a method added for each traverse in the reason activator.

```
@FlowThrough(
        traverse = "M",
        output = @Output(type = ContentAndUserPolicy.class,
        ↪ collection = true),
        functionName = "reason"
)
@Activator(instantiateWithReflection = true)
public static class MarketingBlastReason implements
↪ MarketingBlastReasonRequirements {
    @Override
    public Collection<ContentAndUserPolicy> reason(Map<EmailContent,
    ↪ MarketingType> input0, Map<User, UserPolicy> input1,
    ↪ Collection<EmailAndContent> input2) {
        /* ... */
    }
}
```

Figure 4.9: `MarketingBlastReason`.

Figure 4.10 shows how we sometimes just need to return an empty map since all checks have already been performed.

The limit check between the `MarketingBlast` and `MailSender` only needs to return true because, as mentioned, all checks have already been performed.

```
@FlowThrough(
        traverse = "M",
        output = @Output(type = ContentAndUserPolicy.class,
        ↪  collection = true)
)
@Activator(instantiateWithReflection = true)
public static class MarketingBlastToMailSenderMRequest implements
 ↪  MarketingBlastToMailSenderMRequestRequirements {
    @Override
    public Map<EmailAndContent, ContentAndUserPolicy>
     ↪  M(Collection<ContentAndUserPolicy> input0) {
        return new HashMap<>();
    }
}
```

Figure 4.10: `MarketingBlastToMailSenderMRequest`.

4.1.4 Summary

This relatively simple but complete example showed how the developer's project differs, but this should not be perceived as the only way to handle policies. The rest of this chapter will go through how everything works and shows different examples of using Holt.

There are a few differences that the developer has to take into account when switching their project from DFD to PA-DFD. The significant difference is the generation of more interfaces that the developer must implement with new classes. These include the limit, request, reason, log database, and policy database activators. It can become quite the hassle with all the interfaces, however Holt will generate the log activator. More on this in sub-section 4.2.4.

It is also important that the classes that represent policies need to implement the `Policy` interface, with the reason being able for us to separate the different potential flow output types.

4.2 HPA-DFD

When designing how we were to realize our PA-DFD, we took advantage of the fact that Ray could represent a clear domain before any code got generated. Ray first reads from the XML file representing the DFD to a domain, then applied the metadata through annotations to that domain to later separately generate code from that same domain.

After loading the DFD, we transform it into an HPA-DFD, which has the same

properties as the PA-DFD. HPA-DFD is our implementation for supporting PA-DFD with Ray. This section will discuss the difference between PA-DFD and HPA-DFD and why we needed to extend the definition. However, note that what the developer interacts with behaves like the original proposed PA-DFD, except that the log activator will get generated for the developer.

4.2.1 Guard activator

The original definition of the PA-DFD said that the limit activator should return the data, the policy and the result of the decision, but as we are doing this in Java, we can only return one value. Figure 4.11 illustrates how the original definition wants to return multiple values to both the log activator and the next activator, which in this case is a process that receives these values.

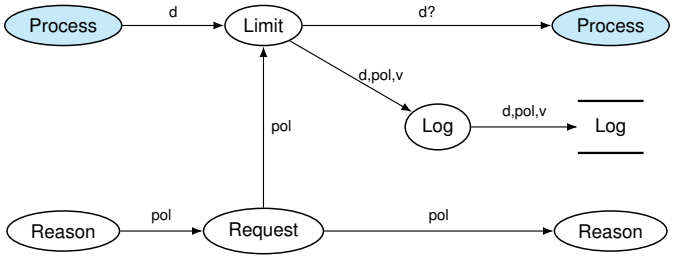


Figure 4.11: Example of how limit activators interact with a PA-DFD.

To resolve the issue of only having one return type, we introduce a new activator and move around some flows, which Figure 4.12 illustrates. It introduces the new activator *Guard* and removes the previous activator's input to limit to instead request that a predicate lambda gets constructed. Figure 4.13 shows an example of a given Note if it is allowed to train an AI.

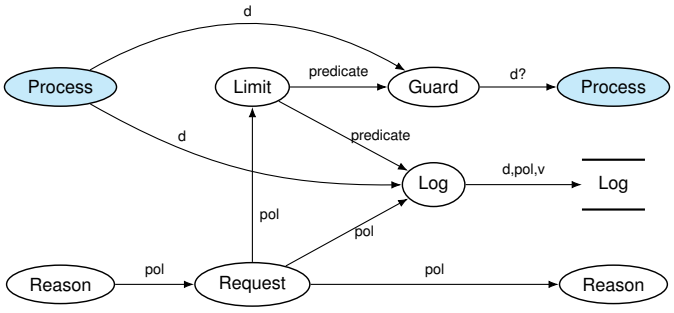


Figure 4.12: How the guard activator fits into the new HPA-DFD.

There is also the problem with us wanting to handle things as a collection, which gets solved by us demanding a predicate from the developer through the limit activator. The developer can handle each item as their own and use the map from the request activator to retrieve relevant policies for that given data.

```java
@Activator(instantiateWithReflection = true)
public class NotesDatabaseToWordCounterCWLimit implements
 ↪  NotesDatabaseToWordCounterCWLimitRequirements {
    @Override
    public Predicate<Note> CW(Map<Note, NotePolicy> notePolicy) {
        return note -> notePolicy.get(note)
            .policies()
            .contains("allowed-ai-training");
    }
}
```

Figure 4.13: Database to Process Limit implementation example using a Predicate.

The guard activator takes in the predicate function from the developer-defined limit activator, goes through all data sent to it, and stops data where the predicate returns false to continue down the line; this helps us handle single items or collections. Figure 4.14 shows how a single data item gets checked, and Figure 4.15 shows instead what happens if there is a collection. The difference is that Figure 4.14 throws an exception if a single item does not approve, while Figure 4.15 filters away the unacceptable data.

```java
public NoteInsertion AN(Predicate<NoteInsertion> tester,
 ↪  NoteInsertion data) {
    if (tester.test(data)) {
        return data;
    } else {
        throw new IllegalStateException();
    }
}
```

Figure 4.14: Part of generated guard example.

```java
public Collection<Note> CW(Predicate<Note> tester, Collection<Note>
 ↪  data) {
    return data.stream().filter(tester::test).toList();
}
```

Figure 4.15: Part of generated guard example with collection.

4.2.2 Combiner activator

Because of the limitation of Java and only being able to return one object, we needed to implement an activator that would combine two or more inputs into one class. This scenario can only occur when a traverse has a flow into an external entity from which it originated. Figure 4.16 shows an example of a data and policy object getting merged into an immutable class called Combo, and Figure 4.17 displays how Combiner would fit into the extended graph.

```
@Generated("holt.PrivacyActivatorJavaFileGenerator")
public final class StatsCWCombiner implements
 ↪  StatsCWCombinerRequirements {
    @Override
    public Combo CW(Integer input0, Map<Integer, CountedWordsPolicy>
     ↪  input1) {
        return new Combo(input0,input1);
    }

    @Generated("holt.PrivacyActivatorJavaFileGenerator")
    public static final class Combo {
        public final Integer v0;

        public final Map<Integer, CountedWordsPolicy> v1;

        Combo(Integer v0, Map<Integer, CountedWordsPolicy> v1) {
            this.v0 = v0;
            this.v1 = v1;
        }

        @Override
        public String toString() {
            return "v0: " + v0 + ", " + "v1: " + v1;
        }
    }
}
```

Figure 4.16: Generated Combiner example.

4.2.3 Querier activator

Figure 4.18 shows how the queries for a normal database and from a policy database, which at a glance looks fine, but when trying to implement, one notices no connection between the data from the normal database and the policies queried. With the same input as the data query, one could probably retrieve the wanted policies. However, we reasoned that to ensure that all needed policies can be retrieved, we

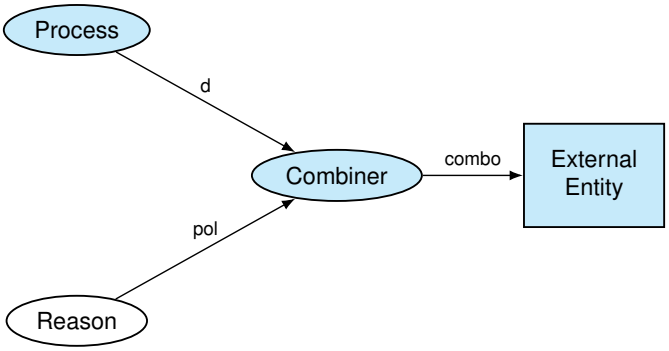


Figure 4.17: How Combiner fits into the new HPA-DFD.

need to use the query result from the normal database when querying for policies in the policy database.

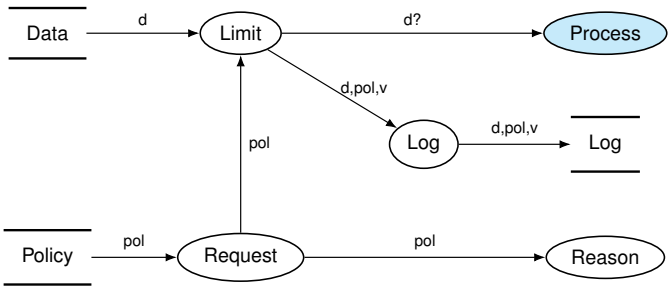


Figure 4.18: How queries are done in PA-DFD.

To resolve the issue of not having access to the query result when querying policies, we add a new activator, called *Querier*, that receives the query data and sends it forward to the request activator to query policies with the normal database query result. Figure 4.19 shows what this looks like in practice behind the scenes from the developer; note that the query definition for the normal database query gets defined in the process activator, and the query definition for the policy database is still in the request activator. Since the querier output is just a normal output from an activator and not a query result, the query definition for the policy database has the result from the normal database as one of its inputs. Figure 4.20 demonstrates how simple the generated code for the new Querier activator is.

The Querier also fits very well with the previous solution to having different outputs from the limit activator. Comparing Figure 4.20 and 4.12 shows the similarities between the Querier and Process in the aspect of sending the data to multiple activators.

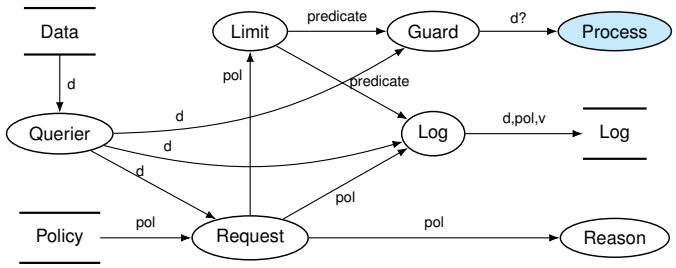


Figure 4.19: How Querier fits into the new HPA-DFD. Note that we are using the guard activator that was previously explained.

```
@Generated("holt.PrivacyActivatorJavaFileGenerator")
public final class NotesDatabaseCWQuerier implements
 ↪  NotesDatabaseCWQuerierRequirements {
  @Override
  public Collection<Note> CW(Collection<Note> queryResult) {
    return queryResult;
  }
}
```

Figure 4.20: Generated Querier example where it just forwards the result. The query definition is defined in the process that is going to receive the data from the database in the end.

4.2.4 Generating the log activator

Because of the changes done with the limit activator, we found that writing the code for the log activator was cumbersome, which led us to generate code for it. Since the activator receives a predicate from the limit activator, we need to call the predicate again to know what the result was. Figure 4.21 shows how this works. Note here that we use `.get(null)` to get the one policy since there is just one data item. However, we need to support the fact that there could be a collection coming in. Figure 4.22 displays how we go through each item in the data collection and use the predicate once more as we are doing in the guard activator, which is fine since the predicate must be deterministic.

Since we are generating the log activator, we added a timestamp to help the developer to support the accountability factor. The log activator uses the Java Instant implementation to do so.

The log database activator is not generated for the developers since they still have to ensure that it is adequately logged to follow the accountability principle again. However, the Row class that we generate inside the log activator does have a proper `toString()` implementation to make it easier to print out the result.

```java
@Generated("holt.PrivacyActivatorJavaFileGenerator")
public final class NoteFormatterToNotesDatabaseANLimitLog
    implements NoteFormatterToNotesDatabaseANLimitLogRequirements {
    @Override
    public Row AN(Predicate<NoteInsertion> tester,
        Map<NoteInsertion, NotePolicy> policyMap,
        NoteInsertion data) {
        return new Row(data, policyMap.get(null),
            tester.test(data), Instant.now());
    }

    @Generated("holt.PrivacyActivatorJavaFileGenerator")
    public static final class Row {
        public final NoteInsertion data;
        public final NotePolicy policy;
        public final Boolean result;
        public final Instant time;

        Row(NoteInsertion data, NotePolicy policy, Boolean result,
            Instant time) {
            this.data = data;
            this.policy = policy;
            this.result = result;
            this.time = time;
        }

        @Override
        public String toString() {
            return "data: " + data + ", " + "policy: " + policy + ",
                " + "result: " + result + ", " + "time: " + time;
        }
    }
}
```

Figure 4.21: Generated log example.

4.2.5 Adding a flow between process activator and its reason activator

To help keep the connection between data and its policy, we want to have the option to receive the data information from the process activator to the reason activator. Figure 4.23 shows the simple change that gets made for HPA-DFD. This scenario might throw some developers off expecting the DFD to look like a PA-DFD. However, in scenarios where the reason otherwise only would have received policies but no way to connect that to any data, then the developer would understand why this flow got

```java
@Override
public Collection<Row> CW(Predicate<Note> tester, Map<Note,
    NotePolicy> policyMap,
    Collection<Note> data) {
        return data
            .stream()
            .map(d -> new Row(d, policyMap.get(d), tester.test(d),
                Instant.now()))
            .toList();
}
```

Figure 4.22: Generated log with collection example. Note that we are only highlighting the main differences from this and Figure 4.21.

added.

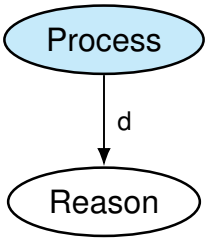


Figure 4.23: Flow between process and reason in HPA-DFD.

4.2.6 Moving the query definition

When the developer decides to migrate from DFD to PA-DFD, then things will, as previously said, expand and change. Something that gets changed back, so to say, is that the query definitions get moved back to where they would be if Holt were not used, from the Querier activator to the process activator. For example, there is a querier and guard activator before the data reaches the intended process activator. Since the query definition, by default, is set up for the process that is going to receive the query result, we lose that query definition that was in the process before activating privacy-aware mode. This is something we move back to, thanks to Ray's flexibility. We move the query definition back to the process, but the data will still go through the querier and guard.

4.3 Clean activator

Figure 2.3h states that a clean activator must be added in the transformation of a DFD to a PA-DFD if there is a store flow from a process to a database; this gets ignored in HPA-DFD. We believe that the clean activator only exists to ensure that

the developer does not forget to regularly delete data that no longer has a purpose to persist.

The position of the clean activator is special since it is not connected to any external entity from where the potential clean traverse can start. Chapter 3 defined traverses to help build a runtime for code execution through a DFD, and one requirement was that each traverse has to begin with an external entity, which is something that the clean activator right now does not. How the clean activator should get integrated into a system is not clear when looking at previous papers, but we believe there can be two possibilities as described below.

The first possibility, the clean activator should be included in any traverse that has to do with the store operation. When something gets attempted to be stored, the clean activator springs into action to check if any other data rows in the database should get removed. Issues arise if the traverse with the store operation does not often get used. If that were the case, then data with no purpose would be stored for a way too long as well as when the clean activator eventually gets called, the traverse could run for longer than expected, resulting in a hiccup in the system.

The other possibility would be that the clean operation would be a separate traverse by itself, which would be run separately from the store operation. This interpretation would require some repeated thread to run this traverse in a timed manner to ensure that data with no purpose would get removed. However, if this timer function running the traverse gets knocked out, no clean operation would get called at all.

We believe that the second possibility makes more sense, even though the developer must ensure that the timer function gets called regularly. The reasoning is that we think that when a traverse gets run, the execution must finish as soon as possible. Adding extra clean operations when time is often of essence would only bog down the speed of getting any response.

Worth noting with both functions is that the limit checks would prevent any data that does not serve its purpose anymore is not allowed by any process. One could even argue that if a given limit check fails, then there should be a quick check whether that piece of data should get removed instantly. However, that again goes against the reasoning why we thought the second possibility is the better option to handle the clean operation, which is to make sure a traverse can finish as soon as possible.

Worth noting is that the original PA-DFD does not have a delete flow directed towards the policy database, which would mean that the data would get deleted as expected but that the policy would remain in the policy database, which is not acceptable. Even though one could argue that a delete flow to the policy database is obvious, that is something we still need to make clear when designing HPA-DFD.

With the second option of having a timed event being the better option, in our

opinion, we could theoretically make a traverse out of three flows. However, we then stumble upon the issue that we have to start the traverse with an external entity. This scenario would be easy if each database only gets used by one external and their traverses, but that is not always the case.

There was an idea of creating a particular clean external entity for each DFD responsible for ensuring these clean activators would get called, but we do not think that is the way to go. Nevertheless, there are two options. Should there exist one traverse for calling all of these clean activators? Or should there be one traverse for each of them? Both have pros and cons, but the major con that exists for both options, and having a clean external activator, would be poor scaling when the number of databases increases.

However, let us say that we have a traverse responsible for cleaning the database; what if there needs to be a cascade of queries to ensure that everything connecting to one piece of data gets removed from other databases as well. For example, if there is a packaging system where a user has a history of all packets sent and received, and there are two databases: one for users and one for packages. When a user gets deleted by the clean operation, there need to be additional queries done to remove all packages connected to that user since those packages can contain personal information such as phone numbers or addresses.

That brings us to the conclusion of how to implement the clean activator with us not doing anything with it and let the developer, who has the references to the two databases, implement their querying and deleting of old data.

With that decision, we lose a bit of the strength of the PA-DFD, which is to make sure the developer has an easier time designing with privacy in mind. We do not force the developer to implement a clean timed function. The sole comforting fact is that the data will not, in the worst case, get processed by any process since the limit checks would stop it.

5
Discussion

This discussion will evaluate and reflect upon the two algorithms, Ray and Holt, described in Chapters 3 and 4, respectively.

5.1 Case study

This case study section will evaluate our two solutions, Ray and Holt, by extending the examples presented at the start of Chapters 3 and 4 in their respective workflow example. We will start by explaining what we will extend with, then evaluate, first based on the DFD and then on the PA-DFD.

As a recap of what the example in Section 3.1 was all about, it had to do with being able to query users from a database and sending marketing to them. When PA-DFD entered the picture, we added checks to ensure that the users had approved that their email address could be used to send marketing.

Figure 5.1 shows how we have extended the functionality present in Figure 3.1 from Chapter 3, with two more traverses: add a user and delete a user. Note that Figure 4.1 for Chapter 4 still has the same functionality as Figure 3.1, but one traverse was removed and also extended to how the remaining traverse would look after being transformed into a PA-DFD. We will later in this chapter shows how Figure 5.1 gets transformed into an HPA-DFD that is used to represent the PA-DFD behind the scenes.

We decided to have a new and separate external entity to support these two new traverses and further encourage the separation of concerns. Adding the traverses instead of starting from the Company external entity could also work. However, we see external entities as having traverses that do the same things roughly.

Modifying the user database was a considerable functionality missing in the workflow example for Chapter 3, and we think adding those traverses helps evaluate a more realistic situation, especially when it comes to adding the relevant user policies for a given user. Right now, we would have to use the database references to add to the databases.

We wanted a user-friendly way to interact with this DFD and get to use the traverses through the external entities. This got achieved by implementing a command-line

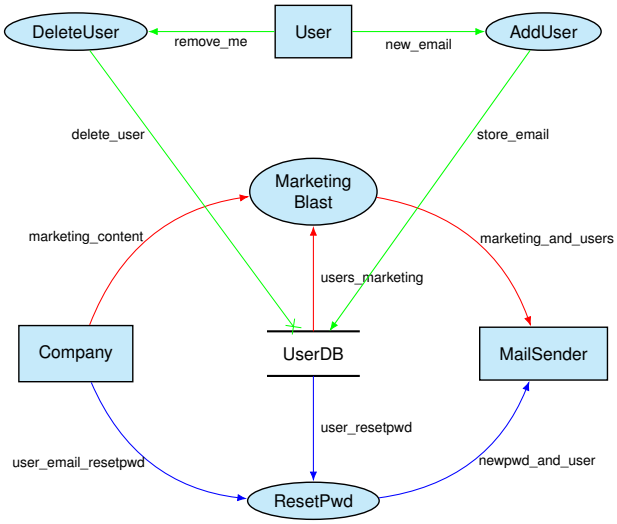


Figure 5.1: Updated DFD with new activators and flows in green and blue.

interface (CLI) program. Figure 5.2 shows some examples of how to interact with the CLI. Note that adding and deleting users will be implemented with this case study.

```
CLI started
>:  user add alice@example.org 25h "login" "marketing"
>:  user add bob@example.org 10h "login"
>:  user remove bob@example.org
>:  clean
>:  company resetpwd alice@example.org
>:  company marketing "Buy Our Product"
```

Figure 5.2: Code examples how to interact with the CLI.

Running Ray results in the code shown in Figure 5.3. We get, in total, two new requirements files and one abstract external entity.

This is not the only way to design the functionality to modify users. Figure 5.4 shows another way where instead, the two traverse goes through one process called `UserManagement`. The fact that developers can decide this for themselves is something we see as a significant strength of Ray's flexibility. This, in turn, helps our case with Contribution 1.

Figure 5.5 displays the traverse setup done to then be able to generate the methods needed two go through the new traverses, which are again pretty straightforward. Figure 5.6 shows how we decide to set up the new processes that are implementing the two requirements: `DeleteUser` and `AddUser`. As seen, the code is quite simple and makes sure that the requests to the database that are being done are not null.

```java
@Generated("holt.JavaFileGenerator")
public abstract class AbstractUser {
  /* ... */
}

@Generated("holt.JavaFileGenerator")
public interface AddUserRequirements {
  /* ... */
}

@Generated("holt.JavaFileGenerator")
public interface DeleteUserRequirements {
  /* ... */
}
```

Figure 5.3: New files generated by Ray.

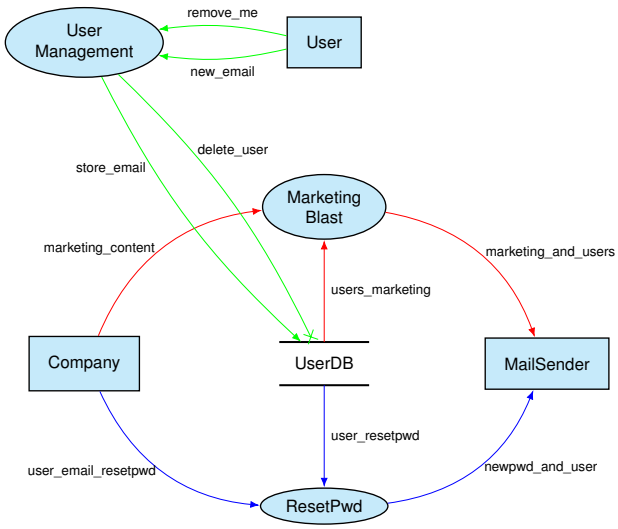


Figure 5.4: Alternative way to design the new functionalities.

The next thing we need to look at is the two new methods that need to get implemented in `UserDB`. It works in the same way as the methods in Figure 3.11, which has the code for `MailSender`. However, this time we are meant to modify the database that the activator `UserDB` manages. As previously shown, this database is just a hashmap for simplicity's sake. The name for these two methods will be the name of the traverse. Unfortunately, the ability to change names for output methods was not implemented due to time constraints. Figure 5.7 shows the code for implementing the functionality needed for the two traverses.

The two methods generated for `UserDB` show the strength of Ray regarding Contribution 1. `UserDB` does not have to be aware, other than to implement these two new methods, that two new activators have been connected to it.

```
@Traverse(
        name = "AU", // Add User
        startTypes = {@Output(type = Email.class)},
        order = {"new_email", "store_email"}
)
@Traverse(
        name = "DU", // Delete User
        startTypes = {@Output(type = Email.class)},
        order = {"remove_me", "delete_user"}
)
@Activator
public class UserEntity extends AbstractUser {
    /* ... */
}
```

Figure 5.5: `@Traverse` setup.

A slight detour we want to present is that inside both `MarketingBlast` and `ResetPwd` processes, they now have access to the methods that have to do with saving and deleting users. A developer with experience with the system might know that the query definitions should not be used to save or delete anything. However, there might be a chance that a new developer could get confused when accessing the database reference inside the query definition and seeing them being able to access the store and delete methods. This is why we chose to design the `@QueriesFor` annotations so that the architecture of the system can create a specific class that has the mission to prevent a developer from misusing those potentially devastating methods. Figure 5.8 shows what this new `UserDBQuerier` class looks like, `UserDB` will now have a method for instantiating the querier. Note that the name for the query methods that were used by `MarketingBlast` and `ResetPwd` processes does not change, which means we do not have to modify their query definitions.

Ideally, we want a method inside a process only to have to do one thing, but right now, the reset process with its usage when sending an OTP does two things: formulates an `EmailAndContent` and generates a new password. Let us say that we want to separate these two functionalities. We can do this by adding a new process that does not have any input but just an output that will get used by our `ResetPwd` process to get a freshly generated one-time password. Figure 5.9 shows what the new DFD looks like and Figure 5.10 has the new method for the `ResetPwd` process. This shows how easy a developer can modify the DFD and get minimal impact for the rest of the traverse and system as a whole. The only thing that must be modified and added would be adding the new flow to the traverse. If the developer ever needs a generated password elsewhere, they add a new flow between the `PwdGen` process to any given activator, and a new generated password will get delivered.

```java
@FlowThrough(
        traverse = "AU",
        output = @Output(type = User.class),
        functionName = "addUser"
)
@Activator(instantiateWithReflection = true)
public class AddUserProcess implements AddUserRequirements {
    @Override
    public User addUser(Email email) {
        if (email == null) {
            throw new IllegalArgumentException();
        }
        return new User(email);
    }
}
```

(a) `AddUserProcess`.

```java
@FlowThrough(
        traverse = "DU",
        output = @Output(type = Email.class),
        functionName = "deleteUser"
)
@Activator(instantiateWithReflection = true)
public class DeleteUserProcess implements DeleteUserRequirements {
    @Override
    public Email deleteUser(Email email) {
        if (email == null) {
            throw new IllegalArgumentException();
        }
        return email;
    }
}
```

(b) `DeleteUserProcess`.

Figure 5.6: New activators.

Now that we have some evaluation done when looking at the DFD, it is time to explore how we can support this after activating the privacy-aware functionality. We will base the newly presented Figure 5.9 that has the new `PwdGen` process. What follows is Figure 5.11, which has two subgraphs where each new traverse is depicted with the reasoning being that there will be a lot of requests, limits, etc., that will get generated when transformed to PA-DFD. Note that we will talk about this as if it is a PA-DFD and not the HPA-DFD which is what it is behind the scenes.

```java
@Activator
public class UserDB implements UserDBRequirements {

    private final Map<Email, User> userMap = new HashMap<>();

    @Override
    public void DU(Email email) {
        userMap.remove(email);
    }

    @Override
    public void AU(User user) {
        userMap.put(user.email(), user);
    }
}
```

Figure 5.7: Code for ending the two traverses `AU` and `DU` in `UserDB`.

```java
@QueriesFor(UserDB.class)
public class UserDBQuerier {
    private final UserDB userDB;

    public UserDBQuerier(UserDB userDB) {
        this.userDB = userDB;
    }

    public List<User> getUsers() {
        return userDB.getUsers();
    }

    public User getUser(Email email) {
        return userDB.getUser(email);
    }
}
```

Figure 5.8: Querier class for UserDB.

To generate all the new requirements files, we add the privacy-aware flag to the `@DFD` and let Ray run. As stated in Chapter 4, the developer does not have to do much when switching into this mode. The only thing is to make sure that objects that are used as policies implement the `Policy` interface we provide. All the requests and reason classes with their methods will do now, is to forward the data they retrieve. Now we will show the limit activator that resides between the `AddUser` and `UserDB`. Figure 5.12 shows the code that gets used, and it is going to be the same check that

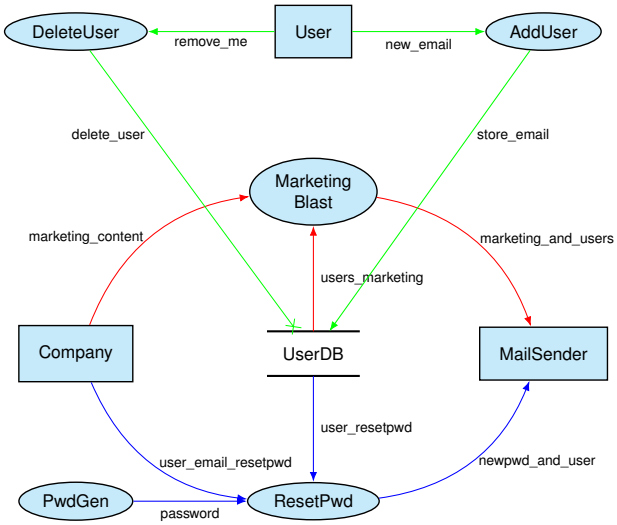


Figure 5.9: New DFD with the new process for generating passwords.

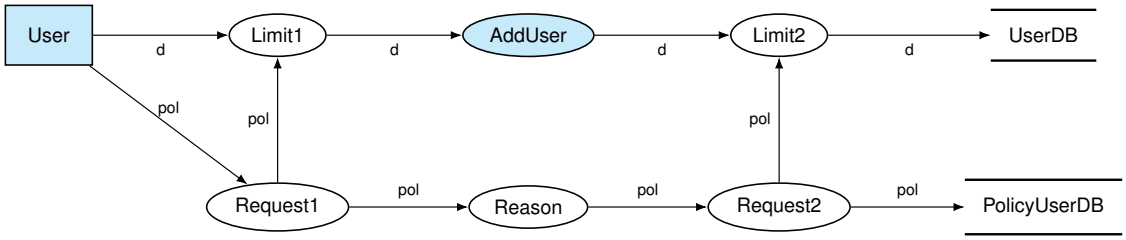
```java
@Override
public EmailAndContent createEmailContent(Email email, User user,
 ↪  Password password) {
    if (user == null) {
        throw new NullPointerException("User not found");
    }

    return new EmailAndContent(
            email,
            new EmailContent(
                    "New password for " + email.email() + " is " +
                     ↪  password.password() + "."
            )
    );
}
```

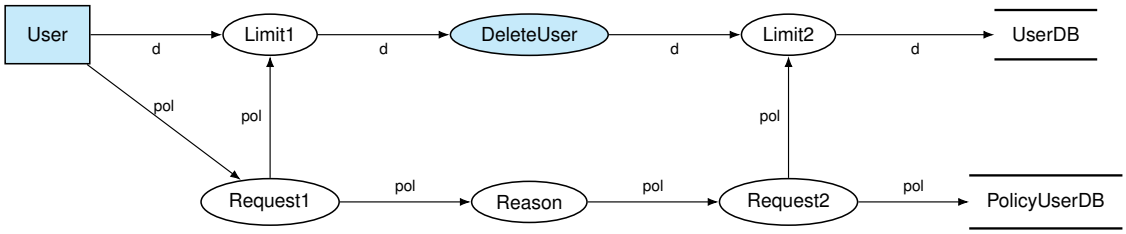
Figure 5.10: New method for `ResetPwd` process.

gets used when retrieving the emails for the other traverses, as depicted in Chapter 4's workflow example.

All the classes that get generated might seem unnecessary, but what it does is that it forces the developer to make a conscious decision at every step. For example, it might be evident that there should be a check between the `AddUser` and `UserDB` activators to ensure that we are not saving something that should not be saved. The external entity that uses this traverse should not need to verify that the correct agreements have gotten approved by the user. That is what PA-DFD is all about and what it must help the developer do.



(a) PA-DFD for the *Add User* traverse.



(b) PA-DFD for the *Delete User* traverse.

Figure 5.11: Traverses in PA-DFD.

```
1 public static Predicate<User> marketingPredicate(Map<User,
  ↪  UserPolicy> userPolicyMap) {
2     return user -> userPolicyMap.get(user)
3         .agreements().contains(AccessUserReason.MARKETING);
4 }
```

Figure 5.12: Example of reusable predicate logic.

Chapter 4 spelled out our reasons for not doing anything regarding the clean activator. These reasons go against what was depicted above regarding wanting the developer to make a conscious decision. However, as the reasoning says in Chapter 4, we can not predict exactly how the data structure is and thus comfortably provide the interface methods that can clean unwanted data properly. What we will do to combat this is to create a class called Cleaner and add a method that can be used to query the policy database to check if any data does not have a reason to be persisted anymore. Figure 5.13 depicts this new class.

5.1.1 Summary

The code can be found on GitHub [8]. This case study shows a fully functional example when using Ray and Holt. The functionality includes:

- Adding and deleting users with specific agreements.
- Sending marketing to users who have accepted receiving it and not being able to send it to those who have not.
- Sending OTP to users.
- Cleaning the database of users whose expiry time has passed.

```java
/* Clean.java */
public void clean() {
    this.userDB.getUsers().forEach(user -> {
        UserPolicy userPolicy = this.userPolicyDB.getPolicy(user);
        userPolicy.getDeleteBefore().ifPresent(deleteBefore -> {
            if (deleteBefore.shouldDelete()) {
                System.out.println("Removing " + user.email());
                this.userDB.DU(user.email());
                this.userPolicyDB.DU(Map.of(user.email(),
                 ↪  AccessUserReason.DELETE));
            }
        });
    });
}
```

Figure 5.13: New class.

- Simulating time going forward in order to simulate time expiry.

To summarize this case study regarding the three contributions:

- **Contribution 1**: We showed that Ray is flexible, for example, when adding new activators. We also showed that Ray is functional as the implemented CLI is working.
- **Contribution 2**: Our case study showed that activating Holt helped the developers to design with privacy in mind.
- **Contribution 3**: The case study also showed that our implementation of policies worked with the PA-DFD to enforce privacy.

5.2 Limitations

This section will go through issues and thus limitations that we have had to put on this thesis.

There is the issue of creating many classes to handle the amount of requirements interfaces that get generated, and this issue only deteriorates when having privacy-aware activated. The risk is that there are so many classes a developer has to create to handle all of them that confusion can occur when trying to separate them with packages and names.

If the developer makes a mistake and tries to specify the traversing order incorrectly, such that a given activator has not received all its input, the code in the abstract external entity will not compile. While this is not the best way to let the developer know of its mistake, they can find out before running any code of their mistake, which only talks strength about the fact that we generate the runtime instead of

setting it up dynamically.

When designing our solution for PA-DFD, we knew we had to generate some of the activators ourselves to make it easier for the developer to use the privacy-aware aspect. For example, Querier was not something we wanted to force on the developer when the only task it has is to send data forward. However, there might be unforeseen situations where a developer needs to modify one of these classes. Maybe a developer wants their own class to return given the data and policy.

Everything generated from our algorithm always starts from scratch, ending with the saved Java files, which has the problem that if something goes wrong after a rebuild, what is generated is affected. The developer gets faced with at least one compiler error per class they have that inherits some of our previously generated code. This process can be jarring, and finding the actual error message can be tricky. The issue often arises if there is some mismatch in the metadata supplied via annotations from the developer.

One of the goals for generating code for DFDs was to help the developer keep their domain logic inside the activators as loosely coupled as possible, but this can have the fallout of not knowing just depending on the input in what context a given activator gets used, especially for processes. This issue is particularly apparent if the developer is not specifying the output types for the processes and queries; since then, the default class `Object` gets used everywhere. However, using the annotations does help, but the developer needs to be aware that they use straightforward classes and do not default to using other Java default classes such as String which could lead to more confusion.

5.3 Related work

This section contains a discussion of previous work related to our thesis.

5.3.1 DFD

In the paper ”Formalization of the data flow diagram rules for consistency check” by Rosziati Ibrahim and Siow Yen Yen, they formalize rules for consistency checks as well as rules for a DFD definition [15]. Our DFD rules, specified in Section 3.2.1, are similar to the ones by Rosziati Ibrahim and Siow Yen Yen, as well as the rules already specified in the paper by Antignac et al. [5]. While the rules from these two papers are similar, there are some essential rules that we only have in common with Ibrahim et al. These rules were mainly related to the uniqueness of names and syntax rules for the DFD.

The paper by Rosziati Ibrahim and Siow Yen Yen also discusses the idea of different levels of a DFD, where the highest level is known as a context diagram [15]. This level concept is very similar to the idea of refinement that was discussed in the paper by Antignac et al. [5]. Both ideas try to solve the problem of decreasing the size

of DFDs. Ray does not utilize a context diagram or refinement. However, designers and developers can utilize that the Ray can handle multiple DFDs similarly to these concepts.

5.3.2 Unified Modeling Language

Unified Modeling Language (UML) is a concept that has a lot in common with DFD. Like DFD, it gives some information on how the system's architecture is designed. However, unlike DFD, it has a different focus on the system's structure, including fields and inheritance, instead of how data flows within the system.

Another difference between UML and DFD is the amount of research surrounding code generation from each of them. There has been much research on automatically creating code from UML, but minimal research on code from DFD [16][17].

In the paper "Automatic generation of Java code from UML diagrams using UJECTOR" by Muhammad Usman and Aamer Nadeem, they did code generation similarly to what we have done in this thesis [18]. However, where we have used DFD and Java Annotations, they used multiple different UMLs. The result of their generation is a set of files that get exported while our result is a firm connection to the developers' codebase. This is where Ray excels in our opinion, as it connects the project's design and code on a deep level. If the DFD changes, so does the code.

5.3.3 Policy frameworks

Several policy languages and frameworks have been developed previously. One of them, Enterprise Privacy Authorization Language (EPAL), is a "formal language for writing enterprise privacy policies to govern data handling practices in IT systems according to fine-grained positive and negative authorization rights" [19]. An EPAL policy consists of data-users, purposes, privacy actions, obligations, and conditions and could be used as a policy within a PA-DFD system. Another, P3P, is similar to EPAL in its purpose and technique but is mainly meant for websites, but it could very well be used within a PA-DFD system [20].

While both of these alternatives could work within a PA-DFD system, we chose not to take advantage of them to have a more flexible algorithm. The flexibility also ensures that we do not force the developers into a specific framework that does not fit into their workflow.

5.4 Future work

When we started with this project, we had the goal of generating code from a PA-DFD, but quickly we realized that we needed to split up the work into two, which was to first focus on generating code from a DFD. That works because a PA-DFD

is nothing more than an extended DFD. We think that designing an algorithm that generates runnable code from a DFD could be a thesis topic on its own. However, we also think that taking Ray and Holt, and evaluating it, testing it on other developers, and improving its design can be a new thesis topic. All design choices we made had the end goal of trying to achieve our primary goal of generating code from a PA-DFD; this goal could have clouded our judgment somewhat. Section 5.1 included a small case study, but we think there is room for a more extensive one.

The runtime that Ray generates gets executed synchronously, but there is much potential to expand Ray and investigate possibilities to run things asynchronously. For example, if there needs to be four queries from four different databases in one process, and right now, they would execute one after another. Significant performance gains could get made if they got scheduled to get called with more than one thread.

Regarding our PA-DFD solution, we feel that it is great that we provide such extensive flexibility to the developers; however, future work could investigate precisely how a system with a more firm connection a policy management system would work.

An idea that grew over time during this thesis's work was that much boilerplate code could get generated by some plugin in an IDE. For example, when an activator gets added to the DFD, a plugin could be able to recognize that event and ask the developer if it should generate a class that implements that requirements file. Generating classes could be beneficial when there are a lot of requests and limits activators that need to get created.

Another extension to Ray would be to investigate how to use parameterized types, like `Map<>`, in the data flows. Our current implementation does allow for Java `Collection` to be sent between the activators, and while they are useful, they are also very limiting.

6
Conclusion

We conclude this thesis by first answering the problem statement from Section 1.1 and then finishing by reviewing the contributions we stated in Section 1.2.

We began our problem statement by asking the question:

RQ 1. How can we bridge the gap between PA-DFD and code?

This question was the long-term goal that guided us throughout our thesis work. We decided pretty early that we needed to split our work into two, first designing a DFD framework that we then can use to realize the PA-DFD. We were successful with both of them. The case study is a good statement of this fact.

- *RQ 2. What sub-algorithms does our algorithm need?*

Both Chapters 3 and 4 describe the sub-algorithms used to generate our final solutions more specifically. The final solution worked due to the collaboration between the two systems, Ray and Holt. Ray took the designed DFD and generated activators and flows from it. When the developer introduces the traverses order, the runtime gets generated along with the methods inside the requirements classes. Holt has a few entry points where it can add new activators, modify existing ones, add new flows, and move query definitions, amongst other things.

There were two specific technical questions that we stated as well.

- *RQ 2.1. How can we represent policies and then enforce them?*

We demonstrated a simple but effective way of structuring a policy with strings of agreements that the user had approved and adding an expiry date when the data should get removed at the latest. Related work goes through examples of other policy management systems and how they could work within our flexible solution.

- *RQ 2.2. How do we make sure that expired data is never processed?*

We make sure that a limit activator, a guard activator, and a request activator get generated and called before data gets moved between two activators. The developer can look at the generated code to ensure that no expired data is processed. They can run tests, especially on the Predicate functions for the limit activators.

- *RQ 3. How do we ensure the correctness of our algorithm?*

This statement got, in a sense, answered for the last statement; we do not have formal proof for either Ray or Holt. However, we have a good case study that goes through a realistic situation where we show a good result for Ray and Holt.

A strength is that all code generated is there for display, which the developer can verify. We have no external library in a separate git repository that the developer has to go through to ensure their program works as expected when using Ray and Holt. We can then ensure that the developer at least has the power to guarantee the correctness of their entire program, which then, by logic, implies that Ray and Holt works.
The following two questions help us answer the above problem statements as well.

- *RQ 3.1. How can we test the implementation of our algorithm?*

We tested both Ray and Holt with small examples throughout our work and ended with a more extensive case study. To test correctly, we need to test it as if we did not know that we used Holt. Writing good examples and acceptance tests is enough.

- *RQ 3.2. What requirements do we need to put on developers using our algorithm?*

We assume one fundamental requirement: the limit activators are correctly implemented and deterministic since we depended on them for the guard and the log activator. We believe that any other human error will result in some compile error, such as the incorrect order of flows or badly inherited requirements.

- *RQ 4. How do we ensure the implementation of our algorithm is usable?*

This is one problem statement that we did not fulfill to the wanted extent. If we could redo this thesis, we would try to involve other developers and get their feedback on our design decisions. We believe still, however that we have designed something useful.

Next, we will go through and review the three contributions we introduced in Section 1.2:

- **Contribution 1: We have designed a flexible and functional DFD framework, called Ray.**

We have shown that our DFD design is flexible and functional through various examples with our case study. The case study brings forth, for example, the strength of being able to quickly add a process to a traverse without interfering with other processes. We have also shown how easy it is to use their solution when it has gotten built with Ray.

- **Contribution 2: Holt, together with Ray, can help developers to design with privacy in mind.**

The case study we provide discusses a proper implementation of a project where privacy gets taken into account at each step. We take the theory that previous authors have specified regarding PA-DFD and realize it into running code, thus proving that using PA-DFD to create a sound system that considers privacy is possible.

- **Contribution 3: The flexible design of Holt lets developers use their representation of policies to enforce privacy.**

Our case study and the related work show that how we represent policies can be

usable in different ways. We believe that Holt is flexible enough to support many other different kinds of policy frameworks.

In the end, we are happy with our work. We believe we have designed a good DFD code generation solution that is flexible enough to solve real problems. We also feel that Holt was a good attempt of realzing PA-DFDs.

Bibliography

[1] "Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec (general data protection regulation) (text with eea relevance)," *OJ*, vol. L 119, 2016-05-04.

[2] "Integritetsskyddsmyndigheten: Imy," May 2021. [Online]. Available: https://www.imy.se/en/organisations/data-protection/this-applies-accordning-to-gdpr/privacy-by-design-and-privacy-by-default/

[3] P. Schaar, "Privacy by design," *Identity in the Information Society*, vol. 3, no. 2, pp. 267–274, 2010.

[4] T. DeMarco, "Structure analysis and system specification," in *Pioneers and Their Contributions to Software Engineering*. Springer, 1979, pp. 255–288.

[5] T. Antignac, R. Scandariato, and G. Schneider, "A privacy-aware conceptual model for handling personal data," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2016, pp. 942–957.

[6] ——, "Privacy compliance via model transformations," in *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2018, pp. 120–126.

[7] H. Alshareef, S. Stucki, and G. Schneider, "Transforming data flow diagrams for privacy compliance (long version)," *arXiv preprint arXiv:2011.12028*, 2020.

[8] "Holt," https://github.com/DATX05-33/Holt/.

[9] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS quarterly*, pp. 75–105, 2004.

[10] "Java documentation: Processor," https://docs.oracle.com/en/java/javase/17/docs/api/java.compiler/javax/annotation/processing/Processor.html.

[11] "Gradle build tool," https://gradle.org/.

[12] "Javapoet," https://github.com/square/javapoet.

[13] "diagrams.net," http://diagrams.net.

[14] M. Henriksen, "drawio-threatmodeling," 2018. [Online]. Available: https://github.com/michenriksen/drawio-threatmodeling

[15] R. Ibrahim and S. Yen Yen, "Formalization of the data flow diagram rules for consistency check," *arXiv preprint arXiv:1011.0278*, 2010.

[16] I. A. Niaz and J. Tanaka, "Code generation from uml statecharts," in *Proc. 7 th IASTED International Conf. on Software Engineering and Application (SEA 2003), Marina Del Rey*, 2003, pp. 315–321.

[17] T. G. Moreira, M. A. Wehrmeister, C. E. Pereira, J.-F. Petin, and E. Levrat, "Automatic code generation for embedded systems: From uml specifications to

vhdl code," in *2010 8th IEEE International Conference on Industrial Informatics*. IEEE, 2010, pp. 1085–1090.

[18] M. Usman and A. Nadeem, "Automatic generation of java code from uml diagrams using ujector," *International Journal of Software Engineering and Its Applications*, vol. 3, no. 2, pp. 21–37, 2009.

[19] P. Ashley, S. Hada, G. Karjoth, C. Powers, and M. Schunter, "Enterprise privacy authorization language (epal)," *IBM Research*, vol. 30, p. 31, 2003.

[20] L. Cranor, *Web privacy with P3P*. " O'Reilly Media, Inc.", 2002.

A
Annotations

```java
@Target(ElementType.TYPE)
@Repeatable(DFDs.class)
public @interface DFD {
    String name();
    String xml();
    boolean privacyAware() default false;
}
```

(a) Definition for `@DFD`.

```java
@Target(ElementType.TYPE)
public @interface DFDs {
    DFD[] value();
}
```

(b) Used by `@DFD` to make it repeatable.

Figure A.1: Definition for `@DFD` and `@DFDs`.

```java
@Target(ElementType.TYPE)
public @interface Activator {
    boolean instantiateWithReflection() default false;
}
```

Figure A.2: Definition for `@Activator`.

```java
@Repeatable(Traverses.class)
public @interface Traverse {
    Output[] startTypes() default {};
    String name();
    String[] order();
}
```

(a) Definition for `@Traverse`.

```java
@Target(ElementType.TYPE)
public @interface Traverses {
    Traverse[] value();
}
```

(b) Used by `@Traverse` to make it repeatable.

Figure A.3: Definition for `@Traverse` and `@Traverses`.

```java
@Target(ElementType.TYPE)
public @interface QueriesFor {
    Class<?> value();
}
```

Figure A.4: Definition for `@QueriesFor`.

```java
@Target(ElementType.TYPE)
public @interface QueryDefinition {
    Class<?> db();
    Class<?> process();
    Output output();
}
```

Figure A.5: Definition for `@QueryDefinition`.

```java
@Target(ElementType.TYPE)
@Repeatable(FlowThroughs.class)
public @interface FlowThrough {
    Output output();
    String traverse();
    String functionName();
    Query[] queries() default {};

    QueryDefinition[] overrideQueries() default {};

    String forActivator() default "";
}
```

(a) Definition for `@FlowThrough`.

```java
@Target(ElementType.TYPE)
public @interface FlowThroughs {
    FlowThrough[] value();
}
```

(b) Used by `@FlowThrough` to make it repeatable.

```java
@Target(ElementType.TYPE)
public @interface Query {
    Class<?> db();
    Output output();
}
```

(c) Definition for `@Query`. Used by `@FlowThrough`

Figure A.6: Definition for `@FlowThrough`, `@FlowThroughs`, and `@Query`.

```java
public @interface Output {
    Class<?> type() default Object.class;
    boolean collection() default false;
}
```

Figure A.7: Definition for `@Output`.

B

PA-DFD

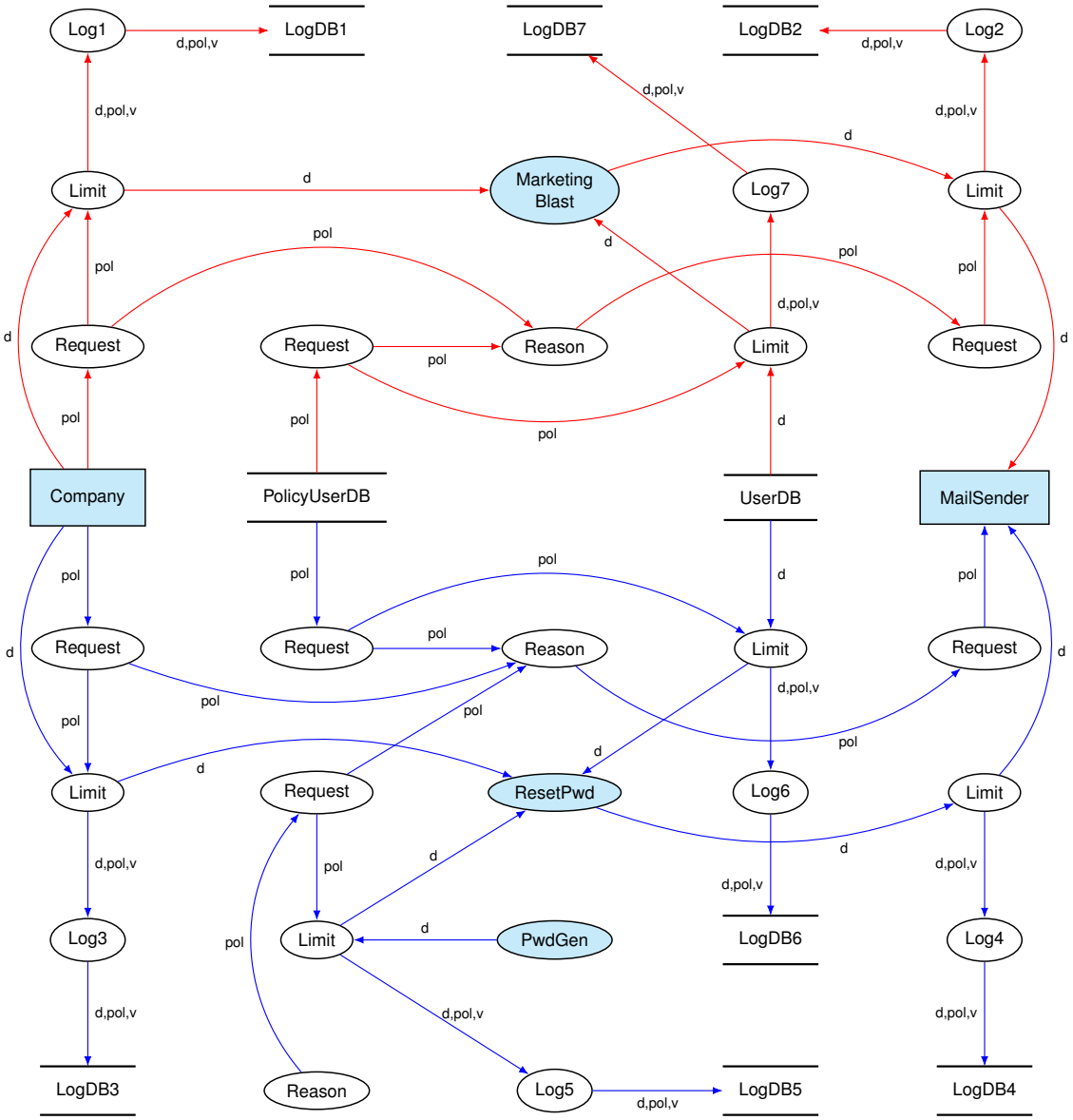


Figure B.1: PA-DFD with two traverses including log and logDB activators. The red traverse is named *Marketing*, shortened `M`, while the blue traverse is named *Reset password*, shortened `RP`. Both traverses start from the `Company` external entity and end in `MailSender`.