Automatically Identifying Parameter Constraints for Complex Web APIs: A Case Study at Adyen

Version of August 20, 2020

H.A. Grent

Automatically Identifying Parameter Constraints for Complex Web APIs: A Case Study at Adyen

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

H.A. Grent
born in Hoorn, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

Adyen
Simon Carmiggeltstraat 6-50,
1011 DJ
Amsterdam, the Netherlands
`www.adyen.com`

Automatically Identifying Parameter Constraints for Complex Web APIs: A Case Study at Adyen

Author: H.A. Grent
Student id: 4440528

Abstract

Web APIs can have constraints on parameters, such that not all parameters are either always required or always optional. Sometimes the presence or value of one parameter could cause another parameter to be required. Additionally, parameters could have restrictions on what kinds of values are valid. We refer to these as inter-parameter and single-parameter constraints respectively. Having a clear overview of the constraints can help API consumers to integrate without the need for additional support and with fewer integration faults.

We developed two approaches for identifying parameter constraints in complex web APIs. One approach uses online documentation to infer inter-parameter constraints, the other depends on static code analysis to extract inter- and single-parameter constraints from the control flow of the API's source code. In our case study at several APIs at Adyen, the documentation- and code-based approach can identify 21% and 53% percent of the constraints respectively. When the constraints identified by both approaches are combined, 66% of the inter-parameter constraints can be identified. Code analysis is able to identify 78% of the single-parameter constraints.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. M. Aniche, Faculty EEMCS, TU Delft
Company supervisor:	A. Akimov, Head of API, Adyen
Committee Member:	Dr. C.B. Poulsen, Faculty EEMCS, TU Delft

Preface

First and foremost, a large thank you to Maurício Aniche and Aleksei Akimov for their guidance and continued involvement. I would also like to thank the people from the Documentation and Developer Experience team for their company and their ideas. Coffee tastes better in good company. Additional thanks to Casper Schröder and Hendrig Sellik for making public transport more fun.

H.A. Grent
Delft, the Netherlands
August 20, 2020

Contents

Chapter 1

Introduction

Web Application Programming Interfaces (Web APIs) allow applications to access the functionality or data of a service through HTTP requests. Web APIs commonly provide an API reference [15], which describes what operations are available through which endpoint and which parameters are required or optional for requests to these endpoints. However, these parameters are not always just required or optional: whether they are required can depend on the presence or value of another parameter [21, 17]. As such, there are constraints between parameters.

Within Adyen, as a payment platform, we find such inter-parameter constraints as well. Take constraints that apply on different payment methods as example; if an API consumer wants to make a payment with iDEAL, then the previously non-required *issuer* and *returnURL* parameter are now required[1]. For other payment methods different parameters become required. As another example, when authorizing a payment, the API expects a bank account or a card as payment details[2]. Without either the request will fail.

Having a clear overview of the constraints in a Web API is important, because it helps API consumers to integrate the API without the need for company support. Incomplete or incorrect documentation on these constraints can waste a lot of time, and cause costly integration faults [3]. Currently, these constraints are documented and maintained manually by API developers, which can be laborious and difficult. This difficulty comes from the size and complexity of the code base of the web service, and documentation being provided by different people than those who write the code. Therefore, tools that help API developers identify and maintain the constraints in their APIs are needed.

In this paper, we evaluate two approaches to automatically identify con-

[1]`https://docs.adyen.com/api-explorer/#/PaymentSetupAndVerificationService/v52/post/payments__reqParam_paymentMethod`

[2]`https://docs.adyen.com/api-explorer/#/Payment/v52/post/authorise__reqParam_bankAccount`

straints for complex Web APIs. One approach analyses the online service documentation, the other analyses the source code. For these approaches, we identify challenges that can complicate the process of automatically identifying constraints. We draw inspiration from Wu et al. [35] who set out to identify inter-parameter constraints from the online API reference and available software development kits (SDKs).

Our approaches are developed to be able to identify constraints for complex, large scale APIs. We consider complexity mainly as the number of parameters an API exposes, and the complexity within the source code itself. For our main APIs under study, the Adyen APIs, complexity largely results from making a large number of payment-related operations available through one interface while being subject to varying legislation or payment method requirements. When examining the Adyen API through the online documentation[3] we can observe several endpoints, with a varying number of parameters. For example, version 52 of the '/payments' endpoint, which exposes a large number of payment methods, features 55 top-level parameters and 371 parameters in total. Other endpoints often expose considerably fewer parameters.

The complexity of an API poses two key challenges. For one, because there are more parameters, exploring the solution space with any search-based technique is made more difficult. An inter-parameter constraint may be between any two (or more) parameters, which means there is an exponential amount of possible constraints. Second, how the code holding the constraints is structured is changed to deal with this complexity accordingly. One could expect a more frequent usage of frameworks, abstraction, and interfaces.

Main Research Question: Can (inter-)parameter constraints in complex Web APIs be identified using automated techniques?

This is split into the following research questions (RQs):
RQ1: How effective are documentation- and static code analysis in identifying parameter constraints in a large-scale enterprise API?
RQ2: What are challenges in using documentation- or static code analysis to identify inter-parameter constraints?

Our results show that static code analysis is more effective than the documentation based approach for identifying parameter constraints. The documentation and code-based approach can identify 21% and 53% percent of the inter-parameter constraints respectively. When the constraints identified by both ap-

[3] `https://docs.adyen.com`

proaches are combined, 66% of the inter-parameter constraints can be identified. Code analysis is able to identify 78% of the single-parameter constraints.

The two approaches face largely separate challenges. The documentation based approach largely suffers from a lack of available explicit information describing the constraints. Static code analysis tends to be able to extract constraints from the source code by maintaining a basic variable stack, evaluating method calls, and analyzing conditions in for-loops, switch statements, and if-else statements. The main challenges we faced within this case study were data flow insensitivity and engineering a sound static code analysis approach.

The main contributions of this research are:

- An approach that infers inter-parameter constraints from online documentation.
- An approach that extracts inter-parameter and single-parameter constraints from source code.
- For both approaches: an overview of challenges of automatically identifying constraints in a complex Web API.

Chapter 2

Related Work

In this chapter, we provide insight into the literature surrounding parameter constraints and its broader surrounding field of API quality. We look at existing approaches related to identifying parameter constraints and what inter-parameter constraints look like in practice.

2.1 API Quality and Usability

There has been an increase in API research [25]. This literature expands on both the technical and soft aspects of API quality attributes. Technical aspects focus on things such as architectural styles [18, 31], specification formats (RAML[1], OAS[2]) and software quality requirements [5, 12]. Soft aspects focus on API usability [20, 26, 28, 10]. The soft and technical aspects are not a strict dichotomy. While APIs in its broadest sense refer to a generic kind of programming interface, our focus is Web APIs: web services over HTTP.

Usability, as an aspect of software quality, frequently takes focus in API design literature, because ultimately APIs are consumed by people to create specific functionality for their own use case. Usability in API design is related to technical aspects of API design by several design aspects such as the overall architecture and organization of classes [34, 4], which in turn impact compatibility [33] and maintainability. Any of such decisions can, and probably will, have an impact on the overall usability of an API. Such is the relationship between the technical- and soft aspects of API quality.

For technical aspects that have a more direct connection to usability, API architectural styles are readily discussed. To maintain some overview, we consider RPC, REST, and GraphQL as the main styles [7]. REST is by far the most popular style [27]. Choosing for one style or the other depends on the use case. Without

[1]`https://raml.org`

[2]`https://swagger.io/specification/`

trying to open a can of worms; RPC may be beneficial for action-oriented services, REST for resource-oriented services and GraphQL for highly relational data [7, 32]. At the end of the day, all these impact the way software architects think about their implementation and the way the functionality is exposed to the API's users.

For soft aspects, API usability is not strictly defined in literature. Within this topic there is a focus on learnability, efficiency and understandability [25, 12]. Rauf et al. [25] describes learnability as the capability of software to be learned by its developers with ease, efficiency in terms of resources (time) needed to complete a task, and understandability as to how well a user can understand the code without confusion. Various works exist on identifying usability factors and sometimes provide metrics for them, with little consistency in the naming of the metric [25]. Readability, satisfaction, and memorability are common metrics [25]. A lack of documentation is a key obstacle for API learnability [26, 23]. Robillard and DeLine [26] identify five documentation factors impacting the developers learning experience: documentation of intent, code examples, mapping usage scenarios, penetrability, and format and presentation.

For developers, API usability is key in the adoption/integration process. Learning obstacles may result in opting for a different service [24, 26] or increased integration efforts in supporting API consumers. Research suggests that a significant portion of faults in API integration can be attributed to invalid or missing user input [3]. These integration faults relate to parameter constraints, such as the absence of (conditionally) required parameters or invalid values for provided parameters.

2.2 Constraints in Practice

The work by Martin-Lopez et al. [17] gives an overview of the frequency of inter-parameter constraints for different industries, considering only REST APIs. According to their research, 85% of the REST APIs have inter-parameter constraints and on average 9.8% of the operations contain constraints. In less expansive studies, Oostvogels [21] and Wu et al. [35] report comparable numbers.

With respect to the different kinds of dependencies, both Martin-Lopez et al., Oostvogels and Wu et al. [17, 21, 35] present data, while categorizing them differently. Most of the constraints in the wild are not complex, only 4% of the dependencies in REST APIs are classified as complex [17]. Complex constraints involve multiple categories from the categories below. For non-complex constraints, all three categories described by Oostvogels are roughly equally common.

Oostvogels describes three categories of constraints:

- **Exclusive:** Exactly one of a set of parameters must be present.
- **Dependent:** The presence or value of one parameter depends on the presence or value of another parameter.
- **Group:** A group of parameters should either all be present or not present.

Whereas the different categories are arguably subsets of the *dependent* category in terms of logical equivalence, it is useful to have a semantic difference between them. We consider the following categories and subcategories:

- **Dependent**
  P1 $\rightarrow$ P2: The presence of P2 depends on the presence of P1.
  P1 = V $\rightarrow$ P2: The presence of P2 depends on the value of P1.
  P1 = V $\rightarrow$ P2 = V2 The value of P2 depends on the value of P1.
- **Exclusive**
  P1 or P2: P1 or P2 is needed, providing both is valid.
  P1 xor P2: P1 or P2 is needed, providing both is invalid.
- **Group**
  P1 <-> P2: Provide both or neither.
- **Arithmetic**[3]
  P1 > P2: P1 has to be greater than P2.
  P1 > P2 - X: P1 has to be greater than P2 minus some constant.
  P1 + P2 + P... = Pn: A sum of parameters has to be equal to the value of another parameter.

The term '(inter-)parameter constraints', as used by Oostvogels [21], is not used consistently as such throughout literature, the concept is referenced to differently in other works. Some other terms used to reference the same concept are '(inter-)parameter dependencies' [17], 'dependency constraints' [35], 'data contracts'[11] or more generally 'method specifications' [22].

We will use the term '(inter-)parameter constraints', with the following working definitions: an inter-parameter constraint describes a requirement on the presence or value of a parameter based on the presence or value of another parameter, for a given web service operation [35]. Single-parameter constraints describe the requirement on the presence or value of single parameter. We consider something a requirement if not adhering to it leads to the request failing, see Section 3.1.2. As a result, any constraints that lead to a different result than intended with a request are not strictly parameter constraints.

[3]The arithmetic constraint's subcategories do not form a complete list. We opted to describe the most common arithmetic constraints.

2.3 Machine-Readable Representation

Oostvogels [21] provides a machine-readable language for expressing parameter constraints. This constraint-centric specification language supports expressing inter-parameter constraints as well as single-parameter constraints. The syntax proposed by Oostvogels, can be seen in Figure 2.1. We use this language to express constraints, with minor differences in representation. For example, provided the constraint that 'if A is provided, then B is required' we would represent this as $A \rightarrow B$ as opposed to $implic(present(A), present(B))$.

$$\begin{aligned}
\mathsf{v} \in \text{Values} &::= \text{Number, String, Boolean or Parameter} \\
\mathsf{f} \in \text{Parameters} &::= \mathsf{s} \mid \mathsf{f.s} \mid \mathsf{f.[]} \\
\mathsf{t} \in \text{Types} &::= \mathsf{string} \mid \mathsf{number} \mid \mathsf{boolean} \mid \mathsf{object} \mid \mathsf{t[]} \mid \mathsf{null} \\
\mathsf{cd} \in \text{Constraint definitions} &::= \mathsf{s}(\mathsf{s}_1, ..., \mathsf{s}_n) = \mathsf{c} \\
\mathsf{c} \in \text{Constraint} &::= \mathsf{o} \mid \mathsf{lc} \mid \mathsf{s}(\mathsf{v}_1, ..., \mathsf{v}_n) \\
\mathsf{o} \in \text{Operations} &::= \mathsf{present(f)} \mid \mathsf{type(f)}{=}\mathsf{t} \mid \mathsf{length(f)} \oplus \mathsf{v} \mid \mathsf{value(f)} \oplus \mathsf{v} \\
\mathsf{lc} \in \text{Logical connectives} &::= \mathsf{and(c, c)} \mid \mathsf{or(c, c)} \mid \mathsf{not(c)} \mid \mathsf{implic(c, c)} \mid \mathsf{iff(c, c)} \\
\oplus \in \text{Math operators} &::= =, !=, <, >, <=, >=
\end{aligned}$$

Figure 2.1: Machine-readable constraint language syntax from Oostvogels [21].

2.4 Automatically Identifying Parameter Constraints

A handful of papers exist outlining approaches for automatically identifying single and inter-parameter constraints. These approaches rely on documentation, API responses, or code analysis to infer such constraints for simple APIs.

Gao et al. [11] uses a decision tree based approach to infer inter-parameter constraints. The information for populating the decision tree is inferred from observing API responses for a given candidate constraint. These candidates are chosen using a set of heuristics and by observing the API's feedback. The latter includes parsing error messages provided as feedback by the API. Whereas the approach was able to infer 145 out of 154 manually identified constraints, few APIs were evaluated. The APIs under study did contain at around 5 parameters per endpoint on average.

Pandita et al. [22] utilizes a number of sources of documentation, including in-code comments, to infer constraints using a NLP based pipeline. These constraints are both inter-parameter constraints as well as single parameter constraints. A large part of the pipeline is responsible for transforming natural text to formal contracts. The constraints are not automatically validated for correctness. This approach yields an average of 92% precision and 93% recall on a number of Facebook Web APIs and .NET libraries.

The work by Atlidakis et al. [2] uses a fuzzing type approach to find dependencies between parameters for different endpoints. That is, it aims at identifying dependencies between a parameter in endpoint A and another endpoint B. To steer the fuzzing process, OpenAPI specifications and the feedback from API responses are used. The fuzzing approach fires a larger number of API requests, at around 5000.

These approaches are designed to infer constraints from documentation, but do not consider code as input. Wu et al. [35] sets out to automatically identify inter-parameter constraints by inferring constraint candidates from the online API reference and available software development kits (SDKs). These candidates are then verified by calling the public web service with request bodies which would satisfy or violate the candidate constraints. The approach by Wu et al. uses a combination of NLP and data flow analysis, for documentation and SDK analysis respectively. We will next dive into details of their approach, as we use it as inspiration for the approach within this paper.

For analyzing the documentation they use a two-phase filtering approach, called the 'loose' and 'rigid' strategy. The rigid strategy uses predefined templates to match with the text. For example, the template 'Either A or B must be specified' is matched with a concrete sentence using the Jaccard distance. The loose strategy "generates a constraint candidate when the number of distinct parameters appearing in its describing sentence matches the number of parameters in a template."[35]. If two parameters occur in the same sentence, then they are assumed to be related in some way. The loose selection is used as the preferred strategy throughout the paper.

For analyzing the API's SDK, they perform data-flow analysis on control flow graphs generated from SDK methods. In this process all possible combinations of sets of parameters are generated, which are then used to infer constraints. The process for generating these combinations depends on branches in the control flow graph of a single method and the combinations of parameters accessed in those branches. A literal or variable in the code is included in the data flow when it corresponds with a parameter name available in the online documentation. The method for inferring constraints from the combinations is not explained.

The approach achieves a precision and recall of around 95% for four simple APIs. Whereas the approach relies on both code and documentation by design, the results indicate that by far most of the inter-parameter constraints are inferred from the documentation. The documentation provided a total of 351 candidates and the code a total of 36 candidates. The documentation based candidates did have a lower precision than the SDK based candidates, at 20.8% and 100.0% respectively, but opposite being true for recall at 82.9% and 40.9% respectively.

Chapter 3

Approach

Our approach has two distinct types of analysis: documentation analysis and code analysis. The goal of both approaches is to automatically obtain constraints of Web APIs in a machine-readable representation. The code analysis extracts single-parameter constraints (e.g. $X > 5$) and inter-parameter constraints (e.g. X or Y), the documentation analysis only infers inter-parameter constraints. Our approach draws inspiration from Wu et al. [35], with the aim to be able to apply our approach on more complex systems being used in the industry. A high-level overview of our process is shown in Figure 3.1. We shortly describe the general process, and later describe the documentation and code analysis in more detail.

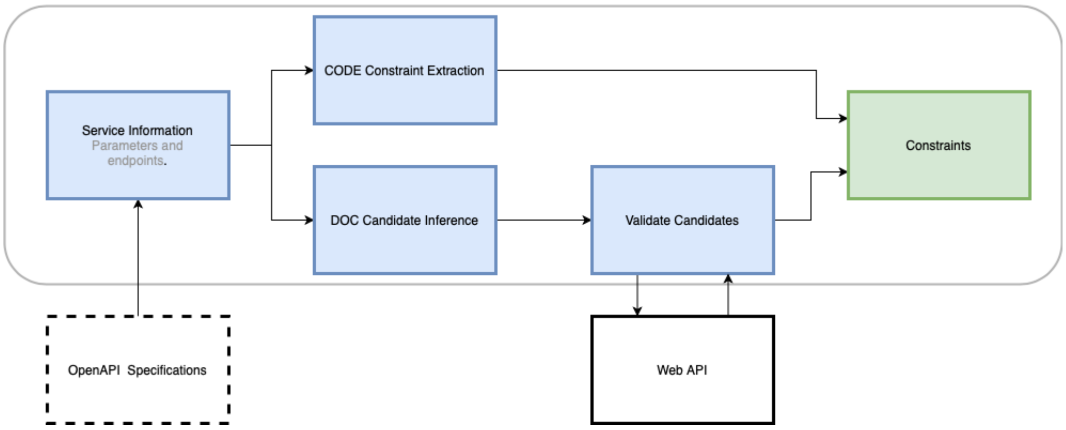

Figure 3.1: Overview of the process, showing the initial service information collection step and then the documentation- and code analysis approach.

In the first step, we collect information about the parameters for the endpoints of a Web API from the OpenAPI Specification[1] (OAS). The most important service information is: the data type of each parameter, whether the parameter is required, any enum values, their description, and parent- and sub-parameters. This information aids us with a number of tasks, such as default value generation for making requests to an API and detecting parameter references. We use this information in both approaches.

For the documentation based approach, we analyze the textual documentation to infer constraints. This process has two steps. First, we extract sets of candidate parameters from the OAS descriptions, e.g., the candidate [*bankAccount, card*]. We will explain how we obtain these candidates in Section 3.1. The candidates are collected to reduce the total number of requests needed to infer the constraints. Since a constraint may be between any two or more parameters, we would have to inspect all combinations for constraints. Trying all combinations would require an exponential number of requests with respect to the amount of parameters, which would require too many requests.

Secondly, we validate the candidates that were collected in the previous step. The candidates only show us which parameters could have a constraint between them. With validation we aim to infer the exact constraint that applies, e.g., given the candidate [*bankAccount, card*] we want to infer that the inter-parameter constraint is $or(bankAccount, card)$. We rely on sending requests to the API to infer the constraint. Whether a request fails or succeeds tells us whether it satisfied the requirements of an API or not.

For the code approach, we look at the control structure of methods within the source code to extract constraints. We aim to infer the usage of parameters within this control structure and the preconditions that apply to their usage. For example, $if(X! = null)\{Y\}$ would allow us to infer that Y is needed with the precondition that X is provided in the request.

3.1 Documentation Analysis

With documentation analysis, we aim to infer if there are constraints between parameters by analyzing the textual documentation of the Web API. Documentation analysis has two distinct steps: finding sets of parameters which might have constraints between them (candidates) and then determining the exact inter-parameter constraints by means of sending API requests to the subject API (validation).

[1]The OAS is an API description standard which provides service information in a structured way, typically using the JSON format. The Swagger Editor[2] gives a quick impression of the format. For this work, it is good to be familiar with the possible data types for parameters [3]. These data types are currently defined as *string, number, integer, boolean, array,* and *object.*

3.1.1 Candidate Inference

We use the OAS' parameter descriptions to find candidates. Adyen's API Explorer[4] visualizes these parameters, along with their descriptions, for all public endpoints. The intuition is that the description of one parameter can refer to another parameter, which hints at a possible constraint between the two parameters. E.g. given the parameter *bankAccount* with the description "The details of the bank account. Either *bankAccount* or *card* is required.", we assume the two can have a constraint between them.

To extract candidates, we use a co-occurrence matrix [6]. This co-occurrence matrix contains a row and column for every parameter in a given endpoint. To populate this co-occurrence matrix, we automatically analyze the description of every parameter; if the description of a parameter contains the name of another parameter, then their corresponding entry in the matrix is updated. E.g. for the earlier example the cell corresponding with *bankAccount* and *card* will be updated by one.

Whether a parameter is required may depend on the value of another parameter. E.g. $paymentMethod = \text{"}iDEAL\text{"} \rightarrow returnUrl$. Such value-dependent constraints require additional information to be inferred in the validation step. More specifically, we need to know which values are relevant for what parameters. We do this by checking if the descriptions mention any of the enum values the OAS provides. When an enum value of a parameter is mentioned in a description, then this value is marked and used in the subsequent validation step.

	card	**bankAccount**	**type**	**reference**
card	-	2	0	X
bankAccount	-	-	1	X
type	-	-	-	X
reference	-	-	-	-

Table 3.1: Example co-occurrence matrix for an endpoint with four parameters. Any X entry occurred too frequently, and is ignored.

Certain parameters may occur extraordinarily often in descriptions. This is often because of parameter names being common as a word in natural text. Words such as 'reference' and 'value' tend to be used without it being a reference to a parameter. This would yield us a lot of irrelevant candidates. Consequently, we ignore parameters that co-occur with too many other parameters. Following Table 3.1 this is indicated with an X.

[4]`https://docs.adyen.com/api-explorer/#/PaymentSetupAndVerificationService/v52/post/payments`

We generate sets of candidates based on the co-occurrence matrix. We generate candidates from every row of the matrix. For each row we include the parameter corresponding to that row in the candidate, and any parameters that co-occurred at least once with that parameter. The candidates with only one parameter are removed. Once again following Table 3.1, we would generate the sets [card, bankAccount], and [bankAccount, type]. After this, we use these candidates as input to the validation phase.

Parsing Semantics

The documentation also provides more specific information about the exact constraint itself. One might find descriptions such as 'either x or y' is required. We do not consider these semantics in the inference of constraints, due to the difficulties involved in parsing such semantics accurately, which results in a loss of accuracy. Not all descriptions for parameters are necessarily accurate or semantically clear either. In the case of 'either A or B' it is not clear whether it is used exclusively or inclusively [5]. Not making any of such assumptions means that all combinations have to be tried. Which is doable, given the small size of the candidates.

We explored the usage of word embeddings [19] as a means to find candidates. When using word embeddings, we can obtain a semantic similarity between words, and sentences. The premise is that parameters with semantically similar descriptions are more likely to be related to each other in a constraint. In practice, this kind of similarity measure did not filter the number of possible parameter combinations enough to be viable on its own. The details of our approach and findings can be found in appendix B.

3.1.2 Validating Candidates

From the documentation analysis we get sets of parameters which might have constraints between them (candidates), and for each parameter which values were found in the documentation. The aim is to figure out the exact inter-parameter constraint that applies on these parameters, if any. We do this by generating requests, and observing the API response for failure (see Section: 3.1.2). If a request fails, this tells us that some constraint was not satisfied.

We generate a truth table for each candidate. In this truth table each row indicates the present, or absent parameters and whether the corresponding request's result was successful. We represent all the possible combinations of parameters in such a truth table. An example of such a group is $[card, bank]$, for

[5] `https://english.stackexchange.com/questions/13889/does-either-a-or-b-preclude-both-a-and-b`

which the corresponding truth table can be seen in Table 3.2. For each row in this table a base request is generated, with the parameters indicated as present included and the parameters indicated as not present removed. This request is then sent to the API and the response is checked for failure. If the request fails, then the Result column is updated accordingly.

Following Table 3.2, either of constraints extracted from the successful rows need to be true in order to satisfy the constraint. The disjunctive normal form between the successful rows forms the constraint that applies between *bank* and *card*. If all requests are successful, then no constraint applies.

For candidates which contain parameters with marked values this process works slightly differently. For a parameter with marked values, all the values identified in the previous step are added as a column in the truth table. Besides the marked values, a column including a non identified value is included. This is to have greater confidence that the constraint actually depends on the value of the parameter, and not simply on the presence of the parameter. An example of this process can be seen in Table 3.3. In which case *state* had *gas* as an identified enum value.

card	**bank**	**Result**	**Constraint**
0	0	F	-
0	1	T	and(!card, bank)
1	0	T	and(card, !bank)
1	1	T	and(card, bank)

Table 3.2: Validation table for two parameters, where 0 and 1 indicate the absence and presence of parameters. The result column indicates whether a request was successful (T) or not (F).

state = gas	**state = solid**	**temperature**	**Result**
0	0	0	F
0	0	1	T
0	1	0	F
0	1	1	T
1	0	0	T
1	0	1	T

Table 3.3: Validation table for two parameters, state can be 'gas' or 'solid'. Rows with both 'state' columns being true are excluded.

Request Building

Building valid requests is a major part of validating the previously generated candidates. To do this, request bodies are built by modifying the *base request* according to the modifications imposed by a truth table. Following such a truth table, the parameters indicated as present are included and the parameters indicated as not present removed from the base request. For example, given that the base request includes the parameter *accountName*, then on the basis of Table 3.1 we would generate the four request bodies shown in Figure 3.2.

```
{
 "accountName": "Medina"
}
```

```
{
 "accountName": "Medina",
 "bankAcount": "DE8098"
}
```

```
{
 "accountName": "Medina",
 "card": "12356",
}
```

```
{
 "accountName": "Medina",
 "card": "12356",
 "bankAcount": "DE8098"
}
```

Figure 3.2: Example request bodies generated on the basis of Table 3.1.

Generating the Base Request

The base request is a default request specified for each endpoint, which should always succeed. These default requests can either be specified manually, or they can be generated from the OAS. The OAS specifies the required parameters, including all should result in a valid base request. When this was not the case, we manually added the missing parameters to the base request.

Generating Parameter Values

The parameters provided in a request need to have valid values. What qualifies as valid depends on what values are meaningful for the given parameter. For example, if a parameter represents a date providing any value which is not a date makes little sense. We use either a manually defined value or default value. The default value depends on the type of the parameter.

The type of the parameter can be inferred from the OAS, which are currently defined as *string*, *number*, *integer*, *boolean*, *array* and *object*[6]. For each of these types a standard value can be configured. A string may by default return ’str’ and an integer may return ’0’. For some parameters, such a default value may not be sufficient, in which case one has to manually define a standard value.

[6] `https://swagger.io/docs/specification/data-models/data-types/`

This would apply on parameters such as the previously mentioned date example, card numbers, and account names.

Identifying Request Failure

Whether a request was successful or not is determined primarily by the HTTP status code returned as a response to a request. Generally, 2xx is considered as a success and 4xx and 5xx as a failure. The specific codes and responses depend on the subject API. An API may respond with a '200: OK' while providing an empty response body, depending on whether this is a normal response such responses may be added to the definition of failure as well. I.e. the definition of failure may have to be adjusted for a specific domain. Requests may also fail due to connectivity issues. For us, when running the API on a local server this was not a problem.

Rate-limiting

Web services may limit the number of requests that are made in a given period of time. Rate-limiting is not a limitation in our validation process, as we have access to a local build of the API which can function the same as a deployed public Web API. We are able to make as many requests as we want, while only limited by the time it takes the local build to process the request. If a local build is not possible, then access to a web service account without such rate limiting is an alternative.

Computational Costs

The total time needed to validate constraints depends on a number of factors, most importantly the number of candidates, the number of parameters for a candidate, and the number of requests we can make each second. For each candidate we need to check all combinations of parameters. The number of combinations, assuming that each parameter is either present or absent is 2^p, where p is the number of parameters. We are able to make at around 5 requests a second. The runtime needed in seconds equals $\sum_{i=1}^{C} 2^{|P_{C_i}|}/5$, where C denotes the set of candidates and $|P_{C_i}|$ is the number of parameters in candidate i.

Given a parameter can be multiple values this calculation changes. The number of combinations is given by $\prod_{i=1}^{p} |values(p_i)| + 1$. Following Table 3.3, knowing that gas can have 2 values and temperature just 1, we make 6 requests. Substituting this term in place of $|P_{C_i}|^2$ would give us the expected runtime for candidates which consider specific values.

Provided this information, we can see that the number of parameters in candidates will dominate the runtime rather quickly. However, in practice candidates tend to have a small number of parameters, thus controlling the number of candidates tends to be the priority.

3.2 Code Analysis

With code analysis, we aim to extract constraints from the control structure of the source code. For this we analyze methods relevant to handling the HTTP requests made to the API. A method called the ’controller method’ is typically responsible for handling requests made to one endpoint of an API. Starting from this controller method, we detect the access of parameters and analyze any control structures and method calls parameters are used in. Within our case study, the Web APIs primarily use Java. As such, the control structures mostly include if-else statements, switch-statements, and for-loops.

Following Snippet 1, we can see how we could infer the dependency of the ’card’ on ’bankAccount’ and the constraint on the value of ’offset’. That is, if the card is not provided for payment details, then we would need the bankAccount from the request. For the offset, we know that it should be smaller or equal to 80. In practice, there is a large number of challenges involved in inferring such dependencies, but with this example we establish a basic intuition.

```
def handle(Request req)
    if req.getCard() != null then
        method = req.getCard()
        validateCard(method)
    else
        method =
          req.getBankAccount()
    end
    if req.getOffset() > 80 then
        throw Exception()
    end
    ⋮
    method.preprocess()
```

Snippet 1: An example method handling an API request.

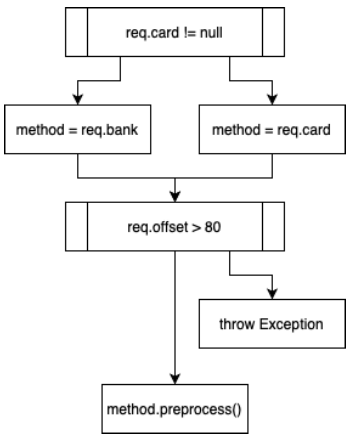

Figure 3.3: Control-flow graph corresponding with the method shown in Snippet 1.

3.2.1 Control Flow Graph

To represent the control structure of a method, we use a control-flow graph (CFG). We generate such CFGs for every method we analyze. The CFG shows what branches can be taken, and as such it can be used to know what parameters are used within those branches and which preconditions apply for those branches. Since the CFG tells us what branches lead to invalid states, such as throwing exceptions, we can infer what preconditions would cause the request to be invalid. For Snippet 1 the CFG shown in Figure 3.3 is generated.

We collect constraints by iterating over the statements of the CFG. In this process we collect a tree of preconditions and consequences to represent the constraints. As an example of such a tree, following Snippet 1 we would obtain the constraints shown in Figure 3.4. The exact preconditions that apply can be inferred from the CFG. For example, from the CFG in Figure 3.3, the precondition $req.card! = null$ has a *true* and a *false* branch. From this we can infer that the negation of this precondition would lead us to the statement $method = req.card$. Within the constraint tree this is represented as $card = null \rightarrow \{present(bankAccount)\}$.

Looping expressions, such as the for-loop, can be difficult to analyze statically. This is because the condition breaking the loop can be complex. However, we noticed that the exact analysis of looping expressions was not important for the inference of constraints. Looping statements were sometimes used for parameters which have an array value. E.g. "people": $[\{"name" : "Frank", ...\}, ...]$. For such array values, any conditions within the body of a loop would apply to all values that would be iterated over. Hence, analyzing the body of a for-loop once would be sufficient.

```
card != null →{present(card) },
card = null →{present(bankAccount) },
offset > 80→{InvalidState }
```

Figure 3.4: An example constraint tree, based on Snippet 1

3.2.2 Sensitivities

We perform analysis which is flow-sensitive, partially path-sensitive and context-sensitive. In the analysis the branches of the control flow graph (CFG) are considered, without explicitly taking the previously evaluated path into account. This makes the analysis only partially path-sensitive. To exemplify this, consider a node C reachable through either A or B. When evaluating C the program is agnostic to whether the execution trace would have gone through A or B. If a variable is modified in two exclusive branches, then the most recent modification is chosen. Similarly, we do also not keep track of data conditions that would result from taking one path or the other. For example, if a branch has $offset > 80$ as a guard then we do not assume anything about the value of *offset* outside the branch's body.

3.2.3 Inter-Procedural Analysis

The controller method calls other methods, including these calls in the analysis is important to find all constraints. Any method the controller method relies on can also make calls to other methods. To represent this network of function calls, we generate a static call graph with the controller method as the root. Starting from the controller method, we recursively construct this call graph up until a predefined depth. For us a depth of 15 was sufficient.

When we occur a function call in the control flow, the body of the called method is evaluated given its current context. Following Snippet 1, the $validateCard(method)$ is an example of such a call, in which $method$ represents its context. After analyzing the body of the method, the constraint tree is integrated with the constraint tree of the controller method. Assume that for the call $validateCard(method)$ we would obtain the tree $card.cvc = null \rightarrow InvalidState$, then this would be added to the original tree as shown in Figure 3.4 and result in the new tree shown in Figure 3.5. Note that the obtained tree is not added at a random level, but in the body of the statement that preceded the method call. I.e. in the body of card != null.

```
card != null →{
  present(card),
  cvc = null →{InvalidState }},
card = null →{present(bank) },
offset > 80→{InvalidState }
```

Figure 3.5: An example constraint tree, based on Snippet 1 with the constraints obtained from analyzing $validateCard(method)$ integrated.

3.2.4 Guard Analysis

Up until now we have only considered simple conditions as guards, however in practice these can get more complicated. Evaluating and parsing them correctly is essential for correct constraint inference, since they form the preconditions of any constraints we find. For example, for the constraint $card.country = "NL" \rightarrow card.cvc! = null$ the precondition $card.country = "NL"$ could be extracted from a statement such as $if(country.equals("NL"))$.

Programming languages tend to have a large variety in what expressions can be used in guards or other parts of the code. In the next sections we explain how we deal with resolving a number of these expressions and how these are used in parsing guards to a constraint oriented machine-readable format.

Variable Stack

Knowing which variables correspond with which parameter is essential for extracting parameter constraints from the code. When a variable is referenced we want to know if it is related to a parameter, and as such relevant for the constraints we will extract. To maintain such references we maintain a variable stack. Our variable stack keeps track of known concrete values for variables and which parameters correspond with which variable. For example, $int\ maxLength = Constants.maxLength$ would be resolved to the $maxLength$ entry shown in Figure 3.6 and $String\ card = request.getCard()$ would be resolved to its corresponding entry. How we know $request.getCard()$ corresponds with the *card* parameter is explained in Section 3.2.5. For the $isValidCard$ variable the condition is the result from a function call (see Section 3.2.4).

```
maxLength: {
  value: 100,
  condition: null
},
card: {
  value: null,
  condition: param(card)
},
isValidCard: {
   value: null,
   condition: param(card.cvc) != null || param(card.iban) != null
}
```

Figure 3.6: An example variable stack.

For Java primitives, including strings, we evaluate basic operations such as addition and subtraction. E.g. $"en" + "\_US"$ is resolved as $"en\_US"$. For booleans we resolve binary operations only if it can be said that they are surely false or true. E.g. given $A||B$ with $A = true$ we know the expression is true. If such expressions are assigned to a variable, then we update the variable stack accordingly. Any expression that we can not resolve result in the value being equal to null.

For collections, such as arrays, we keep track of the contents of the collection if the contents are primitive or enum values. Given APIs often consume simple types, these basic collections are the most significant for inferring constraints. For example, if the stack keeps track of a list of countries in the *countries* variable and the *country* variable corresponds with the parameter $country$, then later on we could parse the statement $if(countries.contains(country))$ to a meaningful precondition of a constraint[7].

[7]We handle the $.contains(arg)$ method as a *common expression* (see Section 3.2.4).

Function Calls

The guards may depend on the result of a (boolean) function call, such as the function shown in Snippet 2. For example, $if(isValidCard(card))\{...\}$. In order to infer which constraints apply to either a 'true' or 'false' result, we use an adaption to the default approach for analyzing function calls. In this adapted approach, all conditions are collected that would result in the function returning 'true'. For simplicity, we assume functions do not return *null*. Following the method in Snippet 2, 'true' would have $or(card.cvc\,!=null, card.iban\,!=null)$ in its set of preconditions. Then, if a guard is false we know that the negation of the conditions in true apply. Figure 3.6 shows how the results from evaluating such a function could be stored in the variable stack. This is relevant for cases such as $boolean\ validCard = isValidCard(card)$.

```
Function isValidCard(Card card)
    if card.getCvc() != null then
        return true
    end
    if card.getIban() != null then
        return true
    end
    return false
end
```

Snippet 2: A boolean method which indicates whether a card is valid or not.

```
card.cvc != null →{return true },
card.iban != null →{return true }
```

Figure 3.7: An example constraint tree adapted for boolean functions, based on Snippet 2.

To collect conditions that result in the function returning true, we maintain an adapted version of the default constraint tree we build. An example of which can be seen in Figure 3.7. In this adapted version the return statements are added to the tree. This way we know all the conditions that correspond with true. If a return statement returns an expression, then we correspond this expression with true. E.g. if the method in Snippet 2 would just contain $return\ card.getCvc()!=null\,||\,card.getIban()!=null$ then those conditions would be corresponded with true.

Common Expressions

The core Java language includes common methods whose logic is hard to infer using static analysis, but can still be given meaning to individually due to their common nature. In this case we do not use the default parsing process, but map the expression to a manually defined machine-readable output. Examples of these are the *.length()* method for strings and the *.contains(arg)* method for collections. We deal with the *.length()* to be able to infer constraints on the length of string type parameters and the *.equals(arg)* operation can be parsed as a simple equality constraint. We applied the same concept for a handful of common methods used within Adyen.

Guard Parsing

When we encounter a conditional statement, such as an if-statement, in the code, we want to parse its guard such that it is expressed in a machine-readable format. We parse guards as a collection of ANDs and ORs. These connectives follow the machine-readable representation presented in Section 2.3. In the process of parsing these statements the expressions that occur directly in the guard are evaluated, i.e., any referenced variables are retrieved from the variable stack and any expression are resolved as described earlier.

For imagining this process, consider the guard $!isValidCard(card)\,\&$ $card.getIssuer()\,! = merchBank$. Assume that $isValidCard(card)$ is the same method described in Snippet 2. We parse this method as a boolean function call, which provides us with the conditions $and(card.cvc = null, card.iban = null)$. Then, since $card.getIssuer()$ returns *issuer* we know that expression corresponds with the *issuer* parameter (see Section 3.2.5). From the variable stack we can see that *merchBank* is set to "Adyen". Putting all this information together, this guard would be parsed to $and(and(card.cvc = null, card.iban = null), issuer\,! = "Adyen")$.

Unparsed Statements

The parts of conditional statements that can not be parsed to a constraint on a parameter are parsed as unparsed, but still shown in the representation. Since code is (often) written to be legible by humans, this allows us to retain some information the condition might have. Following an earlier example $!isValidCard(card)\,\&\,card.getIssuer()\,! = null$. Assume that we could not resolve the reference to $isValidCard(card)$. The guard would be parsed to $and(!Unparsed(isValidCard(card)), issuer\,! = null)$.

Guard Evaluation

Sometimes we can evaluate whether a guard is guaranteed to be false or true. When the guard is guaranteed to be false, then the body corresponding with this path is not evaluated. This excludes any constraints that would be present within the body. Similarly, if the guard is true, then any connected else/else-if statements are not evaluated.

Whether we can evaluate if a guard is guaranteed to be true or false depends on the conditional statement itself. Consider $if(A||B)$ and given $A = true$ then we know that this guard is always true. For $if(A\&B)$ we know that this guard is always false given $A = false$. With these two idioms we evaluate any larger composite statement consisting out of ANDs and ORs.

The boolean value of an expression can be obtained in a number of ways. In most relevant cases, we know the value because it is set in the framework's configuration. This is particularly relevant since a controller method may be reused using different configurations. This configuration can exclude running certain parts of the code and as such exclude constraints present in that part of the code.

3.2.5 Considerations

In this section we underline a number of challenges that need to be considered in order to make the code analysis work on real-world systems. For the following challenges we provide solutions that may not work for all systems, but were used as working solutions for the results of this paper.

Duplicate Parameter Names

In APIs with object encapsulation, the same parameter name may be used multiple times for different parameters. An example of this is the parameter 'reference' in a number of Adyen endpoints[8]. As a result of this, any reference to 'reference' can reference multiple 'reference' parameters.

For documentation analysis, any reference to a duplicate parameter name is assumed to be a reference to the closest parameter with respect to the parameter whose description the name was found in. The distance measure is the number of edges between the two parameters. Where all top level parameters are connected and all subparameters are connected to their parents, forming a tree. With an equal distance both parameters are considered.

For code analysis, the correct parameter is inferred from the context of the most recently accessed variables. For example, given we just accessed the 'card' parameter, then we can infer that the 'reference' parameter probably corresponds with 'card.reference' and not (e.g.) 'bank.reference'.

[8]https://docs.adyen.com/api-explorer/#/PaymentSetupAndVerificationService/v52/post/payments

Request to Object Conversion

Typically the request passed to an API is deserialized from its original format (JSON, XML) to an object model. Due to this conversion the direct link between request parameters and the variables/fields accessed throughout the code may be lost. Linking in-code variables to a request's parameters is essential.

The solution can vary from system to system. Whereas tracing fields in the deserialization process may not feasible, we can make assumptions on the correspondence of class fields with parameter names. For SpringMVC the field names need to match the parameter keys. Within the Adyen APIs, the object models used during deserialization do have fields whose names correspond directly with the parameter names in the original request. This allows us to infer that $address.getCountry()$ corresponds with the parameter *country*, since that method returns the field *country*.

In some cases, the internal class' field names may differ from the parameter names by the request of the API consumer. This can be due to various reasons. We found the use of JAXB annotations[9] and mapping multiple parameters to the same class field during deserialization to be relevant. With JAXB annotations, the external parameter name can differ from the internal class field's name. Since these annotations are placed above class fields, reflection tools can easily detect when this is being done. For mapping parameters to different fields, defining such a mapping manually mitigates the problem. When two parameters were mapped to the same field, then we associated that field with both external parameters.

Parameter Access

In order to detect constraints, we determine whether a part of the code accesses a parameter, or whether a variable represents a parameter. The access of a parameter does not always mean it is required, if it does depends on the framework being used. For example, in Flask parameters are accessed from a map-like structure. In this case, for $param = req['param']$ the parameter being accessed means that it has to be provided. When the framework converts the request to internal object models, this is not the case. In the deserialization the provided parameters are used to fill the related model instances. Later these instances are used in the flow of the controller method.

When the requests are deserialized as such, the access of the object's attribute corresponding with a parameter does not mean that the parameter is required. We determine whether the parameter is required on the basis of assertions. For example, looking at Snippet 3, we know that in order to reach $process(address)$ *card* can not be null and is thus *card* is required in that branch.

[9]`https://javaee.github.io/jaxb-v2/`

```
...
card = request.getCard()
address = request.getAddress()
if card != null then
  | process(address)
end
...
```

Snippet 3: Example snippet showing how the access of an objects attribute corresponding with a parameter does not directly imply that that parameter is required.

Identifying Invalid States

We use throwable exceptions occcuring in the code to know if the preconditions leading to that condition should be avoided. Any code statement that tells us the preconditions should be avoided is marked as an 'invalid state'. In most cases, checking the code for such throwable exceptions was enough for extracting constraints. However, there are cases in which parameter constraints may not be enforced by explicit exceptions. Take a try-catch construction in Java for example. If an error is thrown, we do not directly know what caused it. This may require the use of static null-pointer detection (e.g. [30]). We did not encounter such try-catch constructions, as such, we only dealt with explicit invalid states.

Errors may also be deferred to a later point in program execution. In this case the results of a validation step may be added to a result map, which is later used to throw exceptions. Due to the nature of our static analysis, such flows are difficult to identify. Our solution is to identify patterns, that can be used to identify such a deferred invalid state. For example, any statement containing $x.addError(...)$ could be tagged as an invalid state.

3.2.6 Constraint Tree validation

Static code analysis can produce false positives, so having a validation phase to increase the precision makes sense. However, the usage of unparsed statements makes automatically validating the constraints ineffective.

If any constraint contains unparsed elements, we can not safely generate validation requests; if a validation request fails we do not know if it was due to the constraint described by the unparsed statement. We could still apply the validation process on constraints which do not have unparsed elements. However, most false positives contained unparsed statements. Furthermore, false positives tended to have a large number of hard to interpret unparsed elements. As a result, differentiating between true and false positives tends to be relatively easy for humans.

Chapter 4

Research Methodology

Our main research question is 'Can (inter-)parameter constraints in complex Web APIs be identified using automated techniques?'. To answer this question, we split this question into the following research questions (RQs):

RQ1: How effective are documentation- and static code analysis in identifying parameter constraints in a large-scale enterprise API? We answer this question by comparing automatically identified constraints against a manually collected ground truth, for both single- and inter-parameter constraints. We are particularly interested in how many constraints can be identified and the number of false positives. False positives are particularly important as they can hinder the adoption process of static analysis tools [13].

RQ2: What are challenges in using documentation- or static code analysis to identify inter-parameter constraints? Besides the aforementioned metrics, we aim to understand the challenges that occur when trying to automatically identify constraints for large-scale industry APIs. We do this in particular to gain a deeper understanding of the viability and future directions of each approach; can the approach work, and what is needed to make the approach work? Unique to our work is the analysis of complex APIs, which face unique challenges in terms of the number of parameters to analyze and the structure of the accompanying code.

In the remainder of this chapter, we explain the APIs and endpoints we selected and how, from those endpoints, we collect the ground truth, and finally how we performed the analysis.

4.1 Selected Endpoints

We aimed at selecting a representative set of Adyen APIs and endpoints which were publicly accessible. These APIs and corresponding endpoints can be explored through the API explorer[1]. At the time of writing, there are three distinct

[1] `https://docs.adyen.com/api-explorer`

public APIs: Checkout (checkout), Payments (pal), and Adyen for Platforms (cal). Within Adyen's API Explorer, these different APIs can be distinguished by their server URL. E.g. the Checkout API has https://checkout-test.adyen.com as a default server path.

We selected endpoints on the basis of the following criteria; an endpoint has to contain inter-parameter constraints, and the internal logic must be dissimilar enough from any previously selected endpoint. This dissimilarity criterion comes from the observation that endpoints frequently featured the same (inter-)parameter constraints, as a result of strong code reuse. Including such similar endpoints would lead to an unbalanced set of endpoints, in which we would effectively be analyzing the same code a number of times.

We selected the following APIs and endpoints; Checkout: */payments*, Payments: */authorise*, */capture*, */storeDetailAndSubmitThirdParty*, */getCostEstimate*, Adyen for Platforms: */createAccountHolder*, */getAccountHolder*, */updateAccountHolder*, */createAccount*, */uploadDocument*. These endpoints can all be found through the API Explorer.

4.2 Ground Truth

The ground truth consists out of a representative set of constraints which we manually collected for each of the selected endpoints. These constraints include both inter- and single-parameter constraints. Three aspects of the ground truth are particularly important; the collection, selection, and representation. Note that we do not publish this ground truth for security reasons.

4.2.1 Collection

Given that only a number of constraints were known beforehand, we had to carefully inspect the code of all selected endpoints for constraints. In this process we start at the controller method and follow the code until its end, taking note of any constraints we find along the way. Any constraints were validated by making API requests corresponding with the constraint in order to ensure their correctness. Additionally, developers from the respective APIs were asked for guidance in pointing out constraints known by them and the general logic of handling requests related to that API.

4.2.2 Selection

Some inter-parameter constraints are trivial, as such not every constraint which is technically a constraint is included. For example, given we have an *address* object with, amongst others, a *country* field which is known to be required. In this case, $address \rightarrow country$ is technically an inter-parameter constraint. However, any of such encapsulation constraints are excluded based on their frequent occurrence and triviality due to being known beforehand through the documentation and their straightforward presence within the code.

Not all payment methods[2] are included either, but instead, one representative payment method is used per payment method category, such as *open invoice* or *bank transfer*. This is mainly done because of the sheer number of payment methods, which internally may rely on similar logic.

4.2.3 Representation

How constraints are represented can **strongly** impact the results' statistics. For example, if we choose to represent $A \rightarrow B \,\&\, C$ as $A \rightarrow B$ and $A \rightarrow C$, then we end up with twice the constraints. The same applies for $A \,||B \rightarrow C$.

Generally, we opt to group logical ORs and logical ANDs together. This is done in order to match how constraints would be present in IF-statements; multiple conditions in the guard (left-side) would lead to a number of consequences in the body (right-side). IF-statements are a particularly common control structure to encode constraints.

4.3 Grouping Challenges

To gain insight into the challenges the two approaches face, we place undetected inter-parameter constraints into one or more challenge categories according to the reasons that lead the constraint to be undetected. These challenge categories can be seen in table A.1 and table A.2 for documentation- and code analysis respectively, available in appendix A.

4.4 Analysis

We analyze all selected endpoints for constraints separately for the documentation and code-based approach. Before running the documentation analysis, we deploy a local build of the API and configure the correct authentication details in order to make requests to this API. After this we specify the endpoint we want to analyze. This ensures we load the correct service information from the OAS files. Finally, both approaches perform their analysis as described in the Approach chapter.

[2]Payment methods are primarily used for the /payments endpoint. A payment method is, at the time of writing, provided through the paymentMethod.type parameter.

4.4.1 Ground Truth Comparison

We manually compared the output given by the approaches to the ground truth. If the identified constraint and ground truth constraint are logically equivalent, then we consider them to be the same constraint. Given that both approaches represent the output of constraints using logical formulations, this comparison can be done directly.

Sometimes only part of a constraint was identified. For example, given $A \rightarrow B \& C$, it would only identify $A \rightarrow C$. In these cases we deviate from the representation standard established in Section 4.2.3, and represent the constraint $A \rightarrow B$ as unidentified and $A \rightarrow C$ as identified.

4.4.2 False Positives

Both approaches can produce false positives. For documentation analysis the likelihood of getting a false positive is reduced due to the added validation stage. For the documentation analysis, any truth table which suggested a constraint between two parameters which was not there would be considered as a false positive. For code analysis, any unique path in the constraint three that incorrectly suggests that its preconditions lead to an invalid state is considered a false positive. For example, the constraint tree in Figure 4.1 contains two false positives: $card\,!=null \rightarrow card.cvc\,!=null$ and $card\,!=null \rightarrow card.iban\,!=null$.

```
card != null →{
  card.cvc = null →{InvalidState },
  card.iban = null →{InvalidState }
}
```

Figure 4.1: An example constraint tree, containing two false positives.

Chapter 5

Results

5.1 Research Question 1

How effective are documentation- and static code analysis in identifying parameter constraints in a large-scale enterprise API?

5.1.1 Inter-parameter Constraints

To answer RQ1 for inter-parameter constraints, we show the number of inter-parameter constraints identified by each approach in Table 5.1.

	Total	Code	Code FP	Doc	Doc FP	Both
/payments	17	11	2	0	0	0
/authorise	15	11	4	3	0	2
/capture	5	2	0	1	0	0
/storeDetailAndSubmit...	5	2	0	1	0	1
/createAccountHolder	4	0	0	3	0	0
/getAccountHolder	1	0	0	0	0	0
/updateAccountHolder	1	1	0	0	0	0
/createAccount	1	0	1	1	0	0
/uploadDocument	3	1	1	1	0	1
/getCostEstimate	1	0	0	1	0	0
Total:	53	28	8	11	0	4

Table 5.1: For every endpoint we show: the *total* number of manually identified inter-parameter constraints, the number of constraints identified by the *code* and *documentation* analysis, with their respective false positives (*FP*), and the number of constraints that were identified by *both*.

Observation 1: Code and documentation analysis together detect 66% of the inter-parameter constraints. Code and documentation analysis detect 35 ((28+11)-4) out of the 53 constraints in total. Typically, using both approaches, the analysis is able to find at least some constraints for every endpoint.

Observation 2: Code and documentation analysis detect different constraints. Between the 28 and 11 constraints found, only 4 were found by both code- and documentation analysis. For this, 3 different endpoints contributed to the total overlap of 4.

Observation 3: Code analysis detects more constraints, but with more false positives. Code analysis detects around 2.5 times more constraints than documentation analysis, it does however produce a number of false positives. Documentation analysis finds fewer constraints, but does not produce false positives.

5.1.2 Single-Parameter Constraints

For (single-)parameter constraints we provide information for all endpoints that had parameter constraints. This only includes code analysis, since documentation analysis is not set up to find single-parameter constraints. Although possible, we did not get any false positives.

	Total	**Identified**
/payments	9	8
/authorise	14	10
/capture	5	5
/storeDetailAndSubmit...	4	4
/createAccountHolder	4	1
/createAccount	1	1
Total:	37	29

Table 5.2: For every endpoint, the *total* amount of parameter constraints and the number of *identified* single-parameter constraints using code analysis.

Observation 4: For the majority of the single-parameter constraints, easy to infer code structures are used to check parameter values. For example, the fraudOffset having to be smaller than 999 would be done with a check similar to $if(request.getFraudOffset() < 999)$. Some notable challenging cases include regex patterns not being parsed to something meaningful, and parsing the parsing of dates.

Observation 5: Code analysis detects 78% of the single-parameter constraints. Code analysis detects 29 out of 37 the single-parameter constraints. For some endpoints it manages to find all parameter constraints. The percentage of single-parameter constraints found is larger than the total number of inter-parameter constraints.

> Code and documentation analysis combined obtain 66% of the inter-parameter constraints. Code analysis obtains 78% of the single-parameter constraints.

5.2 Research Question 2

What are challenges in using documentation- or static code analysis to identify inter-parameter constraints?

5.2.1 Documentation Analysis

For every endpoint we labeled constraints with the reasons for being not detected. These four reasons are described in table A.1, and their short descriptions are in the header of table 5.3. To answer RQ2, we inspect and discuss these reasons in more detail.

Lack of Information (A.1)

By far the most common reason for not identifying a constraint is the absence of information about the constraint. Without descriptions describing the constraint directly it gets harder to identify them through the descriptions alone. Checking the parameters' descriptions for references to parameters directly will not work. Using the contextual relatedness of parameter descriptions could potentially be used as information. However, initial results, as discussed in section 3.1.1, do suggest this is not realistically possible.

In some cases we do not even know that the parameter is present through the documentation. The existence of such parameters can be detected by looking through the object models of the code, as discussed later in section 5.2.2. However, for documentation analysis this is not an option, which means inferring any of such constraints is not possible unless additional manual information is provided. This includes providing a

	A1 No Info	**A2 Implicit Info**	**A3 No Value**	**A4 Validation**
/payments	15	2	0	0
/authorise	7	2	3	2
/capture	4	0	0	0
/storeDetailAndSubmit...	3	1	0	0
/createAccountHolder	0	1	0	0
/getAccountHolder	0	0	0	0
/updateAcountHolder...	0	1	0	0
/createAccount	0	0	0	0
/uploadDocument	2	0	0	0
/getCostEstimate	0	0	0	0
Total:	30	7	3	2

Table 5.3: For every endpoint, the table shows the reasons documentation analysis was not able to identify an inter-parameter constraint. One constraint may not have been identified for multiple reasons.

description of the parameter which is not in the OAS. Manually providing descriptions with the sole purpose of inferring constraints is convoluted. In this case, directly specifying the constraints that might apply on that parameter may be more effective.

Implicit References (A.2)

Some constraints were not detected due to the use of implicit information. There were a number of cases in which the OAS did include documentation on a constraint, but the description did not explicitly mention the name of a parameter. For example, the parameter *stateOrProvince*'s description describes 'Required for the US and Canada'. Any human would know that the *country* parameter's value being equal to 'US' or 'CA' would require stateOrProvince.

In order to make the connection between the implicit information and the referenced parameter, word embeddings could potentially be used. So rather than checking for a direct match, similarity measures could be used to utilize the implicit information. However, this may generate too many candidates to validate, similar to the word embeddings described in Section 3.1.1.

Value not Detected (A.3)

Finding the values constraints depend on can be difficult. There are cases in which, usually amongst other reasons, a value dependent constraint is not detected because the value is not detected or entirely unknown. For example, for constraint $recurring.contract = "ONECLICK" \rightarrow card.cvc$ the value *ONECLICK* was not detected. And for constraint $country = "US" \rightarrow stateOrProvince$ the value *US* was not detected. Providing these values as OAS enum values would allow these values to be detected.

These are two different cases of undetected values. The recurring contract types are limited in number and specific to the company, the country codes are a common standard (ISO-3166-1 alpha-2) for which a larger number of values exists. Therefore including these in the OAS would make less sense and would not provide a solution to the problem. Given the values are not always present in the descriptions using special formatting, parsing descriptions to extract values is not generally possible either.

Value dependent constraints remain to be an open challenge for any black-box model, including documentation analysis. The documentation may provide us with some values, such as enum values in the OAS. However, this is not always the case. Whereas manually specifying possible values could be possible in some cases, this is less viable in the case of $value + additionalValue < 100000$ as we likely do not know the value or have any prior information on it.

Unobserved Constraints (A.4)

API requests used for validation may fail due to unobserved constraints. Only part of a constraint may be detected, which results in this detected part not being identified as a constraint. For example, assume we detect a relationship between $contract = "ONECLICK"$ and $card.cvc$. The real constraint is $contract = "ONECLICK" \rightarrow card.cvc \,\&\, shopperInteraction$. In this case, we can not automatically infer the constraint between $contract = "ONECLICK"$ AND $card.cvc$. This is because the unobserved requirement of $shopperInteraction$ will cause some validation requests to fail. Any request including the contract will fail, since it needs the *shopperInteraction* to be present as well.

While we might not be able to detect the unobserved constraint, we can get some information from cases like this. When this scenario occurs the request will always fail when the parameter with an unobserved constraint is present. The unobserved constraint can be an inter-parameter constraint, as described earlier, or a constraint on the value of a parameter. The outcome is the same for both cases; we do not know the cause of failure. When this is the case, we should raise a flag on this occurring, such that the reason might be resolved manually.

Request Dependencies

API requests may depend on a sequence of preceding requests. For Adyen, a classic example is the authorize and capture steps, in which the capture step captures the payment authorized in the authorize step. As a result, authorize has to be used before capture and the capture request depends on a field returned by the authorize request, namely the PSP reference. Without providing a valid PSP reference the capture request will always fail, which makes inferring constraints difficult.

While generating validation requests, such constraints between different endpoints can be a problem. This is especially the case if the value returned by a preceding request can only be used in a dependent second request once. This could for example happen for the creation of accounts. One request could create an account, the other could then close it. Trying to close the already closed account again would result in an error. In this case, the preceding request would have to be sent for every validation request to get the valid value.

To mitigate this problem, any of such dependencies should be resolved. Within this paper, we were able to manually specify default values for such fields. This circumvented the problem with relative ease. As such, it did not impact the results, and this challenge is not included in Table 5.3. When this is not the case, one would have to specify these dependencies. Atlidakis et al. [2] outlines REST-ler, a tool which is able to infer such dependencies in the sequences of requests.

For documentation, by far the most common reason for not detecting constraints is the absence of information explicitly describing the constraints in the OpenAPI Specifications. Dealing with this absence of information is a major challenge.

5.2.2 Static Code Analysis

For code analysis, we identified a number of challenges our static code analysis faces when extracting inter-parameter constraints. Some of these challenges have been addressed such that they no longer impact the results, take the 'Design Pattern' challenge for example. We still include these challenges, since they could be relevant for other APIs. In order to better understand the challenges at hand, in appendix C we provide a dummy API's flow along with code snippets which contain a number of challenges explained in this section.

	B1: Detection	**B2: Dereferenced**	**B3: Variable Stack**	**B4: Preconditions**	**B5: Control Structure**	**B6: Data Flow**	**B7: Arithmetic**	**B8: Framework**
/payments	1	1	0	1	1	1	0	2
/authorise	0	3	2	1	3	0	1	0
/capture	0	3	2	0	3	0	1	0
/storeDetailAndSubmit...	0	0	0	0	0	3	0	3
/createAccountHolder	0	2	2	0	4	2	0	0
/getAccountHolder	0	0	0	0	0	1	0	0
/updateAccountHolder...	0	0	0	0	0	0	0	0
/createAccount	0	1	0	0	0	0	0	0
/uploadDocument	0	2	0	0	0	0	0	0
/getCostEstimate	0	1	1	0	0	0	0	0
Total:	1	13	7	2	11	7	2	5

Table 5.4: For every endpoint, the table shows the reasons code analysis was not able to identify an inter-parameter constraint. One constraint may not have been identified for multiple reasons.

We observe that, unlike documentation analysis, for code analysis a lack of information was not the leading challenge. Frequently, constraints were encoded within the control-flow of the controller methods in a relatively straight forward fashion. This makes a static code analysis approach which focuses on the analysis of control statements effective. There are however cases for which such a basic approach has to be adjusted or extended. We discuss these cases in more detail in the following sections.

Parameter not Detected (B.1)

When an expression is not associated with a parameter, no constraints will be inferred from that related part of the code. Within our approach, this happened because these parameters are not referenced in the OpenAPI specifications. From a business perspective, these parameters may not be referenced because a function is being deprecated or because certain functionality is only intended to be used by a select group of API consumers. From an API design perspective, these parameters might be missing because documentation still needs to be added.

As a solution to this problem, one can specify any access to fields of a data model as an access to a parameter, regardless of whether the accessed fields are in the documentation or not. To do this, we would have to detect whether a class is part of the data models. This could be done by manually specifying the data models. Given the relatively small number of data model classes, this is viable. Manually specifying parameter names which are not in the OAS is an alternative solution. This would however require knowing which parameters are not in the OAS. In our case, this was not something we would always know beforehand.

Parameter Dereferenced (B.2)

A large number of parameter references are dereferenced at some point. When this challenge occurs we initially can associate an expression with a parameter, but after some steps this reference is lost. Typically the reference is lost due to the developed parser not being able to parse all Java expressions. Within the evaluated APIs this usually happened during object creation. Either object creation when deserializing the request to the internal data model or when creating a new object within the controller method's flow. The latter was more common than the former.

Within deserialization two, or more, parameters may be used to fill the same field of a data model. For example, for a card object the countryCode could either be set from the countryCode parameter directly, or be extracted from the provided IBAN. As a result, we do not know that there are constraints on the country code part of the provided IBAN. Maintaining references in Adyens deserialization forms a significant challenge with no easy solution. This is mostly because deserialization steps are tightly interwoven with the framework that is being used (see section 5.2.2 B.9).

Frequently a number of parameters were used in the creation of a new data model object. This object was then validated for constraints. If the fields in the new model did not correspond with known parameter names, constraints could not be detected. As such, statically maintaining objects is relevant for static code analysis. For the creation of objects within the controller method's flow, some references could be retained if we keep track of objects in our static variable stack. When parsing the construction of an object we can theoretically keep track of which fields of that object correspond to which parameter(s). Due to time restrictions, we have not explored this in more detail.

Static Variable Stack (B.3)

Maintaining basic values in the variable stack is sufficient for inferring most inter-parameter constraints. These basic values include Java primitives, strings, and parameter references, as described in section 3.2.4. This tends to be enough since fields in API requests typically handle basic values as well. For example, a 'name' would be a string and an 'amount' would be an integer. Parameters which are objects consequently exist out of combinations of these basic values. E.g. $"person" : \{"name" : "Hanson", ...\}$. Just maintaining basic values is not always enough. This is particularly relevant for objects, as discussed in Section 5.2.2 B.2. Extending the range of expressions that can be parsed to update the variable stack is discussed briefly in Section 5.2.2 B.4.

Preconditions (B.4)

Some constraints have complex preconditions resulting in another parameter being required or not. For example, consider a hypothetical function $isValidIban(iban)$ in which validity of the IBAN itself depends on a large number of conditions, which the approach aims to parse. Typically, such preconditions exhibit expressions that are difficult for static code analysis to parse. As a result, our approach would produce preconditions containing an often large number of unparsed elements. These are hard to read by humans, even if the precondition could be fully parsed. This is why this has its own challenge category, even though the essence of the problem relates to both B.2 and B.3.

In essence, this challenge relates to the kind of expressions our static analysis tool can parse. Given that we can not take all expressions of any language into account [14], it is important to identify which expressions are important when statically analyzing code for parameter constraints.

For example, the Java default method *Boolean.parseBoolean(String s)* may be a function that is universally used through different APIs. Section 3.2.4 addresses this problem partially for the scope of this paper.

Control Structure (B.5)

The basic evaluation of for-loops is relevant for extracting constraints. For-loops are used in a number of cases to validate parameter constraints within the source code itself. When for-loops were used, this was typically done for parameters with array values. E.g. "people": $[\{"name" : "Frank", ...\}, ...]$. As discussed in section 3.2.1, evaluating the body of the for-loop once would be sufficient to infer the constraints. For-loops are also used for arithmetic constraints. In particular for the type $P1 + P2 + ... = Pn$. For this, one could heuristically specify that all elements of an array type parameter contribute to the sum. Using such heuristics could circumvent a number of complications that come from statically keeping track of stop conditions for looping statements.

Theoretically, a more advanced analysis of for-loops could be needed. Consider a for-loop which iterates over a number of parameter objects. Each of these parameter objects could implement their own *validate(request)* method. The aforementioned heuristics would not suffice. However, in practice, we did not encounter such a structure within the code base.

Switch statements were common for value-dependent constraints. Naturally, case statements lend themselves well for executing logic specific to a certain value of a parameter. Supporting case statements is to a large extent trivial, however due to time limitations and case statements not being the only reason for failing to extract a constraint, these are not implemented. However, since they are not implemented they are still counted towards the results in Table 5.4.

Data Flow (B.6)

Data flow limitations involved the lack of path sensitivity and not keeping track of fall through conditions on certain branches. The impact on the results in noticeable, thus when using static code analysis for inferring inter-parameter constraints, having the approach be data flow sensitive is preferred.

For path sensitivity, our approach is limited when two mutually exclusive branches influence the precondition of another branch. An example of this is shown in figure 4. Here, the checks that are applied to the object in the body of the *evaluateCorners* if-statement depends on the shape of the object. We are not able to infer this correctly.

For conditions on fall-through paths, snippet 5 shows how not keeping track of these conditions may cause incorrectly extraction of constraints. In this case, in order to end up at the exception the fall-through conditions $ID == null$ and $user == null$ would apply. We do not keep track of these conditions. As such, the code analysis would extract the incorrect constraint $auth = True \rightarrow InvalidState$.

Making static analysis path-sensitive is possible [8]. However, our current approach is not easily changed to support path-sensitive analysis. As such, within the analysis of parameter constraints, this is still an open challenge.

```
Function validateShape(object)
    if object.shape = 'square' then
        evaluateCorners = true
    else if object.shape = 'circle' then
        evaluateCorners = false

    if evaluateCorners then
        ...
    end
    return false
end
```

Snippet 4: Method evaluating some property of an object.

```
Function getAccount(request)
    if request.auth == True then
        if request.ID != null then
            return ...
        else if request.user != null then
            return ...

        Throw Exception()
    end
end
```

Snippet 5: Method evaluating some property of an object.

Arithmetic Constraint Syntax (B.7)

Arithmetic constraints include parameters that are related to each other by means of arithmetic. For example, $A+B > 10$. These constraints are common [17], with 107 out of the 633 identified constraints being arithmetic constraints. From these 107 constraints 106 had an arity of 2, with $A >= B$ being by far the most common.

Within our case study, such constraints were often directly present in the code. E.g. $A >= B$ would have a corresponding $if(A >= B)$ statement. This made them easy to extract. Another type of arithmetic constraint involves the addition of parameters having to satisfy some criteria. This happened for $amount + additionalAmount < number$ and for $sum(split.value) = total$. Just like the aforementioned arithmetic constraint, the constraint $amount + additionalAmount < number$ was often directly present in the code as such, and thus easy to extract. This was not the case for constraints such as $sum(split.value) = total$, which was encoded by means of a for-loop. For this constraint, we could try to heuristically analyze for-loops, as described in section 5.2.2 B.5. However, we were not able to resolve that a parameter was being referenced within the for-loop, making this ineffective.

Arithmetic constraints can get as complex as mathematics itself. However, in practice complex constraints are rare. Common arithmetic constraints can be supported easily, whereas slightly more complicated constraints ($sum(splits) = total$) provide more challenges to overcome.

Framework (B.8)

Frameworks can make the inference of constraints more complicated. The remote procedure call (RPC) based framework within Adyen was occasionally used to dynamically add new tasks. These tasks get passed through the framework, to then be handled as a kind of internal API request. Due to certain characteristics of the framework, such as multiple layers of abstraction, resolving which tasks get executed is especially difficult.

Design Patterns

Design patterns are used to address common challenges within software engineering. These patterns may be any of the Gang of Four design patterns or locally specific design patterns. Given that these patterns tend to be common and are generally complex to interpret, this challenge falls into its own category. The factory pattern was the only design pattern we encountered for which analysis was necessary in order to infer parameter constraints.

The factory pattern was particularly relevant for payment methods. Payment methods have their own requirements; an instance of a payment method would check if the request satisfies its requirements. Determining which payment method corresponds with which instance is difficult to do fully automatically using static code analysis.

We addressed this challenge by manually specifying which payment method corresponds with which Java class and which method within that class enforced the constraints. This entirely circumvents the need for static code analysis to resolve the corresponding payment method instance.

Non-Code Constraints

The source code is not the only place where constraints may be present. As an example of this, constraints may be enforced at the database layer. In this case, the maximum length of a field may be limited by table specifications or an SQL query might depend on the presence of a certain field.

Ideally, any constraint is handled directly in the code itself. This enables feedback in API responses, making the API more transparent. Whereas non-code constraints are not unthinkable, they did not occur during this case study and seem unlikely to happen.

Sound Static Analysis

A common theme amongst a number of the aforementioned challenges, particularly B.2, B.3, and B.4, is statically analyzing code in a sound way. This includes determining the kind of expressions we need to parse and how we parse them.

Some challenges pose a solution which relies on the simplification of the underlying code, while still being able to correctly extract constraints. For example, with the analysis of for-loops, analyzing the body once can be sufficient (see Section 5.2.2 B.5). Understanding which simplifications we can make without losing the ability to accurately extract constraints is an important future research direction.

Other challenges are more of an engineering challenge, in which we need to know the kinds of expressions which we need to support in parsing. This includes supporting the static analysis of objects (Section 5.2.2 B.2) and knowing which expressions are relevant (Section 5.2.2 B.4). Generalizing the knowledge of which expressions are relevant for parameter constraints is subject to future research.

For code analysis, the main challenges were data flow insensitivity and engineering a sound code analysis approach. This involves deciding how to evaluate looping control structures, maintaining parameter references throughout the API's code, and deciding how to deal with the variety of expressions in a programming language.

Chapter 6

Discussion

In this chapter we compare our work to existing, comparable approaches, discuss the use of formal constraints, possible security concerns, how we expect code analysis to generalize, and threats to validity.

6.1 Comparison to Previous Works

The major differences of our approach with respect to approaches existing in literature are the following; our approach focused on an API with a large number of parameters and our approach focuses on the analysis of code to infer constraints.

Existing works that specifically focused on inter-parameter constraints for Web APIs only evaluated APIs with a small number of parameters. This may make search-based techniques more effective, as is reflected in the results. The works by Gao et al. [11] and Wu et al. [35] both succeed a precision of 90%, whereas we obtain a precision of 20% using a documentation analysis approach largely similar to the one used in the work of Wu et al.

Concerning code analysis, Wu et al. is the only work we found which describes the analysis of code to infer inter-parameter constraints (see section 2.4). There are several differences. First and foremost, their approach performs data-flow analysis whereas we extract a constraint structure directly. The implication of this is that our approach does not have a strict need for the validation of candidates, similar to what the doc analysis does. Generating combinations of accessed parameters in the same way they did also lead to an explosion of possible combinations. Secondly, their approach does not address many of the features any software system has. Key examples are analyzing methods separately, as opposed to within its given context, and not supporting value-dependent constraints. The latter effectively made their approach not support single-parameter constraints or any constraints such as $X = V \rightarrow Y$.

The work by Pandita et al. [22] does not provide any specific information about inter-parameter constraints, nor does it only analyze Web APIs. With the information they provided, a direct comparison is not feasible.

6.2 Code Analysis for Complex APIs

The difference between static code for complex and simple APIs may not be as strong as one might think. Simple and complex APIs rely on the same set of (Java) expressions and often deploy some kind of framework in the process. To this extent, a large number of challenges described in Section 5.2.2 would apply to a simple API as well.

Complex APIs will have more code, which makes fast sound analysis a point of focus. Extensively analyzing every method might be too computationally intensive. The approach described in this paper is able to analyze methods quickly enough for large software systems. However, when continuing to develop this approach, in particular to make it data-flow sensitive, this aspect needs to be considered.

6.3 Using Formal Constraints

Provided a reasonably complete set of formal constraints, there are two main use cases: documentation and conformance testing. As is, we are not able to automatically identify all constraints. This limits the practical use of the constraints that can be detected.

6.3.1 Documentation

External documentation ought to help anyone who uses the API to integrate better. To this end, the formal constraints should be deployed in a way that helps the API consumer. Due to the large number of constraints, it may generally not be beneficial to show all constraints all the time. Therefore having more dynamic documentation which uses progressive disclosure to point out relevant constraints to a user is an interesting use-case. E.g. the requirements of a specific payment method will only be shown when an API consumer indicates to want to use that payment method.

6.3.2 Conformance Testing

Given a formal set of constraints in a machine-readable format, we can check requests and APIs for conformance. Martin-Lopez [16] describes a tool that can do both of these things. Being able to dispatch non-conformant requests before they reach the controller of an API can save time and resources. The tool described by Martin-Lopez also generates a large number of test-cases based on these constraints. This way APIs can also be monitored for changes in constraints and possible non-conformance to the business requirements reflected through the provided constraints. Such constraint oriented testing of Web APIs is not a focus within current literature [1, 2, 9, 29].

6.4 Security Concerns

Constraints extracted from the source code reflect the code to a certain extent. As such, the constraints can reveal information we do not want everybody to know. This is relevant for possible attacks on the APIs and revealing sensitive information.

The constraints provide information on boundary conditions used within the code. A malicious third party could potentially use this information to exploit the API more easily. As such, automatically making all identified constraints public is not preferable. In terms of sensitive information, the constraints may reveal features that are still in development before being announced to the public. The constraints may also reveal parameters that are only intended to be used by a select group of API consumers.

6.5 Generalization

The current approach has been designed for a specific system which uses certain conventions for coding. It uses a remote procedure call (RPC) based framework written in Java. Throughout the development of the approach we kept any assumptions that may limit the approach in other scenarios in mind. In this section, we discuss how we expect the current approach to generalize.

6.5.1 Programming Language

The code analysis approach can work regardless of the key programming language of the API. The main features of the approach are observing the access of parameters in the code and analyzing the control structures in which these parameters are used. Most popular object-oriented languages contain similar control structures. Observing parameter access will differ from language to language and from framework to framework (see section 3.2.5), but as long as a connection between request parameters and expressions in the code can be made this approach will work.

6.5.2 API Frameworks

A large number of API frameworks exist, in this section we discuss Jersey, Spring, and Flask due to their popularity. In general, frameworks deal with dispatching the request sent to the API to the corresponding (controller) method internally. Frameworks do not contain any logic as to what to do with the contents of a request once arrived at its controller method.

The Jersey, Spring, and Flask frameworks provide similar functionality. They create controller classes with controller methods, indicated as such with annotations. E.g. $@GetMapping(''/endpoint'')$ for methods in Spring and $@Path(''endpoint'')$ for classes in Jersey. They also provide functionality for specifying which HTTP method(s) they should handle. Such explicit annotations make it trivial to correspond the correct controller method with an endpoint.

For accessing parameters in dispatched requests, all three frameworks provide conventions. In Flask request bodies can be accessed as a map, where parameters in the request correspond directly with the key needed to access them. For example, $language = req\_data[''language'']$ can be used to access the *language* parameter. Spring and Jersey also have a direct link between the parameters sent in a request and variables in the code. In both cases, this is done by annotating controller method parameters. These annotations are different for each framework. For Jersey, a parameter may be annotated as follows $methodName(@FormParam(''deliveryAddress'')\ String\ deliveryAddress)$.

As a conclusion, the key features needed for static code analysis for inter-parameter constraints are present for three popular frameworks.

6.5.3 Query Style

The query style is how parameters are passed to the API in a request. We consider three main flavors: RPC, REST, and GraphQL.

Remote procedure call (RPC) based frameworks treat API requests as calls to functions, where the arguments to the function are put in either the query string or the body. This is contrary to REST, which often involves path parameters such as */store/orders/orderID*. Path parameters are unlikely to be involved in constraints. Research has shown that about 0.8% of constraints involve path parameters [17], all of which were located within the same API. Both body and query string parameters are used often. The sole difference for the approach is the method of encoding the parameters for a request.

GraphQL deviates in terms of query style. For (HTTP) GET operations the queries can be compared to SQL. Any mutations, such as DELETE and PATCH do use a style similar to RPC. For GET requests, the queries filter data rather than perform operations on it. As such, any constraints that lead to errors are hard to think of and inter-parameter constraints may not be present in such requests.

6.6 Threats to Validity

Internal Validity. The ground truth we used consists out of a representative set of constraints which we manually collected for each of the selected endpoints. In this process[1] we thoroughly inspected the related code and verified them by making requests to the API. As such, we are sure that we have a correct set of constraints. For completeness, we inspected all available endpoints within Adyen for constraints. From the relevant endpoints, we selected the constraints such that they represent a diverse selection of code. As such, we are confident that the results represent the different APIs within Adyen well.

External Validity. Given that this research is a case study done in one company, research into other complex APIs is needed for further generalization of the results. However, given the size and scale of Adyen's software, we are confident that the results found in our study are representative for other large-scale companies. In Section 6.5 we discuss how we expect the approach the generalize across different programming lan-

[1]See Section 4.2 for more details.

guages, frameworks, and query styles. We expect the static code analysis approach to generalize.

Chapter 7

Conclusions and Future Work

We developed two approaches for inferring parameter constraints for complex Web APIs. One approach analyzes online documentation to infer inter-parameter constraints, the other depends on static code analysis to extract inter- and single-parameter constraints from the control flow of the API's source code. The documentation- and code-based approach are able to identify 21% and 53% percent of the constraints respectively. When the constraints identified by both approaches are combined, 66% of the inter-parameter constraints can be identified. Code analysis is able to identify 78% of the single-parameter constraints.

The two approaches face largely separate challenges. The documentation based approach largely suffers from a lack of available explicit information describing the constraints. Static code analysis tends to be able to extract constraints from the source code by maintaining a basic variable stack, evaluating method calls, and analyzing conditions in for-loops, switch statements, and if-else statements. The main challenges we faced within this case study were data flow insensitivity and engineering a sound static code analysis approach.

RQ1: How effective are documentation- and static code analysis in identifying parameter constraints in a large-scale enterprise API?
Code and documentation analysis combined obtain 66% of the inter-parameter constraints. Code analysis obtains 78% of the single-parameter constraints.

RQ2: What are challenges in using documentation- or static code analysis to identify inter-parameter constraints?
For documentation, by far the most common reason for not detecting con-

straints is the absence of information explicitly describing the constraints in the OpenAPI Specifications. Dealing with this absence of information is a major challenge.

For code analysis, the main challenges were data flow insensitivity and engineering a sound code analysis approach. This involves deciding how to evaluate looping control structures, maintaining parameter references throughout the API's code, and deciding how to deal with the variety of expressions in a programming language.

We view future work largely in the light of developing and exploring the different approaches in order to obtain a more complete set of parameter constraints. After tools have matured enough to extract parameter constraints accurately, the applications of these constraints would gain more focus. We discuss possible use-cases of parameter constraints in Section 6.3.

Static code analysis would benefit from expanding the current approach and exploring this approach on a larger variety of APIs. For the current approach, data flow sensitivity is the most significant, general challenge with no straight-forward solution. Exploring a larger variety of APIs would help with understanding in what other ways constraints may be present in the code. This would also help us with engineering a more generally applicable tool by knowing the kinds of expressions that we need to parse, and the kind of simplifications we can make while still obtaining a sound static analysis.

For documentation analysis, we inferred inter-parameter constraints from the API reference. However, there are other sources of documentation which could prove to be useful. In many cases, the API reference is not the only source of documentation. Whether additional documentation could improve a documentation based approach is currently unclear. Extracting the content from more external pages containing documentation or comments within the source code could be beneficial.

Provided the large number of requests that can be made to a local build of an API, search-based approaches could prove to be effective. As discussed in the related work section, Gao et al. [11] uses a decision tree based approach to infer inter-parameter constraints. To guide such a search-based approach, any source of information may prove to be useful. For example, the output of our documentation- and code-based approach could guide such an algorithm.

Bibliography

[1] Andrea Arcuri. Restful api automated test case generation. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 9–20. IEEE, 2017.

[2] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Rest-ler: automatic intelligent rest api fuzzing. *arXiv preprint arXiv:1806.09739*, 2018.

[3] Joop Aué, Maurício Aniche, Maikel Lobbezoo, and Arie van Deursen. An exploratory study on faults inweb api integration in a large-scale payment company. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 13–22. IEEE, 2018.

[4] Len Bass and Bonnie E John. Linking usability to software architecture patterns through general scenarios. *Journal of Systems and Software*, 66(3):187–197, 2003.

[5] Barry W Boehm, John R Brown, and Mlity Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, pages 592–605. IEEE Computer Society Press, 1976.

[6] Stefan Bordag. A comparison of co-occurrence and similarity measures as simulations of context. In *International Conference on Intelligent Text Processing and Computational Linguistics*, pages 52–63. Springer, 2008.

[7] Mark Boyd. How to choose architectural styles and specification formats for your apis, Sep 2017. URL `https://www.programmable`

web.com/news/how-to-choose-architectural-styles-and-specification-formats-your-apis/analysis/2017/09/27.

[8] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 270–280, 2008.

[9] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Automatic generation of test cases for rest apis: a specification-based approach. In *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*, pages 181–190. IEEE, 2018.

[10] Fabian Fagerholm and Jürgen Münch. Developer experience: Concept and definition. In *2012 international conference on software and system process (ICSSP)*, pages 73–77. IEEE, 2012.

[11] Chushu Gao, Jun Wei, Hua Zhong, and Tao Huang. Inferring data contract for web-based api. In *2014 IEEE International Conference on Web Services*, pages 65–72. IEEE, 2014.

[12] ISO/IEC. Iso/iec 25010: 2011 systems and software engineering–systems and software quality requirements and evaluation (square)–system and software quality models, 2011.

[13] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[14] Panagiotis Louridas. Static code analysis. *Ieee Software*, 23(4):58–61, 2006.

[15] Walid Maalej and Martin P Robillard. Patterns of knowledge in api reference documentation. *IEEE Transactions on Software Engineering*, 39(9):1264–1282, 2013.

[16] Alberto Martin-Lopez. Automated analysis of inter-parameter dependencies in web apis. In *International Conference on Software Engineering, ACM Student Research Competition*, 2020.

[17] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. A catalogue of inter-parameter dependencies in restful web apis. In *International Conference on Service-Oriented Computing*, pages 399–414. Springer, 2019.

[18] Mark Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. " O'Reilly Media, Inc.", 2011.

[19] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[20] Brad A Myers and Jeffrey Stylos. Improving api usability. *Communications of the ACM*, 59(6):62–69, 2016.

[21] Nathalie Oostvogels, Joeri De Koster, and Wolfgang De Meuter. Inter-parameter constraints in contemporary web apis. In *International Conference on Web Engineering*, pages 323–335. Springer, 2017.

[22] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring method specifications from natural language api descriptions. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 815–825. IEEE, 2012.

[23] Marco Piccioni, Carlo A Furia, and Bertrand Meyer. An empirical study of api usability. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 5–14. IEEE, 2013.

[24] Irum Rauf, Pekka Perälä, Jouni Huotari, and Ivan Porres. Perceived obstacles by novice developers adopting user interface apis and tools. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 223–227. IEEE, 2016.

[25] Irum Rauf, Elena Troubitsyna, and Ivan Porres. A systematic mapping study of api usability evaluation methods. *Computer Science Review*, 33:49–68, 2019.

[26] Martin P Robillard and Robert Deline. A field study of api learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[27] Wendel Santos. Which api types and architectural styles are most used?, Feb 2018. URL `https://www.programmableweb.com/news/which-api-types-and-architectural-styles-are-most-used/research/2017/11/26`.

[28] Ahmed Seffah and Eduard Metzker. The obstacles and myths of usability and software engineering. *Communications of the ACM*, 47(12):71–76, 2004.

[29] Sergio Segura, José A Parejo, Javier Troya, and Antonio Ruiz-Cortés. Metamorphic testing of restful web apis. *IEEE Transactions on Software Engineering*, 44(11):1083–1099, 2017.

[30] Fausto Spoto. Precise null-pointer analysis. *Software & Systems Modeling*, 10(2):219–252, 2011.

[31] Raj Srinivasan. Rpc: Remote procedure call protocol specification version 2, 1995.

[32] Phil Sturgeon. Understanding rpc vs rest for http apis, September 2016. URL `https://www.smashingmagazine.com/2016/09/understanding-rest-and-rpc-for-http-apis/`.

[33] Jeffrey Stylos, Benjamin Graf, Daniela K Busse, Carsten Ziegler, Ralf Ehret, and Jan Karstens. A case study of api redesign for improved usability. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 189–192. IEEE, 2008.

[34] Jaroslav Tulach. *Practical API design: Confessions of a Java framework architect*. Apress, 2008.

[35] Qian Wu, Ling Wu, Guangtai Liang, Qianxiang Wang, Tao Xie, and Hong Mei. Inferring dependency constraints on parameters for web services. In *Proceedings of the 22nd international conference on World Wide Web*, pages 1421–1432, 2013.

Appendix A

Case Study - Challenge Groups

Code	Short Description	Description
A.1	No info.	No information about the constraint in the API reference.
A.2	Implicit Info.	The constraint was described, but the respective parameters were not referenced directly.
A.3	Value not detected.	The value the constraint depends on was not detected.
A.4	Validation	Validation requests failed due to unobserved constraints.

Table A.1: High-level reasons for failing to detect constraints for doc analysis.

Code	Short Description	Description
B.1	Detection	Parameter was accessed, but not detected because it was not in the OpenAPI specification.
B.2	Dereferenced	Parameter was detected, but dereferenced in variable assignment or object creation.
B.3	Variable Stack	The value of a variable could not be resolved or be maintained fully statically.
B.4	Preconditions	An expression or function imposes a precondition that can not fully be evaluated. E.g boolean cvcRequired = cvcRequired(type)
B.5	Control Structure	A control statement such as 'For, switch, try/-catch' was not handled fully.
B.6	Data Flow	A constraint was missed due to the analysis not being data flow sensitive.
B.7	Arithmetic	Some constraints with arithmetic syntax are not fully supported. E.g. A + B + . . . = X
B.8	Framework	Functionality specific to the API framework was used and caused limitation in extracting constraints.

Table A.2: High-level reasons for failing to detect constraints for code analysis.

Appendix B

Documentation Analysis - Word Embeddings

We explored the usage of word embeddings [19] as a means to find candidates. Using these vectors, we can obtain a semantic similarity between words, and sentences. The premise is that parameters with semantically similar descriptions are more likely to be related to each other in a constraint.

For every parameter, we calculated its semantic similarity to all other parameters of an endpoint. This allows us to create a list of the most similar parameters for each parameter. If the semantically similar parameters are more likely to be related in constraints, then exploring the most similar parameters would allow us to find constraints without having to exhaust all parameters.

In this process we used SpaCy[1] for our word embeddings and similarity measures. We used the *en_core_web_sm* model, which contains vectors trained on written text such as blogs, news, and comments. For similarity, we used SpaCy's default similarity method for sentences. This method assumes a bag-of-words model for sentences, where the vector of a sentence is the average of the vectors of each word. The similarity between sentences is then determined with the cosine similarity between the two vectors.

In order to test the premise, we used the ground-truth to see the position parameters involved in constraints would have in the list of most similar parameters. For each constraint in the ground truth, we made pairs of related parameters. E.g. for $or(card, bank)$ the pair [card, bank] and for

[1] `https://spacy.io`

$tenderReference \to uniqueTerminalId$ the pair [tenderReference, uniqueTerminalId]. Then for every pair we determined the minimum ranking the parameter would have in the other's similarity list between the two parameters. For example, assume that *bank* is on position 8 in *card*'s list and *card* is position 5 on *bank*'s list, then 5 is the rank used for the results shown in Table B.1.

Endpoint	**Ranks**	**#Params**	**#Requests**
/payments	6, 21, 8, 22	371	73458
/authorise	51, 21, 3, 8, 324, 1, 6, 20	378	74844
/capture	31, 1, 18, 8	192	38016
/storeDetailAndSubmit...	1, 1	51	10098
/createAccountHolder	3, 2, 67	103	20394
/updateAccountHolder...	1	3	54
/createAccount	1	4	108
/uploadDocument	1	7	378
/getCostEstimate	1	22	4158
Total:	-	-	221508

Table B.1: Table showing the ranking and theoretical number of requests needed when using word embeddings as a similarity measure. The *ranks* column shows the individual ranks of relevant parameters, the *#Params* the number of parameters for the given endpoint, and *#Requests* the number of requests needed to validate all candidates.

Looking at the results in Table B.1, we can observe that the rank of relevant parameters is typically within the top 22. This heuristic allows us to filter the search space significantly. Using the formulas provided in section 3.1.2, we can calculate the number of requests needed if for every parameter we check the top 22 most similar parameters. For simplicity, we assume every parameter has two values, as such the number of requests is given by $22 * P * 3^2$, where P is the number of parameters in an endpoint. For every endpoint the number of requests needed to validate all the candidates is given in the *#Requests* column.

Following these calculations, we would have to make 221508 requests in order to find most constraints that include 2 parameters, which given 5 requests a second would take little over 12 hours. Largely due to the large number of candidates, even for constraints which only include 2 parameters, the approach is rendered impractical.

Appendix C

Code Analysis - Unhappy Flow

To gain a better understanding of what the challenges may look like in practice, we provide a dummy API model which has enforces multiple constraints in its flow. This flow contains a lot of constructs which are hard for static code analysis to deal with, as such this Appendix is called the *unhappy flow*. As shown in Figure C.1, an HTTP request is sent to the API, given a specific endpoint. The dispatcher handles this request. First, the request is deserialised using a deserialiser (see Listing C.1) into the internal data model (see Listing C.2). This model is passed as a request to the /payment controller, which has two consecutive tasks, as shown in Listing C.3 and Listing C.4. The second task also spawns a new task flow by making a call to the /invoice controller (see Listing C.6).

To relate the challenges identified in Section 5.2.2 to the code, we provide comments in the dummy code below. These comments mention the code of the challenge, along with some additional explanation.

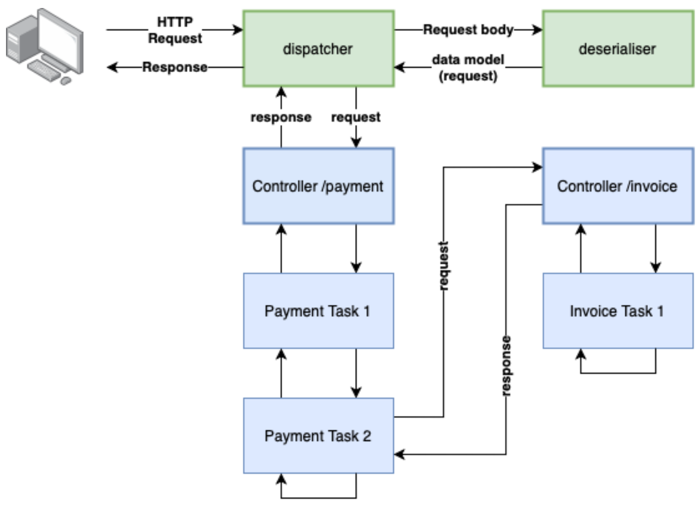

Figure C.1: Dummy API flow, in which a request is made and handled by the dispatcher. First, the request is deserialised into its internal data model (request), then this request is passed to the controller for the /payment endpoint. This endpoint includes two tasks and makes a call to the /invoice controller.

Listing C.1: Dummy request deserializer.

```
class Deserialiser {

   public static PaymentRequest deserialize(HttpRequest request) {
      // Create a new payment request model and populate it with the
          body.
      PaymentRequest pr = new PaymentRequest();
      JSONObject body = request.getBody();

      pr.setCard(body.get("card"));
      pr.setEncryptedCard(body.get("encryptedCard"));
      pr.setPaymentMethod(body.get("paymentMethod"));

      pr.setAmount(body.get("amount"));

      JSONObject invoiceDetails = body.get("invoiceDetails");
      pr.setInvoiceDetails(invoiceDetails);

     // Shopper email might have been provided through a different field.
     // B.2. the invoiceDetails.shopperEmail is dereferenced from the
         request's shopperEmail field.
      String shopperEmail = body.get("shopperEmail") != null ?
                               body.get("shopperEmail") :
                                   invoiceDetails.get("shopperEmail");
      pr.setShopperEmail(shopperEmail);
      pr.setShopperCountry(body.get("shopperCountry"));

      return pr;
   }
}
```

Listing C.2: Dummy model of a payment request.

```
class PaymentRequest extends AbstractRequest {

   Card card;
   EncryptedCard encryptedCard;
   String paymentMethod;

   Amount amount;
   InvoiceDetails invoiceDetails;

   String shopperEmail;
   String shopperCountry;

   // -- getters / setters --
}
```

Listing C.3: First task executed in the dummy /payment flow.

```
class PaymentTask1 {
   public boolean in(AbstractRequest request) {
      // Decrypt the card and add it to the request.
      decryptCard(request);
      // ...
   }

   public boolean out(AbstractRequest request) {
      buildResponse(request);
   }

   public void decryptCard(AbstractRequest request) {
      // Try to decrypt if there is another card.
      if (request.getEncryptedCard() != null) {
         try {
            Card card = Util.decrypt(request.getEncryptedCard());
            request.setCard(card);
         } catch (Exception decryptException) {
            Logger.info("Provided encrypted details invalid.");
         }
      }
   }
}
```

Listing C.4: Second task executed in the dummy /payment flow.

```
class PaymentTask2 {

   public boolean in(AbstractRequest request) {
      // ...
      Card card = request.getCard();
      validateCard(card);

      // Design pattern: An initiation is retrieved from the
          PMInitiation factory, the specific handle method defines the
          constraints.
      PMInitation initation = PMInitation.get(request.getBrand());
      initation.handle(request);

      if (request.getInvoiceDetails() != null) {
         // Spawn a new task chain for invoice requests.
         // B.8: Tracing through the logic of InvoiceController.handle()
             is not feasible due to the complexity of the framework,
             however invoice task 1 does enforce constraints.
         InvoiceController.handle(request);
      }
   }

   public void validateCard(Card card) throws Exception {
      // The card has to be provided.
      if (card != null) {
         // Check the details of the card...
         if (Util.isEmptyOrNull(card.getHolder())) {
            throw Exception("No card holder provided.");
         }
      } else {
         // B.2: Either encryptedCard or card was needed. We can not
             infer this, since we do not know that the card field can be
             populated by both the encryptedCard information and normal
             card information. Payment task 1 relates to this.
         throw Exception("No card provided");
      }
   }
}
```

Listing C.5: Dummy factory pattern used for payment methods.

```
enum PMInitation {
   ideal(IDeal.get()),
   ...

   Handler handler;

   PMInitation(Handler handler) {
      this.handler = handler;
   }

   public static PMInitation get(String method) {
      // Mapping of alternatives.
      switch (method) {
         case "iDeal":
            method = "ideal";
            break;
         default:
            break;
      }

      // Find the correct initiation.
      for(PMInitation pmInitation : PMInitation.values()) {
         if (pmInitation.name().equals(method)) {
            return pmInitation;
         }
      }

      // Fallback option, nothing found.
      return fallback;
   }

   public void handle(Request request) {
      handler.handle(request);
   }
}

class IDeal() {

   public static void handle(Request request) throws Exception {
      if (request.getShopperEmail() == null) {
         throw Exception("Shopper email is missing");
      }
      // ...
   }
}
```

Listing C.6: First, and only, task executed in the dummy /invoice flow.

```
class InvoiceTask1 {
   public boolean in(PaymentRequest request) {
      InvoiceDetails details = request.getInvoiceDetails();

      OpenInvoice openInvoice = details.getOpenInvoice();
      if (openInvoice != null) {
         int whole = openInvoice.getTotalAmount();
         int sum = 0;
         // B.5: We only iterate the body of the for-loop once.
         for (Entry entry : openInvoice.getEntries()) {
            // B.3.: A new object is created in which the reference to
                the invoiceDetails.openInvoice.amount is lost.
            OpenInvoiceParameters params = new OpenInvoiceParameters(
                  openInvoice.getAmount(), openInvoice.getDueDate());
            OpenInvoiceValidator.validate(params);
            // Add the amount of a single entry to the sum.
            sum += openInvoice.getAmount();
         }

         // B.7: We do not keep track of all the amounts that
             contributed to the sum.
         // Therefore we can not infer this arithmetic constraint.
         if (sum != whole) {
            Logger.info("...");
            throw Exception("The whole can not be greater than the sum
                of its parts.");
         }
      }
   }
}

class OpenInvoiceParameters {

   Amount openInvoiceAmount;
   Date dueDate;

   public OpenInvoiceParameters(Amount amount, Date dueDate) {
      this.openInvoiceAmount = amount;
      this.dueDate = dueDate;
   }

   // -- getters / setters
}
```