A Framework for Creating Policy-agnostic Programming Languages

Fabian Bruckner[1], Julia Pampus[1] and Falk Howar[2]
[1]*Data Business, Fraunhofer ISST, Emil-Figge-Strasse 91, 44227 Dortmund, Germany*
[2]*Chair for Software Engineering, TU Dortmund, Otto-Hahn-Strasse 12, 44227 Dortmund, Germany*

Keywords: Usage Control, Data Sovereignty, Code Generation, Model-driven Software Development, Domain Specific Language, Cross-compilation, Policy-agnostic Programming, Industry 4.0.

Abstract: This paper introduces the policy system of the domain specific language D° (spoken di'grē). The central feature of this DSL is the automatic integration of usage control mechanisms into the application logic. The introduced DSL is cross-compiled to a host language. D° implements the policy-agnostic programming paradigm which means that application logic and policy enforcement are considered separately during the development. Both aspects are combined (automatically) in a later state. We propose the well-defined combination of blacklisting and whitelisting which we define as greylisting. Based on a simple example, we present the different aspects of the proposed policy system. Extensibility of the policy system and D° is another central functionality of the DSL. We demonstrate how the policy system and the language itself can be extended by new elements by implementing a simple use case. For this implementation, we use a prototypically implementation of D° which uses Java as host language.

1 INTRODUCTION

Nowadays, many business models are based mainly on data handling and processing (Zolnowski et al., 2016). Well-known examples for companies that rely on data-driven business models are Facebook, Google, and Uber. For this reason, data is evolving into an increasingly important good. This in turn requires that data is reliably protected, especially if the data is shared or exchanged with different entities. Usage control "as a fundamental enhancement of the access matrix" (Sandhu and Park, 2003) can provide the technical implementation for these requirements.

To provide a common solution for the different entities facing the same challenges during data exchange, the International Data Spaces (IDS) have been developed (Otto and Jarke, 2019). The main goal of the IDS is to provide their participants a common ecosystem that ensures a secure and trustful data exchange. Every participant can define rules that determine how other participants can use parts of own data. To match the specific requirements of different domains, multiple verticalizations are developed. Examples for such verticalizations are the Medical Data Space and the Logistics Data Space.

The IDS offer solutions for tracking and limiting data usage (Schuette and Brost, 2018) as well as for adding usage control to existing software (Jung et al., 2014; Eitel et al., 2019). However, there is no solution to consider usage control when developing new services to be used within the IDS.

In order to close this gap, we developed a domain specific programming language called D° (spoken di'grē). One of the main goals of D° is to take usage control into account from the beginning of software development. In this way, the usage control constructs are directly integrated into D°-executables. At the same time, the correct usage of these mechanisms should be simplified by using D°.

D° is neither an interpreted language nor is it directly translated to machine code. Instead, the D°-compiler performs a translation to an arbitrary host language. We developed a prototypically implementation which uses Java as host language. The generated host language code is later compiled to executable code. This way, it is possible to use D° on all platforms and machines that are able to run the host language.

Different scenarios require different types of applications. For example, on the one hand, a multi-user service that needs to be accessible from remote machines may be best implemented as a `REST`-service. On the other hand, for a data transformation that takes place within a workflow on a single machine, a com-

Bruckner, F., Pampus, J. and Howar, F.
A Framework for Creating Policy-agnostic Programming Languages.
DOI: 10.5220/0009782200310042
In *Proceedings of the 9th International Conference on Data Science, Technology and Applications (DATA 2020)*, pages 31-42
ISBN: 978-989-758-440-4

mand line tool is more suitable.

In order to make D° usable for all these different scenarios, the code generation is capable of generating various kinds of applications in the host language. The application type is determined by the configuration which is part of the application code.

The focus of this paper is the policy system of D° and its different components. The policy system allows to define custom policies which can be attached to other language constructs. D° ensures that the required enforcement takes places in the corresponding situations. We demonstrate how a combination of black- and whitelisting can be used to obtain a very generic enforcement algorithm that can be used for various scenarios. We show that the proposed policy system has a structure that allows the creation of policy-agnostic and highly extensible programming languages. Using Java as an example, we show how the features of a host language can be utilized by the policy system by using cross-compilation.

Related Work. There are various existing approaches and solutions for usage control and policy systems. JFlow extends the Java language by specific annotations in order to add the possibility for statically-checked information flows (Myers, 1999). In addition, it is extended by a byte code checker to provide information flow checks on source code as well as on byte code (Barthe et al., 2006). Since JFlow is extending the Java language, there is some overhead involved in using it. In addition, the user is responsible for the correct usage of the JFlow mechanisms. The policy system, which we propose in this paper, tries to remove much of the overhead and responsibility from the developer.

Another solution is Jeeves (Yang, 2015). Jeeves is a policy-agnostic language for automatically enforcing information policies. It can either be used standalone or embedded into a host language (Yang et al., 2012). Jeeves has an available extension for faceted values (Austin et al., 2013). One feature of Jeeves is that it does not simple reject data flows which do not comply to policies. Instead, the data can be modified in a manner that the affected policies are fulfilled. The proposed policy system will not provide such data manipulating features since it can lead to unwanted behavior in some scenarios.

LUCON uses a label-based approach in order to track and limit how data is used within distributed systems (Schuette and Brost, 2018). Actions within workflows can create, consume, and check the existence of labels which are attached to the data. In order to efficiently use LUCON, profound knowledge about the services, which shall work with labels, is required. Otherwise it is not possible to define meaningful and useful rules and labels. On the other hand the proposed policy system does not require detailed knowledge about the internal implementation of elements to be used correctly.

Another domain specific language for developing data-centric applications is LIFTY (Polikarpova et al., 2018). It follows the policy-agnostic programming paradigm and uses program synthesis to patch the policy enforcement code into the application logic. This application logic was previously implemented without respecting policies. LIFTY focuses on information flow and (unintentional) information leaks which are prevented at compile time. The proposed policy system is operating at runtime.

Outline. The rest of this paper is structured as follows. Section 2 introduces the example which is used throughout the paper in order to better demonstrate the different components of the proposed policy system. Section 3 shows the different design goals that were applied during the development of the policy system. Section 4 introduces the different components and functionalities of the policy system and how the previously introduced design goals are fulfilled.

2 EXAMPLE

In order to provide a better understanding of D°, we start with a simple example that will be used throughout this paper. The example is elaborated and mentioned at appropriate places in the paper.

CompA is a company that specializes in providing backup services for its users. In addition, CompA is involved in various IDS verticalizations as a participant. Therefore, CompA wants to develop a backup service which is well-suited for usages within the IDS.

CompA wants the backup service to provide an `HTTP`-endpoint to which users can send files. These files are stored on the backup server. The users have to authenticate by using `JSON` Web Tokens (`JWT`) created by CompA's authorization server. The files can neither be overwritten nor deleted. Each user has a contract with CompA. These contracts define how much storage space the user can use for his or her backups. There are no restrictions on the files that can be sent to the backup service. Therefore, the only policy for this scenario is that each user has to comply with their contractually regulated quota.

Because the service has not been implemented yet and there is a policy which has to be enforced, CompA decides to use D° as programming language.

```
1  App configuration
2  namespace : example
3  name : backupService
4  port : 8080
5  url: backup
6
7  App code
8  [fileName = $FileName, fileContent =
       $Base64] -> begin
9      result = $Text()
10     [result] = ProtectedBackupFile(
           fileName,   fileContent);
11     return [result]
12 end
```

Figure 1: Backup Service Developed with D°.

The corresponding functionality for file retrieval will not be covered within this example.

Figure 1 shows the D° implementation of the service described above. The code for the service is devided into two sections. The first section ranges from line 1 to 5 and contains the application's configuration. Beside general configuration items for the application name and namespace, the provided configuration is used to determine the generated type of application. Since the shown service has configuration items for a used port (line 4) and url (line 5), the compiler will generate a Data App with a HTTP-interface. The service will be reachable at `${IP}:8080/backup`. The second section of the code goes from line 7 to 12 and contains the actual application logic. The inputs of the service are described in line 8. The given application has two inputs: An object of type `FileName`, named `fileName`, and an object of type `Base64`, named `fileContent`. A variable of type `Text`, named `result`, is declared in line 9. The variable is initialized with the default value for the data type `Text` which is the empty string. Line 10 contains the call of the activity `ProtectedBackupFile`. The application's inputs are used as inputs for the activity as well. The application returns a single value of type `Text` which is written to the previously declared variable. The result is returned to the caller in line 11.

Although the description of the example contains a policy which needs to be enforced by the application, the actual code lacks matching source code for policy enforcement. This is because the policy definition and enforcement code is not part of the application itself. The application logic and policy code will be combined automatically during the compilation. This is a central feature of D° and will be described in the following sections.

3 DESIGN

Access control is not capable of fulfilling the requirements of modern applications regarding the security of used data (Sandhu and Park, 2003; Park and Sandhu, 2004; Lazouski et al., 2010). For example, nowadays, it is not sufficient to use policies which define who is allowed to access specific data.

> "In today's highly dynamic, distributed environment, obligations and conditions are also crucial decision factors for richer and finer controls on usage of digital resources." (Sandhu and Park, 2003)

To match these new requirements, the ABC models for usage control have been developed (Park, 2003). Access control uses attributes of subjects and objects in order to determine if specific resource accesses are permitted. The ABC models extend the traditional access control by obligations and conditions. That way, it is possible to not only make access based decisions. Rather, it is possible to define rules which apply for the usage of resources.

At the same time, realistic and useful usage policies can be quite complex (Zdancewic, 2004). Policy systems, that are capable of enforcing many different kinds of policies, use rather low levels of abstraction within policies. This prevents ambiguities regarding the implementation and simplifies the implementation of the system since each individual enforceable element is rather simple. At the same time, this means that policies have to be constructed with the primitives provided by the policy system. Based on that, defining a policy that allows to send aggregated data to all users and unedited data only to specific ones may not be an easy task.

In addition to the increased requirements, the complexity to implement correct solutions for protecting data has increased. There are many different types of policies which may require very different checking and enforcement logic. These policies may have to be enforced at different times and may get combined in order to form more complex policies.

While solutions that allow to extend existing software by usage control are required, the problem can be handled differently for software which gets developed in the context of these new requirements. It is possible to consider usage control right from the beginning of the development and integrate it directly into the application. This could be achieved by manually implementing the required handling and enforcement of policies, or by using middleware. Both approaches can be very cumbersome and require expert knowledge in order to create well-functioning systems. A better solution may be to integrate the re-

quired usage control concepts directly into the used programming language and hide the complexity of usage control as much as possible from the developer.

In order to address the presented problems, we developed the domain specific programming language D°. It aims at the development of data processing applications (so called *Data Apps*). D° is cross-compiled to a host language. Most high-level programming languages can be used as host language. For evaluation purposes, we implemented a prototype which uses Java as host language. During the compilation, the language's policy system is integrated into the application. The usage control system of D° aims an easy usage. We identified a set of design goals for the policy system of D° which will be covered in the following paragraphs.

3.1 Policy-agnostic

The policy-agnostic programming paradigm is based on the separation of program code and policies which need to be enforced and was introduced as part of the programming language Jeeves (Yang, 2015; Yang et al., 2012). The paradigm aims at the usability and enforcement of information flow policies. The program code and the policies are combined during a later stage (runtime or compile time) to ensure that the usage control policies are enforced in all necessary situations. For the developer, however, there is no difference whether there are policies or not. D° adapts the policy-agnostic programming paradigm for multiple reasons.

The most important one is that it allows to form different groups of developers with different skills. While a policy expert can implement usage control policies, a developer can use D° to develop Data Apps. Both of these two user groups can work independently. The only situation where the two user groups may have to interact is when application logic and usage control concepts need to be combined.

In addition, the policy-agnostic programming paradigm can prevent errors because the code required for policy enforcement is automatically generated, making it less prone to human error.

3.1.1 Example

It is easy to see how D° implements the policy-agnostic programming paradigm if we take a look at the example introduced in Section 2.

Separation of Application Logic and Policy Enforcement. Figure 1 in Section 2 shows the example implemented with D°. Besides the application configuration, the application code consists of a variable declaration, a single statement that creates a backup of the given file, and a return statement that returns the result of the backup creation (e.g. success).

Each D° application code has a special preamble which is used to configure the application. It consists of an arbitrary number of key-value pairs which can be set based on the individual requirements. In the application's code, there is no reference to the policy required to enforce the previously defined behavior regarding quotas and different users.

3.2 Adjustable Enforcement

When it comes to the enforcement of usage control policies, a defined process is necessary. This is important to ensure that it is always clear and reproducible why a specific decision was made by the enforcement logic.

Most common policy enforcing systems either use whitelisting or blacklisting for their enforcement. This has implications on the policies, as well as on the behavior in case there are no applicable policies. Depending on the requirements of the used system, as well as the individual use case, either of these approaches may be most appropriate. For example, on the one hand, an application that performs analysis on critical business data is a good candidate for whitelisting, since all actions that are not explicitly allowed will be prohibited. On the other hand, a service which does statistical evaluation for some general, non-critical data is a good candidate for blacklisting. That way, only the restrictions for this service are expressed and enforced.

Choosing the correct type of enforcing is all about determining whether falsely allowed or prohibited actions are less problematic. In the course of this, errors are a result of policies which are enforced and do not match predefined requirements.

Because the developed policy system should be flexible in order to be applicable for as many use cases and scenarios as possible, a combination of whitelisting and blacklisting is used. Basically, a well-defined process is used to avoid any kind of ambiguities. That way, the decisions made by the policy enforcing are reproducible as well as predictable. This is crucial for the creation of an easy to use policy system.

3.3 Extensibility

It can be observed that well-known and sound methods like e.g. the access matrix do not meet the requirements of modern systems and applications and

therefore need changes and enhancement in order to do so (Sandhu and Park, 2003; Katt et al., 2008; Park, 2003; Park et al., 2004). This is correct, as these methods only allow a limited number of decisions and therefore do not meet current requirements. For example, a system designed solely to determine whether access to resources should be granted cannot be used to restrict usage after access has been granted.

To provide a policy system which can adapt new requirements, at least to some extend, the proposed policy system is designed with a high amount of extensibility. Users should be able to both define new elements within the policy system and add completely new elements to the system.

While this applies to the policy system of D°, it is also a requirement for the language itself. In this way, D° can easily be adapted for usage in different domains.

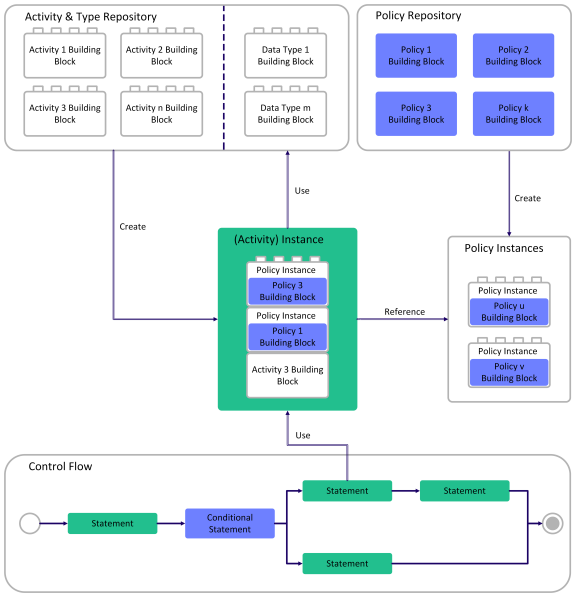

Figure 2: Composing Building Blocks.

4 POLICY SYSTEM

As mentioned before, one key design decision for any policy enforcing system is the enforcing algorithm, which also defines what type of policies the system can handle. If a policy system will be used within known scenarios and environments, it is feasible to find a good matching solution for the system. Since D° aims at fitting as much scenarios and domains as possible while providing a solid and expressive policy system, a combination of whitelisting and blacklisting is used for the policy system of D°.

4.1 Separation of Concerns

In order to provide the separation of application logic and policy enforcement, the proposed policy system is based on building blocks for different language constructs which are combined to achieve the desired behavior. Each building block provides a definition as well as an implementation, if needed. There are various types of building blocks available, e.g. for activities, policies, and data types. These building blocks are the concepts for language elements within D° and cannot be used directly within Data Apps. Activities are the atomic element of executing application logic within D°.

Figure 2 shows how the different types of building blocks are combined in order to form instances which are finally used within the control flow of Data Apps.

To make these building blocks usable within applications, it is necessary to create so-called instances out of the concepts. The concepts form a meta-model of available elements while the instances are the actual model of usable elements which can be used within Data Apps. These instances refer to used concepts, additionally required data (e.g. parameter bindings), and, if available, establish a connection to the implementation. During the definition of some types of building blocks (e.g. for activities), it is possible to reference instances of policy building blocks which need to be enforced during the execution.

With this approach, it is possible to develop and maintain policies and other language constructs completely separate from each other. The policies, which are attached to other building blocks within instances, are hidden to the developer who is using D°. Solely the base building block (e.g. an activity) is visible to the user. That way the policy-agnostic programming paradigm is implemented within the proposed policy system.

4.1.1 Example

The various steps necessary to create usable instances from building blocks can be illustrated using the example presented in Section 2.

Composition of Building Blocks. Since there is only a single logical function (backup file) and a single policy (comply with quota) within the example, we only need one activity and a single policy to provide the required functionality. All elements within D° require a definition which makes them known to the compiler and usable in applications. In addition,

```
BackupFile:
  degree.Activity@BackupFile:
    name:
      Identifier: "BackupFile"
    inputParameters:
      degree.Parameter:
        - name:
            Identifier: "fileName"
          type:
            Type: "FileName"
        - name:
            Identifier:"fileContent"
          type:
            Type: "Base64"
    outputParameters:
      degree.Parameter:
        - name:
            Identifier: "result"
          type:
            Type: "Text"
```

Figure 3: Definition of the File Backup Activity.

an implementation is required in case the element should perform any logic.

Activities within the context of D° are atomic building blocks that are responsible of providing application logic. The described example requires only a single activity that stores files on the disk to meet the requirements. It is necessary to provide a definition of the activity by using e.g. `yaml`. Among other (optional) things, it is required to specify input and output parameters for activities by name and type as well as the activity's name itself. The activity definition shown here is simplified at some points in order to reduce the complexity and keep the focus on the important aspects of this paper.

The activity used in the example has the file name and file content as input parameters and returns a single value which contains information about the result of the execution. The full definition, that contains all the stated information, can be found in Figure 3.

While the activity could be instantiated on its own and used in applications afterwards, we need a policy to provide the required functionality. The definition of the required policy uses the same syntax and similar constructs like input and output parameters. Because of space restrictions this definition as well as instantiations are not shown here. The same applies to the implementation of the used policy.

Since data on the available quota for different users is obtained at runtime from special informations points provided by D°, the definition consists basically only of the policy name.

After the policy and activity have been defined, it is necessary to create instances from these definitions which can be used within the example application. First, the policy has to be instantiated in order to be usable within the activity. The instance declaration for the policy uses the same syntax as the definition and has to refer to the policy definition which is going to be instantiated.

When policy instances are created, it is possible to bind input parameters of the policy to static values. This can be useful in many situations, but is not used in the example. If the example would have been designed as a single user system, the quota of this single user could be an input parameter of the policy which is set at the time the instance is created.

The final step is to create an activity instance which contains the policy instance. The creation of activity instances is a critical aspect in D° as this is the point where application logic and policy enforcement are combined. According to the policy-agnostic programming paradigm, these two parts of D° are considered separately.

The activity definition in Figure 3 is used by the type system of D°. This system is used during the development to define all data types, activities, and policies (as well as instances for activities and policies). During runtime the type system is resposible for the creation and management of element instances.

The type system used for D° aims to define abstract and domain-specific types and functionalities instead of technical types and functions that may have direct hardware support. In most cases, these representations are not space efficient, but this is not the goal of the used type system. It supports complex sub- and supertype relations (e.g. multiple supertypes) as well as the validation of data type instances in writing operations. Within the used type system, new elements are defined by providing a textual definition in e.g. `JSON` or `yaml`.

4.2 Greylisting

The proposed policy system allows to either use whitelisting, blacklisting, or a special combination of both which we define as greylisting. It is important to not mix up the D°-term greylisting with the approach to detect and prevent spam emails. As can be seen in Figure 4, there are various layers involved in the policy enforcement of D°. The next paragraphs will give a description for each of the layers, starting at the bottom layer.

The policy system of D° allows to define policies for APIs (e.g. network or I/O) as well as general attribute based policies (e.g. time constraints & role requirements). Therefore, the lowest level within the context of the proposed policy system is formed by the APIs of the host language to which D° is cross-

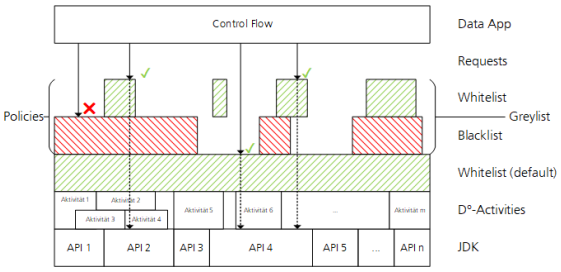

Figure 4: Greylisting Enforcement.

compiled. While the prototype-implementation of D° is cross-compiled to Java code, any other general purpose programming language can be used as host language for D°. The only requirement to the host language is that the APIs to be protected must be interruptible. The benefit when using Java as host language is that various APIs (e.g. network and file I/O) are already instrumented in order to allow the Java security manager to check for policies. Before any method of these instrumented APIs is executed, the security manager is asked for permission. D° uses the same concept and therefore, the prototypically implementation makes heavy use of a custom security manager. For other host languages this feature has to be implemented.

The next layer within the policy system is formed by the atomic execution unit of D°: the activity. Each activity corresponds to an arbitrary complex set of code in the used host language. The only requirement to the host language code is that it must comply to given constraints (e.g. interfaces) to allow correct execution within D° applications. Therefore, any activity can cover none, one, or multiple endpoints of the instrumented host language APIs.

If there are no policies defined, the default behavior of the policy system is used. Every access to the APIs of the host language is allowed. This complies to the policy-agnostic programming paradigm since developers can use the policy system without any kind of policies at all and do not have to make any special assumptions.

To support the classic blacklisting, the policy system allows the definition of prohibition policies (e.g. prohibition to write to a certain folder or prohibition of action for certain users). These policies override the default behavior and can, if there are no allowing policies, cancel the execution if a forbidden action should be executed.

Since whitelisting may be more appropriate for certain use cases, it can be used within the D° policy system. Within the proposed policy system, the policies that allow actions take precedence over all other policies. For example, if there is a prohibiting policy that prohibits writing files at all, and an allowing policy permits writing to the user's home folder, the user can write to his or her home folder despite the prohibiting policy.

We define this controlled combination of white- and blacklisting as greylisting. The set of all policies for a given situation is called greylist. The system is flexible enough to also use normal whitelisting and blacklisting. If there are only prohibitive policies, the system operates like a normal blacklisting system. If there are policies that prohibit all actions in combination with allowed policies, the resulting enforcement is a classic whitelisting approach. The policy system provides special policies that prohibit all actions and thus easily switch to whitelisting.

Besides branching control flow elements, every application developed with D° consists of an arbitrary amount of activities that are executed sequentially. Each of these activities may have attached policies and may or may not use functions from protected APIs. The type and scope of policy enforcement is therefore different for each activity within the application.

During the execution of such Data Apps, there are various situations where policy enforcement is required. On the one hand, policies can be attached to activities in order to enforce pre- and postconditions. On the other hand, each activity can access the protected APIs of the host language. When this happens, execution is interrupted and the policies attached to the activity are evaluated to decide whether the action is allowed or not, using the greylist enforcement described above.

4.2.1 Example

The different enforcement types become clearer when we apply them to the example from Section 2.

Different Enforcement Types within the Example. The example uses a single policy which can either be defined as whitelisting or blacklisting policy. In general, it is possible to translate whitelisting policies into blacklisting ones and vice versa. When it comes to the choice between white- and blacklisting, the decision is not based on expressiveness. Rather, the choice is based on whether the definition of white- or blacklisting is simpler for the given scenario and if false positives (allow actions that should be forbidden) or false negatives (deny actions that should be allowed) is worse in the given scenario.

If whitelisting is to be used for the example, two policies are required. The first one has to prohibit all file writing actions. The special prohibit-all pol-

icy provided by D° can be used for this. The second one has to check the available quota for the requesting user and allow the file storage if there is no policy violation (exceeding the quota).

In case blacklisting is used, a single policy is sufficient to fulfill the requirements. The policy would check the available quota on requests, similar to the one for whitelisting. However, instead of allowing requests that do not exceed the quota, all requests which exceed the quota are rejected.

4.3 Language Extensions

D° is already a domain-specific language, but its domain is the general task of data processing. Therefore, D° is not equally well-suited for different specific domains (e.g. health care or logistics). The problem is that each domain has its own requirements for data types, activities, and perhaps even policies.

In order to address this problem, D° allows easy extensions of all relevant subsystems (e.g. data types & activities). This way, it is possible for users of D° to add missing language elements to the language and ensure that D° can be used for their specific scenario. To create a custom extension which can be used within D°, it is necessary to create definitions and/or instances for the new language elements. Depending on the added elements, implementations must also be provided. The way definitions and instances are created is very similar for all subsystems. In addition, the set of rules to be observed when providing implementations for language elements is similar for the different types of language elements.

4.3.1 Example

There is an example on how to define new elements in Section 4.1.1. In the following, we take a look on how to provide the implementation for the activity used in the example.

Implementing Activities. In addition to its definition, each activity has to provide an implementation that is executed within the applications. The example activity is implemented in Java. To ensure correct functionality, the implementation must use a special annotation and implement the activity interface. In this way, it is possible to dynamically find, load, and execute the activity implementation. The rules which have to be fulfilled may differ for different host languages and therefore only apply to the Java prototype.

Figure 5 shows the implementation of the backup service activity. Besides the handling of input and output variables, there is only one single method call that persists the given file and returns a message indicating the result. This message can indicate both success and failure, for example, if the file name is already in use.

There is another important detail relevant to policy enforcement: file writing is not done by using a standard Java API like `java.io.FileWriter`. Instead, a special API provided by D° is used. The methods thus provided can collect additional metadata. For example, it is possible to determine how much data should be written to a file before the actual write operation or how much data should be sent over an HTTP-connection before sending.

Internally, the D° APIs utilize the standard Java APIs. The decision which APIs are provided by D° to collect additional metadata is based the APIs D° wants to cover with policies. The Java Security Manager is a feature since version 1 of the Java Development Kit (JDK) and allows to restrict access to various APIs (e.g. file and network) by using special policies which are passed to the executing Java Virtual Machine. The prototype implementation of D° utilizes this Java feature since it simplifies the implementation significantly. Since the policy language used for the Java policies is too limited for the intended scope of D°, only the interface of the original security manager is used in order to utilize the instrumentation of various Java APIs. The prototype provides a custom implementation of the security manager and is therefore capable of enforcing D° policies in these situations. In addition, it is possible to restrict the amount of the used security manager's intervention points.

APIs that are instrumented with the security manager are called protected APIs in the context of D°. Each time a protected API method is called, the execution is interrupted and the security manager can allow or deny execution. In combination with the metadata collected within the D° versions of the protected APIs, many policies can be checked and enforced.

Using the D° APIs is not optional and direct access to the protected APIs is blocked by performing stack trace analysis at runtime. The same stack trace analysis ensures that the security manager ignores the boilerplate code generated for each Data App.

4.4 Enforcement

As a result of the policy-agnostic programming paradigm, the users of D° can not evaluate where the policies may be enforced or whether they will be enforced at all. Nevertheless, a well-defined process of where and when policies are enforced is essential in order to ensure the functionality of the policy system. One part of this process is implicitly defined by the

```
1  @ActivityAnnotation("BackupFile")
2  public class BackupFile extends
       BaseActivity {
3  @Override
4  public OutputScope run(InputScope
       input) {
5      String fileName = input.
           getValues["fileName"].read()
           ;
6      String fileContent = input.
           getValues.get("fileContent")
           .read();
7
8      Instance result =
           DegreeFileOperations.
           writeStringToFile(fileName,
           fileContent);
9
10     OutputScope output = new
           OutputScope();
11     output.getValues.put("result",
           result);
12
13     return output;
14 }
15 }
```

Figure 5: Implementation of the Backup Service Activity.

different parts of the policies.

Each policy consists of three different parts. The precondition, the security manager intervention, and the postcondition. While it is not visible from the D°-code, each activity call in the generated code is performed by a so-called *sandbox*. Within the sandbox, each activity call is encapsulated with code for enforcing pre- and postconditions of all policies attached to the activity. These policies are accessible because they are attached to the activity instance. If the validation of any precondition fails due to policy validations, the execution is canceled.

After the preconditions have been validated without errors, the activity logic is executed. There is no policy enforcing code within the activity logic, but, based on the actual code of the activity, the execution can be interrupted by the security manager. This happens every time a method of the protected APIs is called. The security manager retrieves all policies from the currently executed activity and executes their *evaluateSecurityManagerIntervention* methods. Afterwards, the collected permissions and prohibitions are evaluated in order to decide whether the execution must be canceled or not. This interruption by the security manager can occur as many times as desired during the execution of the activity logic and depends only on the activity code.

After the execution of the activity logic is finished, the postconditions of all policies attached to the activity are evaluated. It is important to note that the post-

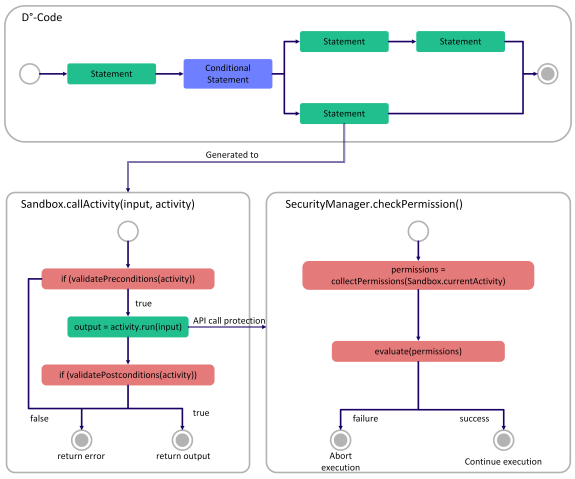

Figure 6: Code Generation for Activity Calls.

conditions are evaluated *after* the execution of the activity logic. Depending on the scenario, this may be problematic since the activity has already been executed and D° does not provide any kind of rollback functionality. For example, on the one hand, if an activity sends data to another machine, this can be critical. On the other hand, if an activity collects data from a database and the policy defines that only a certain format is allowed as return type, the postcondition is the earliest point where this can be checked and violations of this policy do not have such drastic consequences.

The generated code for pre- and postconditions can be compared with an aspect from aspect-oriented programming. An around advice would be used with a join point on the activity's logic method. That way the same behavior can be achieved.

In case the policies need to access parameters of the activity, they have to be mapped when the activity instance is created. In this way, the required parameters are available as normal inputs within the policy methods. If at any time the execution has to be canceled due to policy validations, an error is returned and presented to the user.

Figure 6 shows a block diagram which illustrates what code is generated for a single activity call. The whole diagram uses pseudocode which is shortened in order to illustrate the code generation. It can be seen that, in addition to the actual activity call, all four remaining (red) nodes are policy code that is generated to enforce policies.

4.4.1 Example

Since the different methods of policies and their purpose can be confusing at first, we demonstrate the correct application using the example from Section 2.

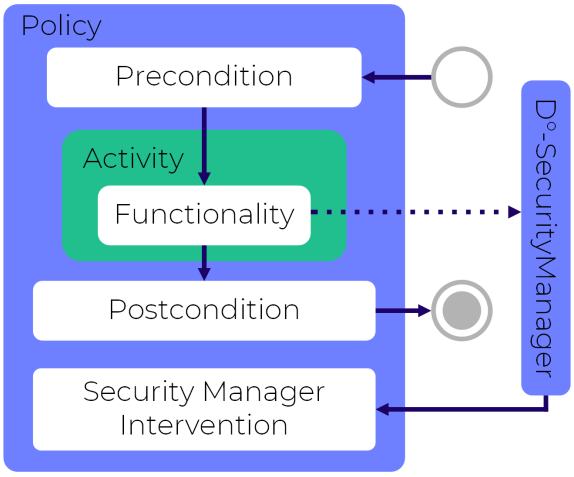

Figure 7: Composing Building Blocks.

Policy Execution Points. As mentioned above, there are three different methods that can provide enforcement logic within policies: the precondition, the postcondition, and the security manager intervention. All three methods have to be implemented when the implementation for a policy is created.

The precondition and postcondition are similar regarding their interpretation. They are evaluated before or after the execution of activities. They return a boolean value that indicates whether the policy is fulfilled. The return value `false` would stop the execution immediately.

The third method of the policy API is the intervention of the security manager. Each time a method of the protected APIs is called, the execution is interrupted and the security manager has to decide if the execution should continue. In these situations, all applicable policies have the ability to create permissions and prohibitions that are taken into account when the security manager decides whether the requested action is allowed to be executed.

Thus, in contrast to the pre- and postconditions, the execution is not discretely canceled due to policy validations. The execution would be canceled later during the evaluation of all prohibitions and permissions by the security manager. In case the execution should nevertheless be terminated immediately, a special exception can be used. This behavior is necessary to implement the greylisting enforcement.

A block diagram showing the control flow for the execution of an activity with attached policy can be found in Figure 7.

We will now take a look at the different parts of the policy used in the example. The precondition checks whether the quota of the requesting user is exceeded. If this is the case, the execution is terminated. Information about the requesting user is obtained from the so-called *context modules*.

Within the context of D°, these modules are named hierarchical data containers, which support different types of data (e.g. read-only, counters, ...) and are placed in the global scope of the Data App. It is possible to read and write data from/to these context modules at any time. They are used to provide data that is required in different places from a central component. Simple queries are used in order to access the elements. They contain the hierarchy of the mentioned elements, separated by dots. To maintain a certain level of traceability of the actions, all writing actions to these modules are logged and persisted. Alternative architectures for policy systems offer similar concepts. For example, the corresponding concept for `XACML` is the PIP (Policy Information Point).

As an example, we take a look at the user data. As mentioned in the example's description, each user must identify himself using `JSON` web tokens. As soon as the request arrives at the application's `HTTP` endpoint, the `JWT` is processed and the contained user information is stored in a context module. This makes it accessible for e.g. activities and policies.

The intervention of the security manager is the central point in this example. Each time the activity wants to write a file to the disk, the execution is interrupted and the intervention of the security manager is evaluated. The file writing process is provided by D°. Therefore, additional metadata has been collected before the call of a file write operation of the JDK caused the interruption.

Thus, the policy knows how much data to write. In combination with the user data from the context modules, it is possible to determine whether this writing operation would exceed the quota and create appropriate permissions and prohibitions.

The element that is executed last is the postcondition. In this scenario, no policy checks are required within the postcondition. Therefore, only one action has to be performed, which is updating the already used quota for the requesting user.

Code Generation. For a better understanding of the generated code and the integration of code which is required for usage control, we have a more detailed look at the actual code generation.

Figure 8 shows the generated class except the `mainBlock`-method which is shown in Figure 9. Despite minior cuts and element renaming, the code is unchanged. The generated application is based on Spring which can be seen on the `main`-method and the used annotations in lines 5 to 7 as well as lines 15 to 17 in Figure 8. The constructor (ranging from line

```java
package example;
    /* imports */
public class backupService extends
    HttpDataApp {

    public static void main(String[]
         args) {
        SpringApplication.run(
            backupService.class,
            args);
    }

    demoProject() {
        super();
        CONFIGURATION_MAP.put("
            namespace", "example");
        /* further handling of
            configuration */
    }

    @RequestMapping(path = "/backup"
        , method = RequestMethod.
        POST)
    @ResponseBody
    public String process(
        @RequestBody String
        inputString) {
        VariableManager
            variableManager = new
            VariableManager(null);
        OutputScope returnScope =
            new OutputScope();
        InputScope input = new
            InputScope();
        input.fromJson(inputString);
        /* input validation */
        try { this.mainBlock(
            variableManager,
            returnScope); }
        catch (Exception e) { /*
            error handling */ }
        return returnScope.toJson();
    }
    ...
}
```

Figure 8: Part 1 of the generated code for the example.

9 to 13 in Figure 8) handles the configuration given in the D°-Code. The whole class is derived from the `HttpDataApp` class in order to minimize the generated code base and increse the maintainability of different Data App types. The `process`-method (line 17 to 26 in Figure 8) is the entrypoint for the execution of the Data App's logic. At first, the execution is prepared by parsing and validating inputs, preparing the output, and setting up variable management. This can be seen in lines 18 to 22 of Figure 8. For each D°-block (denoted by the keywords `begin` and `end`) a separate function is generated. The method for

```java
private void mainBlock(/* ... */) {
    VariableManager variableManager
        = new VariableManager(
        parentVarManager);
    variableManager.registerVariable
        ("var_a");
    Instance var_a = TYPE_TAXONOMY.
        create(Identifier.of("core.
        Text"));
    variableManager.
        initializeVariable("var_a",
        var_a);
    InputScope in_a = new InputScope
        ();
    /* fill input scope with values
        */
    OutputScope out_a = SANDBOX.
        callActivity((
        ActivityInstance)
        ACTIVITY_INSTANCE_REGISTRY.
        read("ProtectedBackupFile"),
         in_a));
    if (out_a == null) {
        throw new
            DegreePolicyValidation
               Exception(/* ... */);
    }
    variableManager.updateVariable("
        var_b", out_a.get("result"))
        ;
    returnScope.add("result",
        variableManager.readVariable
        ("var_b"));
    return;
}
```

Figure 9: Part 2 of the generated code for the example.

the only block of the example is called in line 23 of Figure 8 and shown in Figure 9.

The `mainBlock`-method contains the actual logic of the Data App. Lines 3 to 5 of Figure 9 correspond to the declaration of the variable in line 9 of Figure 1. Lines 6 to 12 in Figure 9 are the activity call from line 10 in Figure 1. At first, the input variables are prepared. Next, the activity call is delegated to the `Sandbox`. The last step is the storing of the return-value(s) in the variable manager. The return statement in line 11 of Figure 1 shows the return value of the Data App which corresponds to line 13 in Figure 9.

The presented generated code does not contain any policy specific code beside the handling of failed policy validation in lines 10 and 11 of Figure 9. To find actual policy relevant code, we have to look at the implementation of `Sandbox.callActivity(...)` which can be found in Figure 10. The code shows that it is ensured that pre- and postconditions are checked before and respectevly after each activity execution. The validation of security manager interventions can-

```
public OutputScope callActivity(/*
    ... */) {
    if (!validatePrecondition(
        activity.getPolicies())) {
        return null;
    }
    OutputScope output = activity.
        run(input);
    if (!validatePostcondition(
        activity.getPolicies())) {
        return null;
    }
    return output;
}
```

Figure 10: Implementation of Sandbox.callActivity(...).

not be seen in this code because it is a result of the instrumented APIs of the JDK.

5 CONCLUSION

We have introduced the policy system, which is used in the domain specific language D°. We demonstrated how D° implements the paradigm of policy-agnostic programming and how to ensure that all defined policies are enforced at the right place at time.

Using an example, we presented how D° is used for application development with a prototype using Java as host language. Using the same example, we demonstrated how the different subsystems of the language can be extended in order to meet specific requirements (regarding the domain).

ACKNOWLEDGMENTS

This work was developed in Fraunhofer-Cluster of Excellence “Cognitive Internet Technologies”.

This research was supported by the Excellence Center for Logistics and IT funded by the Fraunhofer-Gesellschaft and the Ministry of Culture and Science of the German State of North Rhine-Westphalia.

REFERENCES

Austin, T. H., Yang, J., Flanagan, C., and Solar-Lezama, A. (2013). Faceted execution of policy-agnostic programs. In *Proceedings of the Eighth ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 15–26.

Barthe, G., Naumann, D. A., and Rezk, T. (2006). Deriving an information flow checker and certifying compiler for Java. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 229–242.

Eitel, A., Jung, C., Kühnle, C., Bruckner, F., Brost, G., Birnstill, P., Nagel, R., and Bader, S. (2019). Usage Control in International Data Spaces: Version 2.0.

Jung, C., Eitel, A., and Schwarz, R. (2014). Enhancing Cloud Security with Context-aware Usage Control Policies. In *GI-Jahrestagung*, pages 211–222.

Katt, B., Zhang, X., and Breu, R. (2008). A general obligation model and continuity: enhanced policy enforcement engine for usage control. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 123–132.

Lazouski, A., Martinelli, F., and Mori, P. (2010). Usage control in computer security: A survey. *Computer Science Review*, 4(2):81–99.

Myers, A. C. (1999). JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241.

Otto, B. and Jarke, M. (2019). Designing a multi-sided data platform: findings from the International Data Spaces case. *Electronic Markets*, 29(4):561–580.

Park, J. (2003). *Usage control: A unified framework for next generation access control*. Dissertation, George Mason University, Virginia.

Park, J. and Sandhu, R. S. (2004). The UCON ABC usage control model. *ACM Transactions on Information and System Security (TISSEC)*, 7(1):128–174.

Park, J., Zhang, X., and Sandhu, R. S. (2004). Attribute mutability in usage control. In *Research Directions in Data and Applications Security XVIII*, pages 15–29.

Polikarpova, N., Yang, J., Itzhaky, S., Hance, T., and Solar-Lezama, A. (2018). Enforcing information flow policies with type-targeted program synthesis. In *Proceedings of the ACM on Programming Languages*, volume 1.

Sandhu, R. S. and Park, J. (2003). Usage control: A vision for next generation access control. In *International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security*, pages 17–31.

Schuette, J. and Brost, G. S. (2018). LUCON: data flow control for message-based IoT systems. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 289–299.

Yang, J. (2015). *Preventing information leaks with policy-agnostic programming*. Dissertation, Massachusetts Institute of Technology, Massachusett.

Yang, J., Yessenov, K., and Solar-Lezama, A. (2012). A language for automatically enforcing privacy policies. *ACM SIGPLAN Notices*, 47(1):85–96.

Zdancewic, S. (2004). Challenges for information-flow security. In *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence (PLID'04)*, pages 6–11.

Zolnowski, A., Christiansen, T., and Gudat, J. (2016). Business model transformation patterns of data-driven innovations.