Data Generation for JSON Schema

Lyes Attouche

PSL
UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à Université Paris Dauphine

Data Generation for JSON Schema

Soutenue par
Lyes ATTOUCHE
Le 12 Décembre 2024

École doctorale n°543
École Doctorale SDOSE

Spécialité
Informatique

Composition du jury :

Mohamed-Amine BAAZIZI MdC, Sorbonne Université	*Co-directeur de thèse*
Dario COLAZZO Professeur, Université Paris Dauphine - PSL	*Directeur de thèse*
Pierre GENEVÈS Directeur de recherche, Université Grenoble-Alpes	*Rapporteur*
Benjamin NGUYEN Professeur, INSA Centre Val de Loire	*Rapporteur*
Marta RUKOZ Professeure, Université Paris Nanterre	*Présidente*
Katerina TZOMPANAKI MdC, CY Cergy Paris Université	*Examinatrice*

Dauphine | PSL
UNIVERSITÉ PARIS

Contents

Chapter 1

Introduction

1.1 Context and Background

In today's world, data is omnipresent and plays a crucial role in shaping industries, technologies, and daily lives. This explosion of data is driven by advancements in digital technologies, social media, and the proliferation of internet-connected devices. These devices range from personal gadgets such as smartphones and fitness trackers to smart home appliances and public infrastructure. In urban environments, sensors are increasingly utilized to monitor traffic flow and environmental conditions, enhancing urban living. For instance, traffic sensors installed on roads collect real-time data on vehicle counts and speeds, helping to optimize traffic management and reduce congestion.

As data becomes more central and encompasses various types: structured, unstructured, and semi-structured, the need for efficient solutions to store and exchange it has become critical. This is particularly important in an interconnected world dominated by web-based services and platforms. The necessity to transfer data between systems, devices, or applications has driven the development of various storage and exchange solutions. Databases have emerged as key methods for organizing and managing data. Solutions tailored to specific needs range from traditional relational databases, where structured data is organized into tables and known for their robustness, to modern non-relational databases such as document stores and graph databases. These newer data modeling and organization solutions are designed to handle unstructured or semi-structured data, offering more flexibility in managing complex data structures, although they may lack the consistency and reliability of traditional databases.

Each of these models has its advantages and limitations. Relational databases, for example, provide consistency and structure but may struggle with highly complex, hierarchical, or unstructured data. In contrast, non-relational databases, such as key-value stores, offer more flexibility and scalability but may lack the strict transactional guarantees of relational systems. Depending on the nature of the data and the goals of the application, one may choose between a structured or unstructured approach. The challenge lies not only in storing the data but also in ensuring that it can be exchanged easily and efficiently

between systems, which is essential for the seamless operation of modern, interconnected technologies.

In our context, we are particularly interested in semi-structured data, which serves as a bridge between fully structured and unstructured data. Formats like XML, JSON, and HTML have become foundational as they provide a flexible way to describe data, balancing structure and flexibility. XML and JSON, for instance, are widely used in web communication, making it easier for systems to interact with one another, exchange data, and enable integration across platforms.

Historically, XML held the position of the primary data format for web applications, offering a standardized approach to representing structured documents. Introduced in 1998, XML aimed to extend the capabilities of HTML by empowering users to create their own document structures, thus allowing for a more nuanced representation of information. However, due to the verbosity and complexity of XML, coupled with the rapid evolution of web applications that demanded more efficient and flexible data interchange mechanisms, the need for alternatives grew. This shift in requirements ultimately led to the emergence of JavaScript Object Notation (JSON).

JSON quickly became the most widely adopted format due to its concise syntax, ease of use, and lightweight nature. Its simple and readable format is particularly well-suited for data interchange in APIs and web services, where efficient representation and rapid parsing are essential. This shift toward JSON reflects a broader trend in software development focused on improving efficiency and reducing complexity. As JSON gained prominence in modern web applications and services, its straightforward structure proved to align well with the needs of users, offering a more intuitive approach to representing data compared to XML. While XML once served as the foundational technology for early web data exchange, JSON has largely replaced it in modern applications, especially for exchanging data between systems.

1.1.1 JSON

JSON has become an integral part of modern web applications, with a significant majority of APIs utilizing it for data transmission and representation, such as the Twitter API, Facebook Social Graph API, and others. The emergence of large language models (LLMs) has further solidified the importance of JSON, as they often rely on this format for effective communication.

Its widespread adoption driven by the rise of RESTful APIs, is supported by a robust ecosystem of tools that enhance usability. Numerous tools exist for verifying JSON syntax, ensuring data integrity, and converting data from various formats into JSON. These resources assist users in managing and manipulating JSON data efficiently, further streamlining the development process.

Moreover, JSON is supported across many programming languages, each offering built-in means for parsing, generating, and manipulating JSON data. This broad compatibility increases its appeal, allowing developers to work seamlessly with JSON regardless of their programming environment.

In the realm of databases, JSON has emerged as a favored format for data storage in NoSQL systems. For instance, MongoDB utilizes a flexible schema to store data in JSON-like documents (BSON format), allowing for high performance and scalability. Similarly, Couchbase supports JSON natively, enabling rapid data access and complex querying. Additionally, JSON is often used to represent metadata and configuration files.

At its core, JSON consists of key-value pairs and array structures, among other data types, making it a versatile choice for various applications.

Example 1 [1]. *The following text represents a JSON document used by the Twitter API to encode tweets and illustrate their structure. The document includes various pieces of information associated with the tweet, such as the creation date, text content, user information.*


```
{ "created_at": "Thu Apr 06 15:24:15 +0000 2017",
  "id_str": "850006245121695744",
  "text": "1\/ Today we\u2019re sharing our vision for the future of
    the Twitter API platform!\nhttps:\/\/t.co\/XweGngmxlP",
  "user": {
    "id": 2244994945,
    "name": "Twitter Dev",
    "screen_name": "TwitterDev",
    "location": "Internet",
    "url": "https:\/\/dev.twitter.com\/",
    "description": "Your official source for Twitter Platform news,
        updates & events. Need technical help? Visit
        https:\/\/twittercommunity.com\/\u2328\ufe0f #TapIntoTwitter"
  },
  "place": {},
  "entities": {
    "hashtags": [],
    "urls": [
      { "url": "https:\/\/t.co\/XweGngmxlP",
        "unwound": {
          "url": "https:\/\/cards.twitter.com\/cards\/18ce53wgo4h\/3xo1c",
          "title": "Building the Future of the Twitter API Platform"
        }
      }
    ],
   "user_mentions": []
  }
}
```


[1] This example was taken from the Twitter Developer documentation, which can be accessed here

1.1.2 JSON Schema

Despite the widespread adoption of JSON for various applications, its inherent flexibility posed significant challenges. While JSON allowed users to represent almost any kind of data structure, it lacked a standardized way to define or enforce these structures. As the use of JSON expanded, the absence of formal specifications for defining and validating data became increasingly evident.

Initially, validation relied heavily on documentation or custom code to ensure that data adhered to expected formats. Although this method was effective when dealing with simpler data, it became inadequate or challenging when confronted with more complex and deeply nested structures. JSON lacked built-in mechanisms for enforcing data types, required fields, or constraints, which led to unpredictable behavior, bugs, and inefficiencies. For instance, when the same data needed to be processed across different environments or programming languages, developers often had to duplicate efforts, as they were required to write similar code serving the same function in different languages.

The introduction of JSON Schema [2] aimed at resolving these issues by providing a standardized method to define and validate the structure of JSON data. With JSON Schema, developers can specify the type of values the JSON data should follow, required fields, and apply more complex rules, such as patterns for string values, array length restrictions, and numeric value ranges. This formalization allows for automated validation, ensuring that data conforms to predefined standards, which is especially crucial in large systems and APIs where consistency and correctness are paramount.

Moreover, JSON Schema has become essential in API development and is widely used in tools and platforms like OpenAPI [3], Swagger [4], and Postman [5] to define the structure of both request and response bodies and to ensure the integrity of their data pipelines. In many cases, different teams develop separate API components. The absence of a standardized format, combined with JSON's flexibility, can lead to misinterpretations of data formats. JSON Schema provides a shared understanding of data structures, reducing the risk of inconsistencies, security vulnerabilities, and integration errors. By offering this structured approach, JSON Schema introduces an essential layer of validation and standardization, facilitating the scalability and maintenance of JSON-based systems, and enabling teams to focus on their core tasks rather than duplicating efforts to validate data manually.

While JSON Schema is often associated with API validation, its applications extend beyond that. It is increasingly used in data modeling for microservices, NoSQL database schema enforcement, and configuration management in complex systems. By offering a formal definition of data structures, it aids in managing data consistency and integrity throughout the software development lifecycle.

Along with the features it offers for defining constraints for different data types, a key

[2] https://json-schema.org
[3] https://www.openapis.org
[4] https://swagger.io
[5] https://www.postman.com/product/rest-client/

feature of JSON Schema is its support for modular and reusable schemas. Developers can reference one schema from another, promoting reusability and modular design. This is particularly useful in large-scale API ecosystems where the same data structures are reused across multiple endpoints, ensuring consistency and reducing redundancy in schema definitions.

In addition, JSON Schema undergoes continuous specification updates and has evolved through multiple versions, each adding more expressive and powerful features. Draft-04 [43], one of the most widely adopted versions, focused on basic validation. More recent versions, such as Draft-07 [71] and Draft 2020-12 [72], introduced advanced features like conditional validation (e.g., "if", "then", "else" statements) and improved support for schema reuse and modularity. This evolution demonstrates the growing importance of JSON Schema in response to the changing needs of web development and data validation, allowing developers to handle increasingly complex validation scenarios.

JSON Schema's expanding ecosystem of tools and libraries further reinforces its importance in modern development. These tools, ranging from schema validators and migration utilities to schema generation and code generation, make it easier to integrate JSON Schema into diverse environments. Developers can leverage these tools to automate data validation workflows, ensuring consistency and accuracy across different systems. [6]

Through its formalization, modular design, and rich tool support, JSON Schema can significantly improve the efficiency and scalability of systems that rely on JSON. By ensuring data validity, it minimizes errors, reduces manual validation effort, and strengthens the robustness of applications handling large and complex data sets.

1.2 Problem Statement and Motivations

JSON Schema is a schema language that encompasses many interesting features, including the use of logical operators and constraints that can sometimes be complex. Like other schema languages, it faces a variety of challenges, including both general problems that are common to schema languages used for specifications and those specific to JSON Schema. Given its increasing popularity, it is essential to address these issues to provide users with a better understanding of the language and its specificities.

Decision problems, such as *satisfiability*, are crucial for schema languages. Deciding satisfiability involves determining whether a schema admits at least one instance that conforms to its specification. Other important problems include *inclusion checking*, which assesses whether one schema is included in another, essentially verifying if the set of instances that conform to the first schema also conforms to the second. Schema *equivalence* is another decision problem whose goal is to decide whether two schemas capture exactly the same set of instances. Addressing these problems is vital for schema evolution, particularly given the widespread adoption of JSON Schema across various web APIs and systems. As these systems evolve and transition from one version to another, it often necessitates

[6] An exhaustive list of these tools can be found on the official JSON Schema website or the Awesome JSON Schema repository.

the evolution of their schemas as well. This evolution requires the revalidation of existing data to ensure it aligns and complies with the new specifications, thereby maintaining compatibility.

An interesting approach to tackle these challenges is to check inclusion between the old schema S and the new schema S', and this amounts to checking the non-satisfiability of the schema $\neg S' \wedge S$ which is expressible in JSON Schema, and this verification can be done by means of generating data that conforms to the schema $\neg S' \wedge S$. In order to check inclusion in a sound and complete fashion, the data generation approach must be sound and complete, meaning that an instance is generated if and only if the schema is satisfiable. Achieving this is particularly challenging due to the complexities inherent in the JSON Schema language.

While generating a single valid JSON instance can provide insights into the various theoretical problems that JSON Schema faces, our objective extends beyond that. We are particularly interested in how the capacity for generating JSON data can serve multiple applications and practical use cases. For instance, an important usage of multi-instance JSON Schema generation is to help schema understanding. As we will illustrate later, JSON Schema is highly expressive, and schemas can often be challenging to understand, even for experts. Having the ability to generate multiple instances of a schema can be a crucial aid for users, as examples help demonstrate various features of the schema and how they interact. This is particularly important given that, in many cases, these interactions can be subtle and difficult to grasp.

Moreover, the generation of synthetic JSON data is crucial for verifying systems through rigorous testing. By creating diverse instances that adhere to the schema, developers can assess how applications respond to various inputs, including edge cases that may not be immediately apparent. This kind of testing can help anticipate potential mismatches or performance issues, which can have significant implications for the reliability of services once deployed.

Example 2 *To illustrate how synthetic JSON data generation conforming to the specifications defined by a JSON Schema can assist testers in verifying JSON-based systems, consider the context of black-box testing. In black-box testing, the internal logic of the system or API are not relevant; the emphasis is placed on how the API behaves in response to input and output interactions. This makes it particularly suitable for web APIs where data is continually consumed and produced.*

The primary goal is to ensure that the API reliably returns expected outputs based on given inputs, adhering to the contract defined by its input and output specifications, typically represented in the form of JSON or JSON Schema.

By treating the API as a "black box", testers can concentrate on the API's behavior without needing to inspect or modify its internal code. This approach simplifies testing and enhances efficiency, allowing for the validation of whether the API correctly processes input, conforms to schema rules, and produces valid output.

This verification is crucial for detecting mismatches or performance issues that could impact the reliability of services once deployed. Furthermore, generating synthetic data

that conforms to JSON Schema specifications aids in illustrating expected API behavior, enhancing understanding, and facilitating documentation. Through black-box testing and schema-based data generation, developers can ensure consistency and correctness across various API interactions.

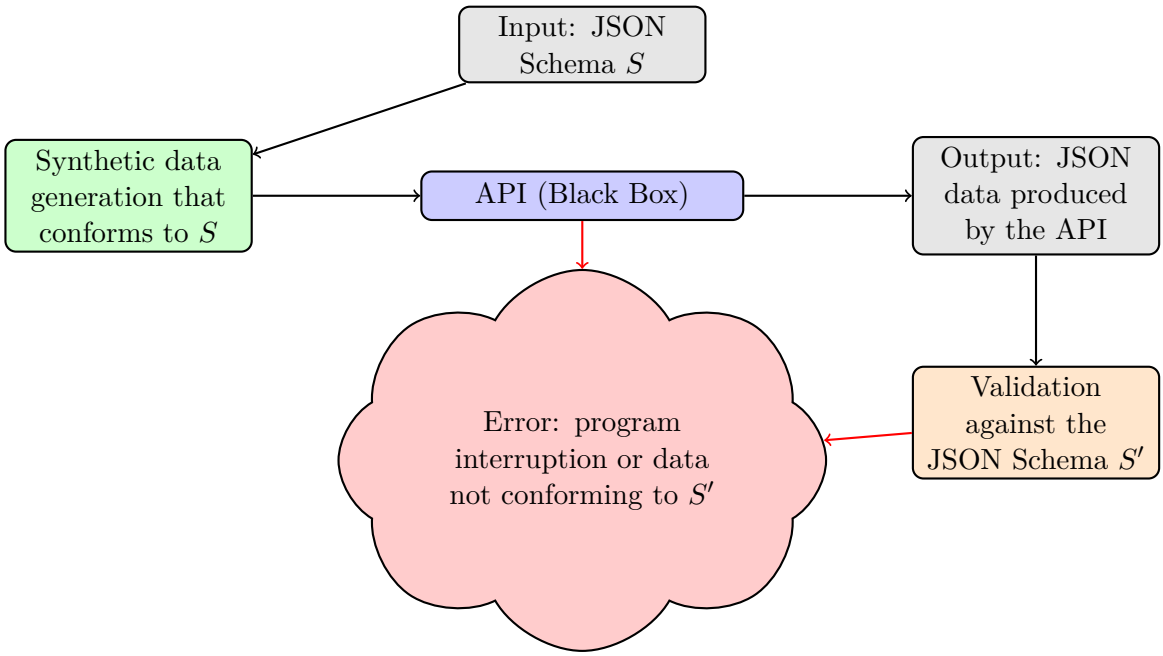


Figure 1.1: Black box testing of JSON-based web APIs

Figure 1.1 presents a typical black-box testing process for an API, where both input and output data structures can be specified using JSON Schema. This process includes the following steps:

1. *JSON Schema definition: if the API does not already use JSON Schema to specify the structure, constraints, and types for the JSON data it consumes (input) and produces (output), define two JSON Schemas, S and S', for the input and output data, respectively. Any input data provided to the API must conform to the schema S, ensuring that it is in the expected format.*

2. *Synthetic data generation: generate synthetic JSON data that conforms to S. This provides various test cases that adhere to the input schema, allowing for comprehensive testing across a range of valid and edge-case scenarios.*

3. *Black-box API testing: the generated data is sent to the API as input, while the internal workings of the API are not examined. The only concern is whether the API responds correctly according to its defined behavior.*

4. *Validation and error identification: when processing the input data, the API can either successfully produce output or, if it fails, indicate that it has encountered errors.*

Such failures suggest there are misconceptions or internal issues within the API. When output data is produced correctly, this data is then validated against the output schema S'. If the API output does not conform to the output schema, we can conclude that there are bugs or logic errors in the API's handling of the data.

In conclusion, our objective is to develop an effective solution that addresses the various challenges associated with JSON Schema through data generation techniques. While tackling theoretical, as well as practical, issues such as satisfiability, inclusion checking, and schema equivalence is crucial, we are also driven by other implications of our work related to multi-instance generation, so as to enable the production of large datasets for important practical applications, as the one illustrated in Example 2.

Our efforts aim to bridge the gap between research and practical application, ensuring our solution is relevant and beneficial across various applications. By concentrating on the formal design and the practical implementation of efficient data generation techniques for JSON Schema, we seek to contribute meaningfully to the evolution of this language and its usability in real-world situations. Through our work, we aspire to enhance how developers and organizations utilize JSON Schema in their projects, improving both efficiency and effectiveness in data and schema handling.

1.3 Thesis Contributions

In this section, we summarize our scientific contributions, highlighting the key advancements we have made in the field of JSON Schema generation and validation.

- **Definition of an experimental protocol:** establishing a comprehensive framework to evaluate the witness generation approach for data generation for JSON Schema [11, 12], this contribution establishes a structured methodology for assessing its performance and reliability. The evaluation focuses on testing both the completeness and correctness of the approach in generating JSON data that conforms to JSON Schema.

- **Development of a new generation technique:** designing a novel generation technique that balances existing methods, this contribution focuses on efficiently generating valid instances for JSON Schema. The solution specifically addresses gaps in current approaches, particularly regarding the uniqueness constraint in arrays, which is often overlooked or inadequately handled, involving solutions based on randomness that do not always guarantee a valid outcome, as well as methods that entail high computational overhead. Furthermore, it enables the generation of multiple *distinct* instances that conform to a JSON Schema, enhancing flexibility and utility in practical applications. While the approach is designed to be both efficient and correct, it is important to note that it accepts some loss of completeness, prioritizing generation speed over exhaustive instance coverage.

- **Introduction of a hybrid generation approach:** developing a hybrid approach that combines the thoroughness of the witness generation approach with the efficiency of our novel generation technique. This approach ensures both correctness and performance for JSON Schema data generation, addressing the limitations of each individual method. By integrating these two strategies, the technique strikes a balance between precision and computational efficiency, making it suitable for a broader range of schemas and use cases.

- **Implementation of a JSON Schema validator:** developing a validator for modern JSON Schema, which corresponds to the version Draft 2020-12 [72] that introduces enhanced features, increasing expressive power and adding a new layer of complexity to the validation process.

1.4 List of Publications

The following is a list of the publications that were produced during the course of this thesis, reflecting the research contributions made in the area of data generation for JSON Schema and related topics:

[11] **Witness Generation for JSON Schema.**
Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, Stefanie Scherzinger.
Proceedings of the VLDB Endowment, Volume 11, Number 12.

[13] **Validation of Modern JSON Schema: Formalization and Complexity.**
Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, Stefanie Scherzinger.
Proceedings of the ACM on Programming Languages, Volume 8, Number POPL.

[10] **A Tool for JSON Schema Witness Generation.**
Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Francesco Falleni, Giorgio Ghelli, Cristiano Landi, Carlo Sartiani, Stefanie Scherzinger.
Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021

[7] **Optimistic Data Generation for JSON Schema.**
Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo.
Transactions on Large-Scale Data- and Knowledge-Centered Systems, Volume 56.

[9] **A Test Suite for JSON Schema Containment.**
Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Yunchen Ding, Michael Fruth, Giorgio Ghelli, Carlo Sartiani, Stefanie Scherzinger.
Proceedings of the ER Demos and Posters 2021.

[6] **Overview and Perspectives for Optimistic JSON Schema. Witness Generation.**
Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo.
39ème Conférence sur la Gestion de Données, Principes, Technologies et Applications (BDA).

[5] **Example Generation for JSON Schema.**
Lyes Attouche.
37ème Conférence sur la Gestion de Données, Principes, Technologies et Applications (BDA).

1.5 Thesis Outline

The thesis is structured into several key chapters, each designed to address specific aspects of the research and contribute to a comprehensive understanding of data generation within the context of JSON Schema. Each chapter builds upon the previous ones, providing a comprehensive narrative that effectively leads the reader through the complexities of the topic.

- In Chapter 2 we delve into related works, exploring existing literature and methodologies pertinent to our research.
- Chapter 3 lays the groundwork by discussing the foundational concepts of the JSON data model and JSON Schema. It includes a presentation of their syntax and semantics, ensuring that readers have a solid understanding of the core principles that underpin our research.
- Moving forward, Chapter 4 presents an existing solution for generating data for JSON Schema. This chapter not only describes the methodology but also introduces the experimental analysis conducted as part of this thesis, emphasizing its contributions to the field.
- In Chapter 5 we introduce a novel method aimed at enhancing data generation while addressing the limitations of prior approaches. This is one of the main contributions of the thesis.
- Following this, Chapter 6 showcases the experimental results that validate the performance of our proposed method. It presents a detailed analysis of various experiments conducted to assess the method's efficiency and reliability.
- Chapter 7 shifts the focus to a hybrid approach that synergizes the strengths of previously discussed techniques. This chapter culminates in the development of a

more efficient data generation system, designed to overcome the limitations inherent in existing methods.

- Finally, Chapter 8 wraps up the thesis, summarizing key findings and suggesting future research directions, offering insights into potential avenues for further exploration.

Chapter 2

Related Work

In this chapter, we delve into the research surrounding JSON Schema and its applications, examining various studies that have explored its features, capabilities, and implications in real-world scenarios. We focus on the methodologies employed to leverage its strengths for data validation and generation. Additionally, we highlight significant contributions from the academic community that have advanced the understanding and usability of JSON Schema, providing a comprehensive overview of its current landscape and future directions.

2.1 Introduction

JSON Schema has gained traction as a robust schema language for representing and validating JSON data. It offers a wide range of features, including constraints on various JSON data types that ensure compliance with specific formats, structures, and value ranges, making it highly effective for enforcing data integrity across diverse applications.

In addition to basic constraints, JSON Schema supports logical operators like disjunction and conjunction, enabling the combination of multiple schemas for describing complex logic. This expressiveness turns out to be useful for capturing real-world situations where different rules govern the structure of data. Another powerful feature is its modularity, achieved through a *reference* mechanism that allows schema fragments to be reused across different parts, reducing redundancy and improving maintainability. The referencing mechanism is not limited to internal fragments since JSON Schema allows referencing external schemas, enabling one to integrate and validate data across schemas defined outside the current context. This feature is particularly useful in large applications and systems where schemas evolve independently but must remain interconnected in some way.

In the following sections, we will discuss various research areas related to JSON Schema, including an overview of techniques, formalization, satisfiability, and inclusion, as well as schema inference for JSON data and the usage of negation in JSON Schema. Additionally, we will explore existing data generators designed for generating JSON data that conforms to the specifications of JSON Schema.

2.2 Techniques and Challenges in JSON Schema and JSON

2.2.1 JSON Schema Formalization, Satisfiability and Inclusion

Many research studies have examined JSON Schema, aiming to understand its expressiveness power and intricacies. In [64], Pezoa et al. introduced the first formalization of JSON Schema [43], investigating the problem of validating a JSON document against a JSON Schema, and demonstrating that the validation process can be achieved in polynomial time. They also compared the language to traditional formalisms such as monadic second-order logic (MSO) and tree automata, concluding that JSON Schema cannot be expressed using either of them in the presence of the *uniqueness* constraint in arrays.

The challenges associated with JSON Schema's expressiveness and complexity were further explored in [30], where Bourhis et al. analyzed the satisfiability of schemas, proving that it is EXPTIME-complete for schemas without the uniqueness constraint in arrays and 2EXPTIME for those that include it.

Additionally, several other studies have focused on the problem of inclusion for JSON Schema. Habib et al. [46, 44, 45] made a significant contribution by developing an inclusion checker for schema containment, which determines whether one schema is a subset of another. This work is particularly important in detecting bugs and incompatibilities in JSON data. When systems like JSON-based web APIs evolve along with their schemas, this containment checker helps verify potential incompatibilities. In their study, they focused on a subset of Draft-04 [43] of JSON Schema and implemented their subschema checker [51] using the Python programming language. Their method can also be adapted to address satisfiability by checking if a schema is contained within the *empty schema*, the schema that admits no instances. If a schema is found to be contained within the empty schema, it is considered unsatisfiable.

Furthermore, the IBM ML framework LALE [27] adopts an incomplete inclusion checking algorithm for JSON Schema to enhance the safety of machine learning pipelines. This implementation highlights the importance of inclusion checking across various domains, including machine learning workflows.

In addition, other research, such as [42], conducted an empirical study on JSON Schema containment on the JSON Schema Test Suit [2], evaluating a first generation of tools in this area. The authors emphasize the need for a dedicated micro-benchmark to assess these tools' applicability to real-world schemas and identify open research opportunities with practical relevance.

2.2.2 Schema Inference for JSON Data

The exploration of schema inference and management in NoSQL databases and JSON Schema has garnered considerable attention. Researchers are investigating methodologies to enhance schema extraction, visualization, and evolution management. These efforts aim to address challenges related to data heterogeneity, structural outliers, and schema

maintenance in dynamic environments. Additionally, the development of tools and approaches focused on improving schema discovery and profiling reflects a growing interest in effectively handling JSON data and the complexities inherent in NoSQL databases.

In [58], a comparative analysis of schema inference approaches targeting the JSON format was conducted, evaluating both static features and dynamic performance, while also addressing the ongoing challenges and open problems in schema inference for NoSQL databases. Notably, in [66], Sevilla et al. pioneered a model-driven engineering approach to infer versioned schemas from NoSQL databases, with subsequent enhancements in [33] by Chillón et al. focusing on schema visualization. Scherzinger et al. in [67] introduced a comprehensive data evolution management strategy, later expanded by Klettke et al. [54] on schema extraction and evolution history reconstruction. Izquierdo and Cabot worked in [52] on schema discovery for JSON web services, while in [41], Frozza et al. presented graph-based extraction techniques for both JSON and BSON document collections.

Additionally, Baazizi et al. made significant contributions in this area with their work on schema inference from massive JSON datasets. In [14], they addressed the challenges of inferring schemas at scale, emphasizing the importance of ensuring that the inferred schemas are both accurate and scalable for large-scale systems. Their follow-up study in [16] extended this approach by introducing a method for parametric schema inference, which enhances both storage efficiency and querying operations. These works have proven instrumental in automating schema generation in environments where manually defining schemas is impractical, particularly in distributed and cloud-based systems. For further exploration of schema inference and related techniques, additional works can be found in [15, 18, 19, 17]

The relevance of schema inference in our discussion lies in its potential to enhance the JSON data generation process by acting as the reverse operation of data generation. By leveraging schema inference, we can infer a JSON Schema S' from previously generated instances and then generate new instances that satisfy the original schema S but not S' (i.e., $S \wedge \neg S'$). This approach may allow the generation of data that differs in structure and content, offering more diversity. This iterative process could improve both data generation strategies and schema understanding, making it particularly valuable in complex environments. We leave the study of this inference-based approach as future work.

2.2.3 Studying the Use of Negation in JSON Schema

Negation plays a crucial role in JSON Schema, allowing schema designers to express constraints and conditions that are not easily captured through positive assertions. The ability to specify what is not allowed within a JSON document enhances the expressiveness of schemas, making it possible to model complex data relationships and constraints. However, the use of negation also introduces complexity, particularly in terms of schema validation and interpretation.

A comprehensive investigation into the practical usage of negation within JSON Schema data modeling was conducted in [21, 20, 24]. Recognizing that negation is a logical operator often underutilized in type systems and schema description languages due to its complexity

in decision-making, the authors set out to determine its relevance in actual applications. They gathered an extensive corpus of JSON Schema documents, analyzing approximately 90,000 schemas sourced from GitHub.

Through systematic analysis, the authors quantified the patterns of negation usage and provided a qualitative assessment of schemas incorporating negation. Their findings confirmed that negation is actively utilized by JSON Schema users, following stable and common patterns, along with peculiar ones that highlight the complexity and diversity of its applications. This work not only sheds light on the practical implications of negation in JSON Schema but also opens avenues for further research into how negation can be effectively supported in tools and frameworks.

In addition to the empirical analysis of negation usage, the concept of negation closure in JSON Schema, studied by the same authors in [23, 22] has received significant attention. They explored negation closure, which refers to the property of a logical formalism that allows every negated assertion to be rewritten into a negation-free form. This property is essential for ensuring that reasoning about schemas remains consistent and manageable.

This research conducted by the authors demonstrated that, unlike many logical systems that enjoy negation closure, JSON Schema does not possess this characteristic. They examined how negation interacts with various operators within the schema, revealing that some JSON Schema operators do not exhibit negation closure, such as the constraint that restricts numbers to be multiples of a certain value. This finding highlights a critical gap in the language's expressiveness, particularly when attempting to manipulate schemas containing negation.

The implications of this lack of negation closure are significant. Without a systematic way to eliminate negation, users may encounter challenges during schema validation and interpretation. The authors proposed an extension of JSON Schema that incorporates negation closure, introducing a not-elimination algorithm designed to transform negated assertions into equivalent non-negated forms. This extension not only enhances the expressiveness of JSON Schema but also facilitates the development of tools and frameworks that can effectively handle negation.

By integrating these findings, the body of work surrounding negation in JSON Schema offers valuable insights into both its practical applications and theoretical implications. The authors underscore the importance of supporting negation in schema design, not only to improve usability but also to pave the way for future enhancements in the JSON Schema ecosystem. As we will see in Chapter 4 and Chapter 5, the involvement of negation also plays a critical role in the process of generating JSON data that conforms to a JSON Schema, further illustrating its significance in the broader context of JSON Schema applications.

2.3 Data Generators for JSON Schema

Research has increasingly focused on the challenge of generating JSON data that conforms to JSON Schema, especially within the context of testing JSON-based systems. This in-

terest is driven by the necessity for robust and valid test data that aligns with the specific structural and semantic rules defined by JSON Schema. Generating schema-compliant data is essential for ensuring that applications behave as expected, facilitating early detection of issues during development. Consequently, numerous research works, tools and methodologies have emerged to assist in the automated generation of compliant JSON data, streamlining the testing process and enhancing data integrity across various applications.

2.3.1 Literature Review

One of the first contributions in the area of data generation for JSON Schema is *jsongen*, a library developed by Fredlund et al. [40, 38] for generating test JSON data for web services testing, which is further explored in [26]. *jsongen* relies on *QuickCheck* [34], a property-based testing tool primarily used in Haskell that allows developers to define and automatically test properties of programs using randomly generated input or custom data generators when needed. *jsongen* employs *QuickCheck* to generate data that conforms to a JSON Schema by deriving a generator based on the schema's structural and validation rules. Following this, it is important to note that *jsongen* is now considered obsolete, along with open-source tools such as *Schematic-ipsum* [53] and *Json-schema-random* [61], as they primarily focused on the earlier versions of JSON Schema, specifically Draft-03 [73] and Draft-04 [43], and have not been actively maintained since then.

Since then, more recent works have been investigating the generation of data for JSON Schema, concentrating on the latest drafts that introduce more expressiveness to the language through a variety of features. Advancements in the domain of web API testing have led to the development of tools like *Schemathesis* [49], designed to detect defects in web APIs. Similar to *jsongen*, *Schemathesis* relies on property-based testing frameworks to generate data that conforms to JSON Schema constraints, while also leveraging fuzzing techniques. It employs the property-based framework *Hypothesis* [59], specifically utilizing the *hypothesis-jsonschema* library [48], which is a data generator for JSON Schema that supports versions Draft-04 [43] through Draft-07 [71] of the language. Furthermore, *Schemathesis* incorporates schema preprocessing techniques prior to invoking the generator to optimize the generation of both valid and invalid data, enhancing its effectiveness in testing the robustness of web APIs. The *hypothesis-jsonschema* library will be discussed later in the next section.

Moreover, the need for efficient generation approaches has led to other works focusing on data generation for JSON Schema, addressing various challenges in this domain. In [37], *DataGen* was initially conceived as a JSON data generator that did not incorporate JSON Schema; however, it has since evolved into a versatile tool that generates both JSON and XML data. More specifically, it now integrates JSON Schema in its process of generating JSON data. In [32], Cardoso and Ramalho introduced a new version of *DataGen*, referred to as *DataGen From Schemas*, which extends *DataGen* and is designed to support all the features of the latest version of JSON Schema, Draft 2020-12 [72]. The developed data generation workflow begins by parsing the schema into an intermediate structure using *PEG.js* [56]. This intermediate structure is then translated into a predefined Domain

Specific Language (DSL). Finally, *DataGen* takes over by processing the DSL model to generate the JSON data.

An important aspect of our interest in this research work is how the uniqueness constraint in arrays is addressed. The adopted strategy involves regenerating values when a duplicate is detected, allowing up to 10 attempts before stopping the regeneration process, where the last generated value is considered, potentially violating the uniqueness constraint. This method relies on randomness for resolution. While this approach may work in some scenarios, it is not efficient, particularly when the schemas being processed have a finite and limited domain size, where the regeneration of the same values is more probable. Depending on randomness to enforce uniqueness can lead to significant computational costs, especially when dealing with complex schemas. Additionally, the inherent lack of control in random generation may result in failures to produce arrays that satisfy the uniqueness constraint, despite the existence of simpler, more effective solutions.

It is worth noting that the work does not provide experimental results to evaluate the performance or efficiency of the tool, which could offer valuable insights into its practical applicability and effectiveness in real-world scenarios.

A more recent study on the generation of data for JSON Schema was introduced in [63], where the authors present *Fences*, a tool that generates JSON data for JSON Schema using an approach based on flow graphs. *Fences* supports JSON Schema Draft 2020-12 [72] with all its features, targeting the generation of both valid and invalid data, where the latter can potentially be used to detect the *provenance* of defective data in schemas, leading to the identification of problematic fragments of the schemas. The generation process of *Fences* includes a preprocessing phase, a step that many generators generally rely on to simplify schemas by effectively eliminating conjunctions and negations, thereby making the schema easier to generate. Furthermore, the authors introduced a *coverage metric* that assesses the effectiveness of generators, providing a qualitative measure for the generated data.

Fences was evaluated against *hypothesis-jsonschema* [48], using both the official JSON Schema test suite [2] and a real-world schema collection from Industry 4.0 [62]. The paper outlines that *Fences* achieved slightly higher schema coverage even though it generated fewer JSON instances than *hypothesis-jsonschema*, which makes it faster overall.

Further studies on data generation for JSON Schema include [11], where an overview of the method will be covered in Chapter 4, as the experimental protocol conducted in this research was one of the contributions of the thesis. This research introduced *witness generation*, a sound and complete approach for instance generation for JSON Schema. A more in-depth discussion is provided in [12], which presents various algorithms and proofs concerning the soundness and completeness of the method for generating witnesses for JSON Schema. Additionally, our work in [6, 7], which introduces a new efficient technique for data generation, forms a core part of the thesis and will be presented in detail in Chapter 5.

Another notable contribution is discussed in [55], which focuses on the challenge of translating *ECMA-262* regular expressions [39] used in JSON Schema into *Brics* syntax [60]. This work highlights the translation mechanism employed in [11, 12] for generating string data using the *Brics* library.

2.3.2 Overview of Open-Source Tools

In this section, we provide an overview of open-source tools that perform data generation for JSON Schema, focusing on those that are actively maintained and regularly updated. These tools play a crucial role in supporting schema-based testing and validation by automating the creation of JSON instances. We examine their core features, and limitations, in order to identify potential gaps that motivate the need for more efficient and adaptable data generation techniques.

JSON Generator (*DG*)[28]. This tool is developed in *Java* and is currently at version 0.4.7. An online version is provided [1], and it supports JSON Schema Draft-07 [71] as well as earlier versions.

It adopts a two-phase approach: an initial generation phase followed by a fixing phase, during which the generated data is validated using an external validator [29]. If validation errors occur, the tool attempts to repair the generated instance. This repair process stops if the same error is encountered twice or if the instance becomes fixed and valid. However, the fixing approach is somewhat naive, as it involves redoing the generation to address issues that may not have been captured during the initial generation phase.

Negation is not addressed, as it is ignored during the initial generation phase, and in the fixing phase, the tool handles the error of not satisfying negation by returning a default value of zero. Moreover, hen encountering exclusive disjunctions, where instances must satisfy exactly one schema, the tool naively resolves these by randomly selecting a branch of the schema to satisfy, similar to how disjunctions are handled.

Furthermore, the generation of arrays in the presence of the uniqueness constraint is mentioned but not adequately addressed. If a value has already been generated, it will not be added to the array, effectively moving to the next item to generate, which can result in arrays that are smaller than desired when a minimal bound is imposed. In the fixing phase, when a uniqueness error is detected, the array is simply filtered to remove duplicates, potentially leading to issues with size constraints. This special case highlights the limitations of the repair process, which can result in infinite cycles of reparation, as errors can be interdependent; fixing the uniqueness constraint can violate the minimal bound, and adding items to satisfy the minimal bound can lead to violations of the uniqueness constraint.

One notable feature of the tool is its ability to generate string values using the *Randexp* library [2]. This adds a layer of randomness that improves the quality of the generated data; however, it may occasionally produce invalid values. Additionally, the tool incorporates hyperparameters that give users control over various preferences, further enhancing the customization of the data generation process

Overall, despite the limitations outlined, the tool provides a robust framework for JSON data generation that conforms to JSON Schema, as will be detailed in the experimental analysis of Chapter 4.

[1] https://tryjsonschematypes.appspot.com/#generate

[2] https://github.com/fent/randexp.js

JSON-Schema-faker (*JSF*)[1]. This generator, is written in *JavaScript*, supports Draft-04 [43] of JSON Schema and is currently at version 0.5.6. An online version is also provided [3] (for the version 0.5.4 of the tool), along with a command-line interface (CLI) [4]. Additionally, a *Python* version of the tool has been developed [5].

Differently from the previous tool, this one does not include a fixing mechanism. Nevertheless, it has comparable limitations, as negation is not addressed, along with other object-related constraints such as dependencies between fields. It also struggles when implicit conjunctions between constraints are present. In addition, dealing with the uniqueness constraint in arrays is limited and managed similarly to how it is handled in *DataGen From Schemas* [32]. The approach involves regenerating values until a distinct one is found, allowing for up to 100 attempts.

Similar to the *DG* tool, it also incorporates randomness, as the generation of string values relies on the same *Randexp* library[2]. It also utilizes other libraries such as *Faker.js*[6] and *Chance.js*[7], which are used not only for string values but also for numeric values.

In line with the previous tool, it allows users to specify a broader range of options to suit their specific needs, such as overriding certain constraints defined by the schema (e.g., size of arrays and the depth level of references when resolved). Nevertheless, these customization options can sometimes become unwieldy and lead to the generation of invalid values.

JSF serves as a versatile solution for developers seeking to simulate realistic datasets based on defined schemas. It is comparable to the *DG* tool, though it comes with a few additional limitations.

JSON-everything (*JE*) [36]. The library we discuss here encompasses various tools for querying and managing JSON data written in *C#*, along with tools dedicated to JSON Schema, such as a tool for schema generation from data. It also includes a data generator for JSON Schema [8]. Currently at version 2.2.0, the generator also has a web-based version available for users [9].

While its documentation does not explicitly state which version of JSON Schema it supports, it references keywords like if/then/else that were introduced in Draft-07 [71], suggesting compatibility with this version. However, it does not cover all features; for example, constraints on dependencies between fields in the object type are not supported. Additionally, it does not support the generation of arrays with the uniqueness constraint.

The generator employs the *Bogus* library [10] for generating string values in schemas where there are no specific conditions or restrictions applied to the string type. Conversely,

[3] https://json-schema-faker.js.org/
[4] https://github.com/oprogramador/json-schema-faker-cli
[5] https://github.com/ghandic/jsf
[6] https://fakerjs.dev/
[7] https://chancejs.com/
[8] GitHub repository of the data generation tool.
[9] https://json-everything.net/json-schema
[10] https://github.com/bchavez/Bogus

when restrictions are defined, it utilizes the *Fare* library [11] for generating values that must adhere to the specified constraints. However, regular expressions not supported by the latter library result in invalid generation, and it also does not handle logical operations involving regular expressions.

The tool is significantly limited compared to existing alternatives, particularly because of the limitations associated with string generation.

Hypothesis-jsonschema (*HJS*)[48]. This data generator is implemented in *Python* and is currently at version 0.23.1. It supports Draft-04 [43], Draft-06 [70], and Draft-07 [71] of JSON Schema, covering nearly all the features of these drafts, with the exception of recursive schema references.

This generator acts as a bridge between JSON Schema and property-based testing, extending the capabilities of *Hypothesis* [59] [12].

Hypothesis is a powerful *Python* library designed for generating unit tests based on specifications. It automatically generates input data for tests by defining strategies, which describe the types of data to be used. *Hypothesis* represents the possible values for these strategies as a tree structure, especially for nested or complex data types. Using heuristics, it explores this structure to generate a wide range of data, including edge cases. The generated data is random, which helps ensure thorough test coverage by exposing unexpected behaviors or edge case scenarios that might otherwise be missed.

HJS acts as a translator between JSON Schema and *Hypothesis*, converting JSON schemas into corresponding search strategies that generate data conforming to the schema's constraints. It includes a preprocessing phase to simplify the schema, ensuring more efficient and accurate data generation. For instance, during this phase, *HJS* combines strategies when handling complex schema elements like conjunctions in the original JSON Schema. Additionally, the tool supports advanced features of JSON Schema, such as enforcing uniqueness constraints on array elements, though its handling of some of these features may be limited in certain cases.

Despite its strengths, *HJS* does have some limitations. Its reliance on property-based testing through *Hypothesis* can lead to higher computational costs, especially when dealing with highly complex schemas. Another significant limitation is its lack of support for recursive references, a common feature in many real-world JSON Schemas.

Containment checker (*CC*)[51, 46, 44, 45]. The last tool we describe is *jsonsubschema*, briefly introduced in Section 2.2.1. While not a data generator, it remains an important part of the JSON Schema ecosystem and is used in our study, as will be presented in Chapter 4.

This tool checks schema inclusion, returning a simple *true* or $false$ to indicate whether one schema is included in another, without generating examples or instances. Currently at version 0.0.7, *jsonsubschema* supports only Draft-04 [43] schemas and was developed in

[11] https://github.com/moodmosaic/Fare

[12] https://hypothesis.readthedocs.io/en/latest/ and https://hypothesis.works/

Python. The tool has certain limitations in addition to only supporting the aforementioned version of JSON Schema: key language features, such as recursion and negation, are not handled.

Despite its limitations in handling the full language and its lack of data generation capabilities, the tool introduced key operations within its inclusion checking process, particularly schema preprocessing, which involves schema manipulations that are directly applied in our current work. Therefore, it was important to highlight the tool here and point out its capabilities.

2.4 Conclusion

In this chapter, we discussed how JSON Schema has been studied in relation to various problems associated with schema languages, such as satisfiability and inclusion. The research studies conducted on those aspects underscore the complexity and expressive power of the language, highlighting the challenges involved when manipulating JSON Schema.

We also explored schema inference techniques that enhance the extraction and management of schemas for JSON data. These methods are crucial for improving schema design and facilitating data discovery, and we also highlighted a scenario that relates to our concern of data generation, and how schema inference can possibly be used in our context.

Negation in JSON Schema was examined, emphasizing its significance in defining more complex constraints and in increasing the expressiveness of the language.

Lastly, we addressed existing tools that perform data generation for JSON Schema, which aim at automating the creation of data instances, and outlined how they are conceived and the different techniques used for achieving generation.

Overall, this chapter illustrates the advancements made in understanding and applying JSON Schema techniques, while also emphasizing the need for ongoing research to tackle existing challenges and improve practical applications within the JSON Schema ecosystem.

Chapter 3

Preliminaries

In this chapter, we explore the foundational concepts needed to understand the upcoming sections of this document. We begin with a detailed introduction to the JSON data model, the widely used format for structuring and exchanging data on the web. Next, we provide an in-depth overview of JSON Schema, highlighting its key features and capabilities. Finally, we examine the syntax and semantics of JSON Schema in detail, laying the groundwork for a comprehensive understanding of its use and validation mechanisms. To illustrate these concepts, we will also present an example demonstrating how validation is performed against a schema, showing how different keywords guide the validation process and affect the outcome.

3.1 The JSON Data Model

JSON (JavaScript Object Notation) is a widely-used format for storing and exchanging data in web applications, defined by standards such as ECMA-404 [35] and IETF RFC 8259 [31]. Its syntax uses braces, brackets, commas, and colons to represent key-value pairs clearly, making it suitable for diverse applications. JSON is language-independent, allowing various programming languages to parse and generate JSON data easily. Its support for a hierarchy of nested objects and arrays enables flexible data descriptions, contributing to its popularity in modern programming environments.

Figure 3.1 presents the data types supported by JSON, where values can be of several types: `null`, `boolean`, `number`, `string`, `array`, or `object`. Arrays are ordered lists of JSON values, separated by commas, and enclosed in brackets. They can be empty and may contain elements of any type, including other arrays and objects. Objects consist of a (potentially empty) set of key-value pairs, separated by commas and delimited by opening and closing braces. While it is technically possible to have duplicate keys in an object, this is strongly discouraged, as it can lead to unexpected behaviors. In our context, keys l_i must be unique and are of type `string`, while the values can be any JSON value.

$d \in \mathsf{Num}, s \in \mathsf{Str}, n \in \mathsf{Int}, n \geq 0, l_i \in \mathsf{Str}$

$J ::= \texttt{null} \mid true \mid false \mid d \mid s \mid [J_1, \ldots, J_n] \mid \{ l_1 : J_1, \ldots, l_n : J_n \} \quad i \neq j \Rightarrow l_i \neq l_j$

Figure 3.1: The JSON data model

In the grammar described in the figure, `null` is the only value possible for the `null` type, representing an empty value. The `boolean` type only allows the values *false* and *true*. Num represents the set of all possible numerical values, which must consist of finite and well-defined numbers, as numeric values that cannot be represented as sequences of digits (such as infinity) are not permitted. Notably, numbers can also be represented in exponential notation, so `1e3` is a valid JSON number. The set of all possible string values is denoted by Str, where a `string` value is defined as a sequence of zero or more Unicode characters, delimited by double quotation marks, and supports a backslash escaping syntax for special characters. For instance, to represent the character '\', we use "\\" in JSON.

Example 3 *Consider the following JSON* `object`*, consisting of four key-value pairs. The value associated with the key* `"name"` *is of type* `string`*. For the* `"age"` *key, it holds an* `integer` *value. The value of the key* `"address"` *is of type* `object` *and is composed of 3 key-value pairs, all of which are of* `string` *type. Lastly, the value of* `"hobbies"` *is an* `array` *containing strings.*

```
{ "name": "John Doe",
  "age": 24,
  "address": {
    "street": "4 Pl. Jussieu",
    "city": "Paris",
    "zipCode": "75005"
  },
  "hobbies": ["Hiking","Cooking","Painting"]
}
```

3.2 Overview of JSON Schema

In this section, we provide an overview of the key aspects of JSON Schema, focusing on a subset of Draft 2019 [69] and earlier versions, as these are the most widely adopted in practice. While more recent versions such as Draft 2020-12 [72] have introduced advanced features, we limit our scope to the widely used functionality in order to maintain relevance and compatibility with common implementations.

A JSON Schema specification is itself a JSON document that defines various *typed assertions* to enforce constraints on specific JSON structures (e.g., objects, arrays, etc.). These typed assertions are typically expressed using the `type` keyword, which indicates

the category of JSON data they refer to. In this section, we will explore the key types of assertions and their practical usage, before delving into their formal semantics later on.

- **Base value assertions:** these define constraints for basic types such as strings and numbers. For strings, regular expressions can be used to describe patterns (`pattern`), and their length can be restricted by specifying minimum and maximum values (`minLength`, `maxLength`). For numbers, constraints can be defined through intervals (`minimum`, `maximum`) or by specifying multiples (`multipleOf`).

- **Array assertions:** these are used to specify the types of elements that can appear in an array using `items` and `prefixItems`, and to restrict the size of the array using an interval (`minItems`, `maxItems`). It is also possible to require a specific number of elements to satisfy a schema using the constraint `contains`, along with `minContains` and `maxContains`. Ensuring the uniqueness of elements in an array can be achieved through the `uniqueItems` constraint.

- **Object assertions:** these allow developers to define the structure of objects by specifying the schema for individual properties (`properties`) or by using regular expressions to capture property names (`patternProperties`). Required properties are defined through `required`, and constraints on the number of properties can be set with `minProperties` and `maxProperties`. It is also possible to enforce that property names follow a specific convention using `propertyNames`. Additionally, handling undefined properties (`additionalProperties`) and expressing dependencies between properties (`dependentRequired` and `dependentSchemas`) are key features for managing complex object structures.

Beyond these basic capabilities of defining constraints on the different JSON data types, JSON Schema supports the construction of complex expressions through boolean operators such as disjunction (`anyOf`), conjunction (`allOf`), negation (`not`), and exclusive disjunction (`oneOf`). Furthermore, JSON Schema provides mechanisms for defining reusable schema fragments through `$defs` and `$ref`, facilitating modular and maintainable schema designs, especially in large-scale systems. As a result, it is also possible to define schemas that reference themselves, enabling the creation of recursive structures.

Example 4 *To give an overview of assertions that can be expressed in JSON Schema, consider the following schema. It restricts valid values to objects containing a property* `"a"` *whose value must be a string that matches the pattern* `"a(c|e)*"`*. Additionally, it allows for the presence of other properties with unrestricted values.*

```
{ "type": "object", "required": ["a"],
  "properties": { "a": { "type": "string", "pattern": "a(c|e)*" } },
  "additionalProperties": true
}
```

While the previous example is straightforward, schemas can become considerably more intricate when overlapping constraints or dependencies are introduced. For instance, the `patternProperties` keyword allows constraints to be applied to properties whose names match a given regular expression, potentially overlapping with constraints defined in `properties`.

Example 5 *The following schema extends the previous one by introducing a* `patternProperties` *assertion. Properties whose names match the regular expression* `"a+"` *must have a minimum length of 3. This results in an overlap with the previous assertion for the property* `"a"`*, which must satisfy both the minimum length constraint and the pattern* `"a(c|e)*"`*.*

```
{ "type": "object", "required": ["a"],
  "properties": { "a": { "type": "string", "pattern": "a(c|e)*" } },
  "patternProperties": { "a+": { "minLength": 3 } }
}
```

The above example, while simple, is very helpful in highlighting the limitations of existing implementations of data generation for JSON Schema, which are unable to handle overlapping constraints due to their limited support for *conjunction*.

Overlapping constraints can also occur in array assertions, as shown in the following example.

Example 6 *The following schema describes arrays with a minimum of four and a maximum of ten items. Each item in the array must be a string with a minimum length of two characters, and at least one of these strings must match the pattern* `"a(b|c)a"`*.*

```
{ "type": "array", "minItems": 4, "maxItems": 10,
  "items": { "type": "string" , "minLength": 2 },
  "contains": { "pattern": "a(b|c)a" }
}
```

To illustrate the application of boolean operators in JSON Schema, consider the following schema, which demonstrates the use of both conjunction and disjunction.

Example 7 *The following schema restricts valid values to objects that must either contain a property* `"a"` *with a string value or a property* `"b"` *with an integer value.*

```
{ "type": "object",
  "properties": { "a": { "type": "string" }, "b": { "type": "integer" } },
  "anyOf": [ { "required": ["a"] }, { "required": ["b"] } ],
  "allOf": [
    { "properties": { "a": { "minLength": 3 }, "b": { "minimum": 10 } } }
  ],
  "additionalProperties": false
}
```

This example showcases the use of boolean operators like `anyOf` (disjunction), requiring either property `"a"` or `"b"` to be present, and `allOf` (conjunction), ensuring that if present, `"a"` must be a string of at least three characters, and `"b"` must be an integer greater than or equal to `10`.

3.3 Syntax and Semantics of JSON Schema

In this section, we will explore the syntax and semantics of JSON Schema, concentrating on the specific assertions relevant to our study. As mentioned in the previous section, we only focus on a subset of the version Draft 2019 [69], by excluding keywords that introduce extra complexity to validation, such as `$anchor`, `$dynamicAnchor`, `$dynamicRef`, as well as keywords whose semantics rely on annotations, namely unevaluatedProperties and unevaluatedItems.

Figure 3.2 highlights the assertions included in our study. As specified in the grammar, a valid JSON Schema S can be a boolean `true` or `false`, or it can also be a JSON object, which may be empty; in this case, it is equivalent to the schema `true`. In the grammar, Num represents the set of all possible numerical values, and Str represents the set of all possible string values. The variable i is a positive integer.

The value of `$ref`, denoted as *uri*, is of type `string`. In this context, we have excluded the feature of referencing external schemas and are focusing solely on referencing fragments of the schema itself in different places. Thus, *uri* consists of a path within the schema, where the root is represented with `#`. For example, `"#/a/b"` is a valid URI that allows for referencing the schema of `"b"`, which is located inside the schema of `"a"` that is situated at the root level.

The `format` [1] constraint enables the identification of certain string values. Its value for is a string that can represent a wide range of defined values, such as `"date"`, `"email"`, and so on. Finally, *JVal* represents the set of all valid JSON values.

Remark 1 *In our representations, we use {|...|} to denote a set of values, while the notation { ... } is specifically reserved for representing JSON objects.*

We define the semantics of JSON Schema in an operational manner based on prior work in [13], which studies validation for the latest version, Draft 2020-12 of JSON Schema [72]. We adapt the set of validation rules established in that context to suit our specific needs. The validation of an instance J against a schema S, yielding a boolean value, is represented through a main judgment and two auxiliary judgments. These judgments are interdependent and guided by the syntax of JSON Schema, facilitating a clear and structured approach to understanding the semantics of the different assertions.

- The *schema judgment* is the main one, denoted $J \; ? \; S \mapsto r$, captures the validation of J against S and returns the boolean value r;

[1] The complete list of allowed values for format can be accessed here.

$b \in \{\!|false, true|\!\}, q \in \mathsf{Num}, i \in \mathbb{N}, k \in \mathsf{Str}, uri \in \mathsf{Str}, for \in \mathsf{Str}, p \in \mathsf{Str}, J \in JVal$

$$
\begin{array}{rcl}
Tp & ::= & \mathsf{object} \mid \mathsf{number} \mid \texttt{integer} \mid \mathsf{string} \mid \mathsf{array} \mid \mathsf{boolean} \mid \texttt{null} \\
S & ::= & \texttt{true} \mid \texttt{false} \mid \{\ K\ (,K)^*\ \} \mid \{\ \} \\
K & ::= & \texttt{type} : Tp \mid \texttt{type} : [\,Tp_1, \ldots, Tp_n\,] \mid \texttt{minimum} : q \mid \texttt{maximum} : q \\
 & & \mid \texttt{exclusiveMinimum} : q \mid \texttt{exclusiveMaximum} : q \mid \texttt{multipleOf} : q \\
 & & \mid \texttt{pattern} : p \mid \texttt{minLength} : i \mid \texttt{maxLength} : i \mid \texttt{format} : for \\
 & & \mid \texttt{minProperties} : i \mid \texttt{maxProperties} : i \mid \texttt{required} : [\,k_1, \ldots, k_n\,] \\
 & & \mid \texttt{properties} : \{\,k_1 : S_1, \ldots, k_m : S_m\,\} \mid \texttt{additionalProperties} : S \\
 & & \mid \texttt{patternProperties} : \{\,p_1 : S_1, \ldots, p_m : S_m\,\} \mid \texttt{propertyNames} : S \\
 & & \mid \texttt{dependentSchemas} : \{\,k_1 : S_1, \ldots, k_n : S_n\,\} \\
 & & \mid \texttt{dependentRequired} : \{\,k_1 : [\,k_1^1, \ldots, k_{o_1}^1\,], \ldots, k_m : [\,k_1^n, \ldots, k_{o_n}^n\,]\,\} \\
 & & \mid \texttt{minItems} : i \mid \texttt{maxItems} : i \mid \texttt{minContains} : i \mid \texttt{maxContains} : i \\
 & & \mid \texttt{uniqueItems} : b \mid \texttt{items} : S \mid \texttt{prefixItems} : [\,S_1, \ldots, S_n\,] \mid \texttt{contains} : S \\
 & & \mid \texttt{\$defs} : \{\,k_1 : S_1, \ldots, k_n : S_n\,\} \mid \texttt{\$ref} : uri \mid \texttt{anyOf} : [\,S_1, \ldots, S_n\,] \\
 & & \mid \texttt{allOf} : [\,S_1, \ldots, S_n\,] \mid \texttt{oneOf} : [\,S_1, \ldots, S_n\,] \mid \texttt{not} : S \\
 & & \mid \texttt{if} : S \mid \texttt{else} : S \mid \texttt{then} : S \mid \texttt{const} : J_c \mid \texttt{enum} : [\,J_1, \ldots, J_n\,]
\end{array}
$$

Figure 3.2: Grammar of normalized JSON Schema Draft 2019

- The auxiliary judgments, *keyword judgment* denoted $J \ ?\ K \to r$, and *keywords-list judgment* denoted $J \ ?\ \vec{K} \Rightarrow r$, are meant to capture the validation of J over a single keyword K or a list of keywords $\vec{K}$, respectively. This latter judgment is useful for defining the semantics of keywords whose semantics depends on the evaluation of adjacent keywords like in the case of `additionalProperties`.

Figure 3.3 outlines the rules for the *schema judgments* and *keywords-list judgments*. The rules (`false-schema`) and (`true-schema`) correspond to the boolean schemas `false` and `true` respectively. The validation rules for these schemas are straightforward. The `false` schema represents the universally unsatisfiable schema, which does not admit any JSON instance, and as a result, any validation against this schema will always yield $false$ (denoted $\mathcal{F}$). Conversely, the `true` schema is universally satisfiable, meaning it admits every possible JSON value, and validation against it will always yield $true$ (denoted $\mathcal{T}$).

The rule (KwList-(n+1)) represents the process of validating an instance against a set of schema keywords, incrementally building upon the results of validating the instance against each keyword in the list. It follows an inductive structure, where the overall validation result is derived by taking the conjunction of the validation results from each individual keyword in the list.

The rule (emptyKwList) is trivial and serves as the base case for the induction of the previous rule, where the absence of constraints allows for any instance to be valid.

Rules (schema-true) and (schema-false) rely on the validation of the list of keywords contained within the schema. If the keywords-list validation succeeds and returns $\mathcal{T}$, then the validation of an instance against the schema also yields $\mathcal{T}$. Conversely, if the keywords-list validation fails and returns $\mathcal{F}$ ($false$), the validation of the instance against the schema also results in $\mathcal{F}$.

$$J \ ? \ \texttt{false} \mapsto \mathcal{F} \ (\texttt{false-schema}) \qquad \frac{J \ ? \ \vec{K} \Rightarrow r_l \qquad J \ ? \ K \to r}{J \ ? \ (\vec{K} + K) \Rightarrow r_l \wedge r} \ (\mathsf{KwList\text{-}(n{+}1)})$$

$$J \ ? \ \texttt{true} \mapsto \mathcal{T} \quad (\texttt{true-schema})$$

$$\frac{J \ ? \ [\, K_1, \ldots, K_n \,] \Rightarrow \mathcal{T}}{J \ ? \ \{\, K_1, \ldots, K_n \,\} \mapsto \mathcal{T}} \ (\mathsf{schema\text{-}true}) \qquad \frac{J \ ? \ [\, K_1, \ldots, K_n \,] \Rightarrow \mathcal{F}}{J \ ? \ \{\, K_1, \ldots, K_n \,\} \mapsto \mathcal{F}} \ (\mathsf{schema\text{-}false})$$

$$J \ ? \ [\,] \Rightarrow \mathcal{T} \quad (\mathsf{emptyKwList})$$

Figure 3.3: Schema and keywords-list jugements

In the following, we will present the semantics of the different keywords listed in the grammar. We follow the terminology of the JSON Schema standard by distinguishing between terminal assertions, boolean applicators, and structural applicators.

3.3.1 Terminal Assertions

These assertions assemble keywords that do not contain any sub-schema. First, we have the *type-uniform* ones that do not make a differentiation based on the type of the instances; these are `enum`, `const`, `type` : $[\mathsf{Tp}_1, \ldots, \mathsf{Tp}_n]$, and `type` : Tp. Then, we have the implicative keywords, which are specific to a certain type T and always return $\mathcal{T}$ ($true$) when applied to instances whose types differ from T. To capture the semantics of a terminal assertion `kw` : J' when applied to an instance J, we consider two validation rules ($kw\mathsf{Triv}$) and (kw). The former rule applies to the trivial case of the typed assertions, while the latter applies to both typed assertions in the general case and the type-uniform assertions. For the type-uniform ones, the condition $TypeOf(J) = TypeOf(kw)$ is ignored, where $TypeOf(J)$ extracts the type of J, while $TypeOf(kw)$ indicates the type to which kw refers. Table 3.1 presents all the terminal assertions and specifies the condition verified for each assertion when applied to the instance J.

$$\frac{TypeOf(J) \neq TypeOf(kw)}{J \ ? \ kw : J' \to \mathcal{T}} \qquad (kw\mathsf{Triv})$$

$$\frac{TypeOf(J) = TypeOf(kw) \qquad r = \mathsf{cond}(J, kw{:}J')}{J \ ? \ kw{:}J' \to r} \qquad (kw)$$

assertion kw:J'	$TypeOf(kw)$	cond(J,kw:J')
`enum` : $[J_1, \ldots, J_n]$	*no type*	$J \in \{\!\lvert J_1, \ldots, J_n \rvert\!\}$
`const` : J_c	*no type*	$J = J_c$
`type` : Tp	*no type*	$TypeOf(J) = \mathsf{Tp}$
`type` : $[\mathsf{Tp}_1, \ldots, \mathsf{Tp}_n]$	*no type*	$TypeOf(J) \in \{\!\lvert \mathsf{Tp}_1, \ldots, \mathsf{Tp}_n \rvert\!\}$
`exclusiveMinimum`: q	number	$J > q$
`exclusiveMaximum`: q	number	$J < q$
`minimum`: q	number	J $\geq$ q
`maximum`: q	number	J $\leq$ q
`multipleOf`: q	number	$\exists i \in \mathsf{Int}.\ J = i \times q$
`pattern`: p	string	$J \in L(p)$
`minLength`: i	string	$\lvert J \rvert \geq i$
`maxLength`: i	string	$\lvert J \rvert \leq i$
`minProperties`: i	object	$\lvert J \rvert \geq i$
`maxProperties`: i	object	$\lvert J \rvert \leq i$
`required` : $[\,k_1, \ldots, k_n\,]$	object	$\forall i.\ k_i \in \mathtt{names}(J)$
`uniqueItems`: *true*	array	$J = [J_1, \ldots, J_n]$ with $n \geq 0$ $\wedge\ \forall i, j.\ 1 \leq i \neq j \leq n \Rightarrow J_i \neq J_j$
`uniqueItems`: *false*	array	T
`minItems`: i	array	$\lvert J \rvert \geq i$
`maxItems`: i	array	$\lvert J \rvert \leq i$
`format`: *format*	string	$J \in L(format)$
`dependentRequired` : $\{\ k_1 : [\,k_1^1, \ldots, k_{o_1}^1\,]$ $\ldots, k_n : [\,k_1^n, \ldots, k_{o_n}^n\,]\ \}$	object	$\forall i \in \{\!\lvert 1 \ldots n \rvert\!\}.$ $k_i \in \mathtt{names}(J)$ $\Rightarrow \{\!\lvert k_1^i, \ldots, k_{o_i}^i \rvert\!\} \subseteq \mathtt{names}(J)$

Table 3.1: Terminal assertions: conditions

3.3.2 Boolean Applicators

The semantics of the boolean applicators is straightforward and rely on applying the corresponding logical connectives to the validation results of each schema.

$$\frac{J\ ?\ S \mapsto r}{J\ ?\ \mathtt{not} : S \to \neg r} \quad (\mathsf{not})$$

$$\frac{\forall i \in \{\!\lvert 1 \ldots n \rvert\!\}.\ \ J\ ?\ S_i \mapsto r_i \qquad r = \vee(\{\!\lvert r_i \rvert\!\}^{i \in \{\!\lvert 1 \ldots n \rvert\!\}})}{J\ ?\ \mathtt{anyOf} : [S_1, ..., S_n] \to r} \quad (\mathsf{anyOf})$$

$$\frac{\forall i \in \{\!\lvert 1 \ldots n \rvert\!\}.\ J\ ?\ S_i \mapsto r_i \qquad r = \wedge(\{\!\lvert r_i \rvert\!\}^{i \in \{\!\lvert 1 \ldots n \rvert\!\}})}{J\ ?\ \mathtt{allOf} : [S_1, ..., S_n] \to r} \quad (\mathsf{allOf})$$

$$\frac{\forall i \in \{\!|1 \dots n|\!\}.\ J\ ?\ S_i \mapsto r_i \qquad r\ =\ (\ |\{\!|i\ \mid\ r_i = \mathcal{T}|\!\}| = 1\ )}{J\ ?\ \texttt{oneOf} : [S_1, ..., S_n] \to r} \quad \text{(oneOf)}$$

3.3.3 Structural Applicators

We differentiate between three families of applicators: those related to *objects, arrays*, and others. For each family, we further distinguish between independent applicators, whose semantics are described using the *keyword* judgment, and dependent applicators, whose semantics depend on the evaluation of adjacent applicators and are hence captured using the *keywords-list* judgment.

Object Keywords. These keywords are specific to the object type, hence, their semantics are trivial and they are always satisfied when applied to instances that are not of type object. They are composed of sub-schemas, thus requiring verification of the internal structures of the instance.

First, to illustrate the trivial behavior of these keywords when encountering an instance that is not of type object, we present the trivial rule for the `properties` keyword, that is:

$$\frac{\mathit{TypeOf}(J) \neq \mathsf{object}}{J\ ?\ \texttt{properties} : \{\ k_1 : S_1, \dots, k_m : S_m\ \} \to \mathcal{T}} \quad \text{(propertiesTriv)}$$

The (properties) rule captures the non-trivial case, that is, when J is an object of the form $J = \{\ k'_1 : J_1, \dots, k'_n : J_n\ \}$. This rule states that, in order for J to validate $\texttt{properties} : \{\ k_1 : S_1, \dots, k_m : S_m\ \}$, every value J_i whose key k'_i *matches* a keyword $k_j : S_j$, must validate the schema S_j. This rule relies on collecting the set of indexes of matching pairs in order to perform validation whose result is combined using conjunction.

$$\frac{\begin{array}{c} J = \{\ k'_1 : J_1, \dots, k'_n : J_n\ \} \qquad \{\!|(i_1, j_1), \dots, (i_l, j_l)|\!\} = \{\!|(i, j)\ \mid\ k'_i = k_j|\!\} \\ \forall q \in \{\!|1 \dots l|\!\}.\ J_{i_q}\ ?\ S_{j_q} \mapsto r_q \qquad r = \wedge(\{\!|r_q|\!\}^{q \in \{\!|1 \dots l|\!\}}) \end{array}}{J\ ?\ \texttt{properties} : \{\ k_1 : S_1, \dots, k_m : S_m\ \} \to r} \quad \text{(properties)}$$

The (patternProperties) rule generalizes the previous one by considering key-pattern *membership* expressed by $k'_i \in L(p_j)$ where $L(p_j)$ denotes the language of the pattern p_j, instead of *exact* label-keyword matching. The same conjunctive semantics used in the previous rule is adopted here.

$$\frac{\begin{array}{c} J = \{\ k'_1 : J_1, \dots, k'_n : J_n\ \} \qquad \{\!|(i_1, j_1), \dots, (i_l, j_l)|\!\} = \{\!|(i, j)\ \mid\ k'_i \in L(p_j)|\!\} \\ \forall q \in \{\!|1 \dots l|\!\}.\ J_{i_q}\ ?\ S_{j_q} \mapsto r_q \qquad r = \wedge(\{\!|r_q|\!\}^{q \in \{\!|1 \dots l|\!\}}) \end{array}}{J\ ?\ \texttt{patternProperties} : \{\ p_1 : S_1, \dots, p_m : S_m\ \} \to r} \quad \text{(patternProperties)}$$

The (propertyNames) rule captures the semantics of `propertyNames` which, differently from other operators which constrain the instance values based on a specific schema, constrains the instance labels by imposing them to adhere to the associated schema S. The semantics of `propertyNames` is trivially conjunctive, as expressed in the premise of the rule.

$$\frac{J = \{\, k_1 : J_1, \ldots, k_n : J_n \,\} \qquad \forall i \in \{\!|1 \ldots n|\!\}.\ k_i \ ?\ S \mapsto r_i \qquad r = \wedge(\{\!|r(\sigma_i)|\!\}^{i \in \{\!|1 \ldots n|\!\}})}{J \ ?\ \texttt{propertyNames} : S \to r}\ (\mathsf{propertyNames})$$

The semantics of `dependentSchemas` is conditional and allows applying a sub-schema in case a label exists. The corresponding rule expresses this fact by collecting all sub-schemas that need to be applied on the validated instance J and combining the validation results with a conjunction.

$$\frac{\begin{array}{c} J = \{\, k'_1 : J_1, \ldots,\ k'_m : J_m \,\} \qquad \{\!|i_1, \ldots, i_l|\!\} = \{\!|i \mid i \in \{\!|1 \ldots n|\!\},\ k_i \in \{\!|k'_1, \ldots,\ k'_m|\!\}\}\\ \forall q \in \{\!|1 \ldots l|\!\}.\ J \ ?\ S_{i_q} \mapsto r_q \qquad r = \wedge(\{\!|r_q|\!\}^{q \in \{\!|1 \ldots l|\!\}}) \end{array}}{J \ ?\ \texttt{dependentSchemas} : \{\, k_1 : S_1, \ldots, k_n : S_n \,\} \to r}\ (\mathsf{dependentSchemas})$$

The `additionalProperties` operator is evaluated relatively to a context where any value J_{i_q} which has not been validated by an adjacent `properties` or `patternProperties` must adhere to the schema S. The evaluated values are those remaining after eliminating the pairs whose keys are in the languages of the properties and patterns extracted by the function $\texttt{propsOf}(\vec{K})$. The function `propsOf` is defined as follows, where the notation $\underline{k_i}$ indicates a pattern whose language accepts only the string value k_i:

$$\begin{array}{lcll} \texttt{propsOf}(\texttt{properties} : \{\, k_1 : S_1, \ldots, k_m : S_m \,\}) & = & \underline{k_1} \cdot | \cdot \ldots \cdot | \cdot \underline{k_n} & \\ \texttt{propsOf}(\texttt{patternProperties} : \{\, p_1 : S_1, \ldots, p_m : S_m \,\}) & = & p_1 \cdot | \cdot \ldots \cdot | \cdot p_m & \\ \texttt{propsOf}(K) & = & \emptyset & \text{otherwise} \\ \texttt{propsOf}([\, K_1, \ldots, K_n \,]) & = & \texttt{propsOf}(K_1) \cdot | \cdot \ldots \cdot | \cdot \texttt{propsOf}(K_n) & \end{array}$$

The semantics of this operator is also conjunctive, and the associated rule is a specific case of the (KwList-(n+1)) rule which collects the validation results of a keyword-list $\vec{K}$ in a conjunctive manner.

$$\frac{\begin{array}{c} J = \{\, k_1 : J_1, \ldots, k_n : J_n \,\} \qquad J \ ?\ \vec{K} \Rightarrow r\\ \{\!|i_1, \ldots, i_l|\!\} = \{\!|i \mid 1 \leq i \leq n \wedge\ k_i \notin L(\texttt{propsOf}(\vec{K}))|\!\}\\ \forall q \in \{\!|1 \ldots l|\!\}.\ J_{i_q} \ ?\ S \to r_q \qquad r' = \wedge(\{\!|r_q|\!\}^{q \in \{\!|1 \ldots l|\!\}}) \end{array}}{J \ ?\ (\vec{K} + \texttt{additionalProperties} : S) \Rightarrow r \wedge r'}\ (\mathsf{additionalProperties})$$

Array Keywords. These keywords only impact instances of type array, otherwise, their effect is trivial. Similar to the object keywords, evaluating an instance against such keywords necessitates the verification of the instance's internal structure.

The following rule evaluates array-type instances against a set of schemas specified within the `prefixItems` keyword. It asserts that each item of the instance located at position i, where $i \leq n$, is validated against the ith schema of `prefixItems`. The overall validity of the instance is obtained by combining the boolean values returned for each item. When either the instance or the list of schemas is empty, the returned value is $\mathcal{T}$.

$$\frac{J = [\, J_1, \ldots, J_m \,] \qquad \forall i \in \{\!|1 \ldots \mathsf{min}(n,m)|\!\}.\ J_i \ ?\ S_i \mapsto r_i \qquad r = \wedge(\{\!|r_i|\!\}^{i \in \{\!|1 \ldots \mathsf{min}(n,m)|\!\}})}{J \ ?\ \texttt{prefixItems} : [\, S_1, \ldots, S_n \,] \to r} \ (\mathsf{prefixItems})$$

The (contains) rule checks whether an item of the instance satisfies the schema S. Consequently, it returns $\mathcal{T}$ if at least one item fulfills the condition, and $\mathcal{F}$ otherwise.

$$\frac{J = [J_1, \ldots, J_n] \qquad \forall i \in \{\!|1 \ldots n|\!\}.\ J_i \ ?\ S \mapsto r_i \qquad r = \vee(\{\!|r_i|\!\}^{i \in \{\!|1 \ldots n|\!\}})}{J \ ?\ \texttt{contains} : S \to r} \ (\mathsf{contains})$$

The following rule checks the validity of the items that have not been evaluated by an adjecent `prefixItems` keyword present in $\vec{K}$. The list of items to verify are all those whose indexes come after the index captured by the function *maxPrefixOf*, which is defined as follows:

$$\begin{array}{lcll} \texttt{maxPrefixOf}(\texttt{prefixItems} : [\, S_1, \ldots, S_m \,]) & = & m & \\ \texttt{maxPrefixOf}(K) & = & 0 & \text{otherwise} \\ \texttt{maxPrefixOf}([\, K_1, \ldots, K_n \,]) & = & \max_{i \in \{\!|1 \ldots n|\!\}} \texttt{maxPrefixOf}(K_i) & \end{array}$$

The overall validity of the instance is obtained by combining the boolean values returned for each item.

$$\frac{\begin{array}{c} J = [\, J_1, ..., J_n \,] \qquad J \ ?\ \vec{K} \Rightarrow r \qquad \{\!|i_1, \ldots, i_l|\!\} = \{\!|1 \ldots n|\!\} \setminus \{\!|1 \ldots \mathit{maxPrefixOf}(\vec{K})|\!\} \\ \forall q \in \{\!|1 \ldots l|\!\}.\ J_{i_q} \ ?\ S \mapsto r_q \qquad r' = \wedge(\{\!|r_q|\!\}^{q \in \{\!|1 \ldots l|\!\}}) \end{array}}{J \ ?\ (\vec{K} + \texttt{items} : S) \Rightarrow r \wedge r'} \ (\mathsf{items})$$

The (contains-bounds) rule captures the semantics of `contains` in presence of the `minContains` and `maxContains` keywords. It generalizes the `contains` rules by introducing cardinality constraints: an array is valid when the cardinality of the items satisfying `contains` ranges between the values of `minContains` and `maxContains`.

$$\frac{\begin{array}{c} J = [\, J_1, ..., J_n \,] \\ \forall i \in \{\!|1 \ldots n|\!\}.\ J_i \ ?\ S \mapsto r_i \qquad \kappa_c = \{\!|i \mid r_i = \mathcal{T}|\!\} \qquad r_c = (i \leq |\kappa_c| \leq j) \end{array}}{J \ ?\ (\texttt{contains} : S + \texttt{minContains} : i + \texttt{maxContains} : j) \Rightarrow r_c} \ (\mathsf{contains\text{-}bounds})$$

Other Keywords. The $ref rule states the obvious validation of a `$ref`: *uri* statement which amounts to validating schema S' referenced by *uri*.

The two remaining rules capture the validation of conditionals which relies on validating either the *then* or *else* sub-schema depending on the validation result of the *if* statement.

$$\frac{S' = \texttt{getSchema}(uri) \qquad J \ ? \ S' \mapsto r}{J \ ? \ \texttt{\$ref} : uri \to r} \tag{\$ref}$$

$$\frac{J \ ? \ \vec{K} \Rightarrow r \qquad J \ ? \ S_i \mapsto \mathcal{T} \qquad J \ ? \ S_t \mapsto r'}{J \ ? \ (\vec{K} + \texttt{if} : S_i + \texttt{then} : S_t + \texttt{else} : S_e \ ) \Rightarrow r \wedge r'} \tag{if-true-then}$$

$$\frac{J \ ? \ \vec{K} \Rightarrow r \qquad J \ ? \ S_i \mapsto \mathcal{F} \qquad J \ ? \ S_e \mapsto r'}{J \ ? \ (\vec{K} + \texttt{if} : S_i + \texttt{then} : S_t + \texttt{else} : S_e \ ) \Rightarrow r \wedge r'} \tag{if-false-else}$$

3.4 Validation Example

In the following, we present a detailed example to illustrate the validation process of an instance against a schema in JSON Schema. After defining the core concepts of JSON and JSON Schema, as well as their syntax and semantics, it is crucial to demonstrate how validation works in practice. Through this example, we aim to showcase the interplay between different schema keywords and how the validation rules operate, particularly emphasizing how instances are evaluated against the assertions and constraints defined by the schema.

By walking through this example, we provide a concrete understanding of the validation process, allowing us to see how schemas are applied to specific data structures and how the operational semantics come into play. This example will further clarify how the judgments and rules interact to determine whether a given JSON instance conforms to the specified schema.

Example 8 *Consider the following schema, which defines a set of constraints for a JSON value to be valid: Firstly, through the* `type` *keyword, it specifies that valid instances must be of type* `object`*. Additionally, the instance must contain a minimum of three properties, as indicated by the* `minProperties` *constraint. The presence of the property* `"ab"` *is mandatory, which is enforced through the* `required` *keyword.*

Moreover, the schema utilizes `patternProperties` *to apply specific requirements based on property names. For properties whose names match the pattern* `"^a.*c$"`*, the value must be an integer that is a multiple of 2. Properties that match the pattern* `"^a.+$"` *must have values that are numbers greater than or equal to 20 and also multiples of 7. Lastly, for properties that match the pattern* `"^ab.*$"`*, the value must be a string consisting of at least two uppercase letters.*

```
{ "type": "object", "minProperties": 3, "required": ["ab"],
  "patternProperties": {
    "^a.*c$" :{ "type": "integer", "multipleOf": 2 },
    "^a.+$": {"minimum": 20, "multipleOf": 7 },
    "^ab.*$": {"type":"string", "pattern": "[A-Z]{2,}" }
  }
}
```

Now consider the following instances that we want to validate against the previous schema. These instances all share the same keys `"ab"` *and* `"ac"`*, both with identical values. The instance* J_1 *differs from the others by including an additional key,* `"acc"`*. In contrast, both* J_2 *and* J_3 *contain the key* `"abc"`*, but with different values associated with it.*

$$J_1 = \{ \texttt{"ab": "AA", "ac": 28, "acc": 28} \}$$
$$J_2 = \{ \texttt{"ab": "AA", "ac": 28, "abc": 28} \}$$
$$J_3 = \{ \texttt{"ab": "AA", "ac": 28, "abc": "AA"} \}$$

- *All three instances are valid against the keywords* `type`, `minProperties`*, and* `required` *as they are all JSON objects containing at least three properties, including the required property* `"ab"`*.*
- `patternProperties` *validation:*
  - `"ab"`*: this property has the same value,* `"AA"`*, of type* `string`*, in all three instances. It matches the patterns* `"^a.+$"` *and* `"^ab.*$"` *present in the schema of* `patternProperties`*.*

    Regarding the former pattern, `"^a.+$"`*, its corresponding schema does not specify the type of JSON values it admits. It only contains the* `minimum` *and* `multipleOf` *keywords, which are not specific to the type* `string`*, thus the validation is trivial and returns* $\mathcal{T}$*.*

    For the latter pattern, the `patternProperties` *specifies that if a property matches this pattern, then its value must be of type* `string` *and match the regular expression constraint defined by the* `pattern` *keyword, which is* `"[A-Z]{2,}"`*. The value of* `"ab"`*, which is* `"AA"`*, belongs to the language of this regular expression, making the pair (*`"ab"`*,* `"AA"`*) valid against this pattern and its schema.*
  - `"ac"`*: this property also has the same value,* 28*, across all instances. It is captured by both patterns* `"^a.*c$"` *and* `"^a.+$"`*.*

    The value 28 *is valid according to the schema for the former pattern since it is of type* `integer` *and a multiple of* 2*. Additionally, this value is also valid against the schema of the second pattern. As the schema contains constraints specific to the type* `integer`*, validation is non-trivial and must be performed. Since the value* 28 *is greater than* 20 *and is a multiple of* 7*, it is valid. Similarly to the previous property, the pair (*`"ac"`*,* 28*) is valid against the schema of* `patternProperties`*.*

- `"acc"`: *this property only appears in the instance J_1, with its value being* `28`*. Similar to the previous property, it matches both the first and second patterns, allowing us to conclude that its validation against the schema of* `patternProperties` *returns $\mathcal{T}$. Since the values of the three properties in J_1 are all valid against the various schemas of the patterns they match, this makes the instance J_1 valid against the entire schema S.*
- `"abc"`: *this property appears in both J_2 and J_3 with different values. in J_2 its value is* `28` *and in J_3 it is* `"AA"`*, and it matches the three patterns of* `patternProperties`*.*

  The value `28` *is valid against the schemas of the first two patterns but is not valid against the schema of the third pattern, as that schema only accepts values of type* `string`*. Therefore, the instance J_2 not valid against S.*

  Similarly, the value `"AA"` *is valid against the schemas of the second and third patterns, but it is not valid against the schema of the first pattern since it is not of type* `integer`*, making the instance J_3 invalid against S.*

  In fact, any instance containing a property that matches both the first pattern `"^a.*c$"` *and the third pattern* `"^ab.*$"` *will always be invalid against S. This is due to the implicit conjunction in the schema, as values of properties matching these patterns must satisfy both schemas. Since one schema only admits string values and the other admits only integer values, this contradiction makes it impossible for any value to satisfy the schemas of both patterns, resulting in an invalid instance.*

In summary, this example illustrates the validation process of JSON instances against a schema defined in JSON Schema. While the schema itself may appear straightforward, the overlapping patterns in `patternProperties` *highlight the potential complications that can arise from differing constraints. Even with relatively simple constraints, the interaction between schema keywords and the schemas associated with the same keyword can introduce complexities not only for validation but also for other tasks involving JSON Schema.*

In conclusion, this chapter has laid a solid foundation for understanding both the JSON data model and the JSON Schema vocabulary, emphasizing their syntax, semantics, and key features. By outlining the mechanisms of schema validation, we have provided essential context for how JSON Schema ensures data integrity and structure. This understanding sets the stage for the subsequent chapters, enabling a deeper comprehension of how data generation for JSON Schema is performed in the presence of various schema constraints.

Chapter 4

Witness generation

In this chapter, we describe an existing solution for instance generation for JSON Schema. This technique is notable for its soundness and completeness, and for being the only solution that can either generate a valid instance or determine whether a schema is unsatisfiable. We will discuss the main features of this method and the techniques it employs to achieve reliable instance generation.

4.1 Introduction

Witness generation is a crucial method for solving complex problems related to JSON Schema, such as *inclusion*, *equivalence*, and *satisfiability*. It involves determining whether, for a given JSON S, there exists a JSON value J that satisfies all the constraints imposed by S. If such value exists, it is called a *witness*, and the goal of the generation process is to return this witness. Although a schema can admit an infinite number of witnesses, the objective is to return at least one. Conversely, if no such value can be generated, the schema is said to be unsatisfiable, meaning that there is no JSON value that complies with the constraints defined in the schema.

This method is essential in practice because it provides concrete examples of JSON values that conform to a schema, which can help in verifying the schema's correctness and answering problems such as schema inclusion. For instance, when examining schema inclusion between two schemas S and S', if we want to determine whether the former is included in the latter, we can attempt to generate a witness for $S \wedge \neg S'$. If this conjunction is unsatisfiable, then the inclusion holds. Otherwise, if a JSON value is generated, the inclusion does not hold, and the generated value helps clarify why certain inclusion criteria are not met.

However, witness generation is a computationally challenging task due to the expressiveness of the JSON Schema language. The presence of the logical operators (`allOf`, `anyOf`, `not`, and `oneOf`) introduced earlier, along with the recursive mechanism using the keyword

`$ref` and various structural constraints introduces significant complexity. Previous research [30] has shown that the satisfiability problem for JSON Schema is EXPTIME-complete, illustrating the inherent difficulty of designing efficient algorithms to solve it.

To address this challenge, a complete and sound algorithm for witness generation has been developed. This approach can handle the entire set of JSON Schema operators, with the exception of `uniqueItems`, which is excluded because it significantly increases the complexity of the problem. The algorithm introduces several novel techniques, such as a preparation phase for objects and arrays, and a lazy and-completion strategy to optimize the handling of conjunctions. These techniques allow the algorithm to generate witnesses efficiently, even for complex schemas, while ensuring both soundness and completeness. In addition to the algorithm, extensive experiments have demonstrated the practicality of the solution. The results indicate that, despite the inherent complexity of the problem, the approach performs effectively on real-world schemas. This makes the algorithm a viable solution for generating JSON data.

In the following sections, we will describe the key phases of the algorithm by applying them to a specific example. We will illustrate the generation process for objects, demonstrating how each phase of the algorithm operates in practice. This detailed exploration aims to provide a comprehensive understanding of how to tackle the challenges of generating data for JSON Schema. We will also highlight the results previously discussed in [11, 12].

4.2 Understanding Witness Generation Through an Example

In this section, we will first introduce and describe the overall process of witness generation, outlining each key phase and its role in the algorithm. After presenting this structured overview, we will apply these steps to a specific schema example, demonstrating how the process works in practice. Following this example, we will provide insights of the experimental analysis to highlight the algorithm's performance and efficiency. Finally, we will conclude by discussing the strengths and limitations of the witness generation approach.

4.2.1 Witness Generation Overview

The witness generation process begins by translating an input JSON Schema into an algebraic representation. JSON Schema is inherently non-algebraic because the semantics of certain assertions can change depending on their surrounding context, making formal manipulation more complex. Additionally, certain operators in JSON Schema are redundant and can be expressed using logical combinations of existing ones. To address this, the schema is translated into a core algebra that minimizes redundancy by replacing those redundant operators with others. For instance, the keyword `const` is transformed into a

typed schema that only describes the value corresponding to this operator. For example, `{ "const": 1 }` is translated to `{ "type": "number", "minimum": 1, "maximum": 1 }`.

It is important to note that schemas are represented using variables, and those that contain sub-schemas are expressed as logical combinations of these variables. To manage this efficiently, Reduced Ordered Boolean Decision Diagrams (ROBDDs) are used. ROBDDs ensure that the same variable is not added twice when representing a schema. By computing the ROBDD representation of a schema and storing it, the process avoids duplicating equivalent variables. This method optimizes schema representation, improving the algorithm's efficiency while ensuring that schema handling remains consistent throughout the process.

After the translation comes the negation elimination phase. The algebra used during this phase is the resulting algebra from the translation process, extended with new operators to remove the negation operator. These new operators correspond to JSON Schema operators that do not possess a dual, meaning they cannot be expressed using their dual operators combined with negation, such as `multipleOf`. The negation elimination process involves defining a complement variable for each variable defined during translation and then pushing the negation inward. Whenever a negation is encountered, it is replaced by the corresponding complement variable.

Next, the schema undergoes stratification, which involves substituting schemas corresponding to typed assertions with newly defined variables when those schemas are not already variables. For every newly defined variable, a complement variable is also defined. The goal of this process is to ensure that all typed assertions have a single variable as their argument.

Afterwards, the schema is transformed into Canonical Guarded Disjunctive Normal Form (GDNF). In GDNF, a schema is expressed as a disjunction of conjunctions of terms, where each term is a typed assertion. This transformation standardizes the schema and makes it efficient for analysis. Following this, the canonicalization process further refines the GDNF schema. Canonicalization involves splitting conjunctive terms into sets of typed groups, with each typed group consisting of a single typed assertion and denoting instances of the same type.

In the following, we will discuss the step preceding generation: the preparation of object and array groups, which, as mentioned earlier, is a novel technique introduced in this witness generation approach.

Object Preparation. The object preparation process distinguishes object assertions into two separate categories: constraints and requirements. Requirements are assertions that must be met, establishing the minimum conditions that the object must satisfy. These requirements ensure that specific patterns or properties are included in the object; they consist of the assertions `minProperties`, `required`, and `propertyNames`. On the other hand, constraints are those assertions that set the maximum limits on object properties, ensuring that certain conditions are not exceeded. These constraints include the assertions `maxProperties`, `properties`, and `patternProperties`.

To clarify this distinction, consider an empty object: an empty object will always satisfy the constraints, but it may not satisfy the requirements if any are present. Conversely, once an object satisfies all the requirements, it will continue to do so regardless of any additional key-value pairs added. However, adding new key-value pairs could potentially lead to violations of the constraints.

The preparation process begins by incrementally constructing objects. Initially, an empty object is created, and members are added step by step to meet all the requirements. These members consist of keys and the corresponding schemas that they must satisfy. Each addition is carefully verified to ensure that it also respects the constraints. In simple cases, this process is straightforward. However, in more complex scenarios, where patterns overlap or the object schema is non-trivial, additional steps are necessary. In these situations, the preparation process involves explicitly managing the interactions between constraints and requirements. Instead of trying to build a solution and generating objects in a trial-and-error manner, the process analyzes all possible combinations upfront, making the interactions between constraints and requirements explicit.

Once these interactions are identified, the next step is selecting the viable combinations that fulfill the necessary conditions and can lead to successful object generation. Some combinations lead to valid objects, while others do not; for instance, two overlapping patterns may contain contradictory conditions in their schemas. To handle this process effectively, the system may introduce new variables to manage the conjunction of multiple constraints and requirements. Preparation of objects might sometimes rely on previous phases, such as GDNF, and utilize ROBDDs to avoid creating variables that already exist.

Array Preparation. Similar to objects, array preparation involves managing two main categories of assertions: constraints and requirements. Constraints are conditions that arrays must meet but are inherently satisfied by an empty array; these include limits on the number of items or specific schemas the items must comply to. Requirements, on the other hand, set minimum criteria that an array must fulfill, such as having a minimum number of elements or ensuring that certain values are present at specific positions. In array preparation, requirements are defined by assertions such as `minItems`, `minContains` and `contains`, while constraints encompass all other array assertions.

The interplay between requirements and constraints introduces complexities that are addressed during this preparation phase. These interactions can lead to situations where array elements must satisfy both constraints and requirements, resulting in conjunctions of conditions. Arrays face additional challenges due to constraints like the `maxContains` assertion, which involves managing negations and imposes upper bounds on the number of items satisfying certain criteria.

Array preparation differs from object preparation in that it needs to deal with the positions of items within the array since arrays are ordered lists of items. This positional aspect adds complexity to the preparation which not only has to consider all possible combinations of constraints and requirements, but also on which indexes these combinations need to be satisfied. The counting requirements also introduce extra complexity since it

requires keeping track of the number of generated items that satisfy a specific schema.

Remark 2 *Given that the approach seeks to exhaustively enumerate all possible solutions by examining every combination of constraints and requirements to generate witnesses, witness generation will be referred to as the* ***pessimistic*** *approach in the subsequent chapters.*

4.2.2 Witness Generation Process

Before running the algorithm on the example, we will first introduce the key JSON Schema operators used in the example and their corresponding algebraic representations. It is important to note that once the schema is translated into the algebra, the subsequent phases are performed on this algebraic form of the schema. The Table 4.1 outlines these operators and their equivalent in the algebra.

Assertion	Algebraic Representation
`type` : Tp	$\mathsf{type}(\mathsf{Tp})$
`minLength`: i	len_i^{∞}
`pattern`: p	$\mathsf{pattern}(p)$
`minProperties`: i	pro_i^{∞}
`required`: $[k_1, \ldots, k_n]$	$\mathsf{req}(k_1, \ldots, k_n)$
`properties`: $\{k_1 : S_1, \ldots, k_n : S_n\}$	$\mathsf{props}(k_1 : S_i, \ldots, k_n : S_n)$
`patternProperties`: $\{p_1 : S_1, \ldots, p_n : S_n\}$	$\mathsf{props}(p_1 : S_i, \ldots, p_n : S_n)$

Table 4.1: Translation of JSON Schema operators to the algebra

In the following, we will illustrate the object generation process through an example schema that demonstrates the interactions between requirements and constraints. While we have streamlined the process for clarity, the key concepts remain relevant and provide a foundational understanding of the overall idea introduced in [11, 12].

Example 9 *The following schema describes objects with both requirements and constraints. The objects must have at least two properties, as indicated by the* `minProperties` *constraint. One required property,* `"a"`*, must be a string that matches the pattern* `"(a|b)c.*"`*, meaning it must start with either* `"a"` *or* `"b"` *followed by the character* `"c"`*, and then followed by zero or more occurrences of any character. Additionally, the* `patternProperties` *field specifies that any property whose name matches the pattern* `"a+"` *(i.e., one or more occurrences of the character* `"a"`*) must have a value with a minimum length of 3 characters. Together, this schema introduces interactions between the* `required` *and* `patternProperties` *keywords, as well as between* `pattern` *and* `minLength`*, requiring instances that satisfy the schema to adhere to these interactions during generation.*

```
{ "type": "object", "required": ["a"], "minProperties": 2,
  "properties": { "a": { "type": "string", "pattern": "(a|b)c.*" } },
  "patternProperties": { "a+": {"minLength": 3 } }
}
```

The generation of a witness for this schema proceeds as follows:

1. Translation to the core algebra: as outlined in the overview of the approach, the first step consists of translating the schema into an algebraic representation. This step is straightforward, as it involves mapping each JSON Schema keyword to its corresponding algebraic form described in Table 4.1. Here, $\mathbf{r}$ represents the root schema, while $\mathbf{x}$ and $\mathbf{y}$ correspond to the schemas appearing within the `properties` and `patternProperties` keywords, respectively.

$$\mathbf{r} = \mathtt{type}(\mathsf{Obj}) \wedge \mathsf{req}(a) \wedge \mathsf{props}(a : x) \wedge \mathsf{props}(a+ : y) \wedge \mathsf{pro}_2^\infty$$
$$\mathbf{x} = \mathtt{type}(\mathsf{Str}) \wedge \mathsf{pattern}((a|b)c.*), \quad \mathbf{y} = \mathsf{len}_3^\infty$$

2. The positive algebra: as stated in the previous section, the algebraic representation is extended with new operators to eliminate the operator `not`. In this step, compared to the earlier schema representation, the operator pattReq is introduced to handle the negation of `required` when negation is involved. It combines the semantics of both `required` and `properties`, ensuring that a valid JSON instance must contain the key a, and the associated value must satisfy the schema represented by the variable $\mathbf{x}$.

$$\mathbf{r} = \mathtt{type}(\mathsf{Obj}) \wedge \mathsf{pattReq}(a : x) \wedge \mathsf{props}(a+ : y) \wedge \mathsf{pro}_2^\infty$$
$$\mathbf{x} = \mathtt{type}(\mathsf{Str}) \wedge \mathsf{pattern}((a|b)c.*), \quad \mathbf{y} = \mathsf{len}_3^\infty$$

3. Negation elimination: during this phase, the complement of each variable previously defined is determined. Here, $\mathbf{co}(\mathbf{r})$ is the complement of the root variable $\mathbf{r}$, and its corresponding schema describes the JSON values that are not valid against the schema of $\mathbf{r}$. These include all values that are not of type `object`, hence the disjunction of all other types, and object values that do not satisfy the constraints defined in the schema of $\mathbf{r}$. Here, the schema of $\mathbf{r}$ includes three distinct object-specific constraints, each inducing a different schema that captures the values that do not conform to the original schema. For instance, an invalid object value against the schema of $\mathbf{r}$ is one that has less than 2 properties, represented by the schema $\mathtt{type}(\mathsf{Obj}) \wedge \mathsf{pro}_0^1$.

   The schema of the complement variable of $\mathbf{x}$ describes all values that are not of type `string` or values of type `string` that are in the language of the pattern $\overline{(a|b)c.*}$, which denotes the pattern that matches any string not matched by $(a|b)c.*$.

   The schema of $\mathbf{y}$ allows for all value types, but when a value is a string, it must meet the length constraint. Consequently, its complement does not include instances of other types, and the string values must be limited to a maximum of 2 characters.

   The negation is pushed inside each operator, where pattReq becomes props with negation pushed inside, resulting in $\mathbf{co}(\mathbf{x})$ being the schema of the key a, and vice versa, where `properties` becomes pattReq with $\mathbf{co}(\mathbf{y})$ being the schema describing the values of the pattern $a+$.

$\mathbf{r} = \texttt{type}(\mathsf{Obj}) \wedge \mathsf{pattReq}(a : x) \wedge \mathsf{props}(a+ : y) \wedge \mathsf{pro}_2^\infty$
$\mathbf{x} = \texttt{type}(\mathsf{Str}) \wedge \mathsf{pattern}((a|b)c.*), \quad \mathbf{y} = \mathsf{len}_3^\infty$

$\mathbf{co}(\mathbf{r}) = \texttt{type}(\mathsf{Null}) \vee \texttt{type}(\mathsf{Bool}) \vee \texttt{type}(\mathsf{Num}) \vee \texttt{type}(\mathsf{Str}) \vee \texttt{type}(\mathsf{Arr}) \vee$
$(\texttt{type}(\mathsf{Obj}) \wedge \mathsf{props}(a : co(x))) \vee (\texttt{type}(\mathsf{Obj}) \wedge \mathsf{pattReq}(a+ : co(y))) \vee (\texttt{type}(\mathsf{Obj}) \wedge \mathsf{pro}_0^1)$

$\mathbf{co}(\mathbf{x}) = \texttt{type}(\mathsf{Null}) \vee \texttt{type}(\mathsf{Bool}) \vee \texttt{type}(\mathsf{Num}) \vee \texttt{type}(\mathsf{Arr}) \vee \texttt{type}(\mathsf{Obj}) \vee$
$(\texttt{type}(\mathsf{Str}) \wedge \mathsf{pattern}(\overline{(a|b)c.*}))$

$\mathbf{co}(\mathbf{y}) = \texttt{type}(\mathsf{Str}) \wedge \mathsf{len}_0^2$

4. In the previous representation, no combination of variables is contained within a typed assertion; hence, all the schemas already meet the stratification requirements. Next comes the transformation into GDNF. All schemas are already in GDNF; for instance, $\mathbf{r}$ is already a typed object group, except for the variable $\mathbf{y}$, which is transformed to capture a disjunction of all the other types along with the string typed assertion $\texttt{type}(\mathsf{Str}) \wedge \mathsf{len}_3^\infty$.

$\mathbf{r} = \texttt{type}(\mathsf{Obj}) \wedge \mathsf{pattReq}(a : x) \wedge \mathsf{props}(a+ : y) \wedge \mathsf{pro}_2^\infty$
$\mathbf{x} = \texttt{type}(\mathsf{Str}) \wedge \mathsf{pattern}((a|b)c.*)$

$\mathbf{y} = \texttt{type}(\mathsf{Null}) \vee \texttt{type}(\mathsf{Bool}) \vee \texttt{type}(\mathsf{Num}) \vee \texttt{type}(\mathsf{Arr}) \vee \texttt{type}(\mathsf{Obj}) \vee (\texttt{type}(\mathsf{Str}) \wedge \mathsf{len}_3^\infty)$

$\mathbf{co}(\mathbf{r}) = \texttt{type}(\mathsf{Null}) \vee \texttt{type}(\mathsf{Bool}) \vee \texttt{type}(\mathsf{Num}) \vee \texttt{type}(\mathsf{Str}) \vee \texttt{type}(\mathsf{Arr}) \vee$
$(\texttt{type}(\mathsf{Obj}) \wedge \mathsf{props}(a : co(x))) \vee (\texttt{type}(\mathsf{Obj}) \wedge \mathsf{pattReq}(a+ : co(y))) \vee (\texttt{type}(\mathsf{Obj}) \wedge \mathsf{pro}_0^1)$

$\mathbf{co}(\mathbf{x}) = \texttt{type}(\mathsf{Null}) \vee \texttt{type}(\mathsf{Bool}) \vee \texttt{type}(\mathsf{Num}) \vee \texttt{type}(\mathsf{Arr}) \vee \texttt{type}(\mathsf{Obj}) \vee$
$(\texttt{type}(\mathsf{Str}) \wedge \mathsf{pattern}(\overline{(a|b)c.*}))$

$\mathbf{co}(\mathbf{y}) = \texttt{type}(\mathsf{Str}) \wedge \mathsf{len}_0^2$

5. Object preparation: during this phase, the algorithm explores the different combinations of requirements and constraints to determine the combinations that will produce an instance that satisfies the schema.

   The requirements in the schema of the example are represented by the assertions $\mathsf{pattReq}(a : x)$ and pro_2^∞, while the constraints are represented by $\mathsf{props}(a+ : y)$.

   To satisfy the first requirement (i.e., the presence of a), two possible choices exist:

   - Generating a key in the language defined by the intersection $a \cap a+$.
   - Generating a key in the language defined by the intersection $a \cap \overline{a+}$.

Given that a is in the language of the pattern $a+$, the second option is not viable. Therefore, the first option is chosen, which means the value of the key a must satisfy the schema defined by $\mathbf{x} \wedge \mathbf{y}$. A new variable will be created to express this conjunction, and afterward, it will be transformed into GDNF and canonicalized. This process consists of distributing the schema of $\mathbf{x}$ over the disjunctive schema of $\mathbf{y}$, resulting in the schema $\mathsf{type}(\mathsf{Str}) \wedge \mathsf{pattern}((a|b)c.*) \wedge \mathsf{len}_3^\infty$. Here, the generation process will create a string value with a minimum length of 3 that is in the language of the pattern $(a|b)c.*$.

Secondly, to satisfy the second requirement pro_2^∞, we need to choose a second pair of patterns and schemas. This can be achieved by selecting a key that is not in the languages of both patterns a and a, or that is only in the language of $a+$ but not in a. The inclusion of a and $a+$ is not a viable solution since it has already produced the key a, and it is the only value possible given that the intersection of the patterns only accepts the string a.

For instance, if the chosen solution does not consider both patterns, then the key to generate is in the language of the pattern $\overline{a} \cup \overline{a+}$, and its value must adhere to the schema of the variable whose body is $\mathbf{co}(\mathbf{x}) \wedge \mathbf{co}(\mathbf{y})$. After it undergoes the different transformations outlined previously, this schema results in $\mathsf{type}(\mathsf{Str}) \wedge \mathsf{len}_0^2 \wedge \mathsf{pattern}(\overline{(a|b)c.*})$. Finally, generation proceeds to create a valid key-value pair that will be added to the previously generated key to form a valid object.

4.3 Experimental Analysis

In this section, we present the experimental evaluation of the witness generation algorithm for JSON Schema, using the same results obtained in prior research. These experiments were designed to evaluate the algorithm's performance in terms of correctness, completeness, and efficiency when applied to various schema collections. Witness generation is computationally challenging due to the expressiveness of the JSON Schema language, and the experiments aimed to determine whether the algorithm could generate valid instances within a reasonable timeframe, even for complex schemas. Additionally, the goal was to compare its performance with existing JSON Schema data generators to determine whether it advances the state of the art.

The experimental analysis conducted in this earlier research is an important contribution of this thesis, providing valuable insights into the behavior and structure of JSON Schema. Through the results, we gained a clearer understanding of the algorithm's strengths and limitations when dealing with the challenges of data generation.

In the following, we will describe the experimental setup, outline the schema datasets used, and summarize the key findings.

4.3.1 Implementation and Experimental Setup

The witness generation algorithm was implemented in Java 11, utilizing the Brics [60] library for generating string values from patterns and the jdd library [68] for handling ROBDDs. The experiments were conducted on a Precision 7550 laptop with a 12-core Intel i7 2.70GHz CPU, 32GB of RAM, running Ubuntu 21.10. Each schema was processed by a single thread, with the JVM heap size set to 10GB. Witnesses were validated using an external validator [3] and manually verified in cases where the validator reported false negatives. The schemas for each collection were processed sequentially in a single execution, with a 60-minute timeout enforced per schema.

Remark 3 *Since the Brics library lacks support for many features of the ECMA-262 [39] regular expressions used in JSON Schema, all patterns are transformed during preprocessing to ensure compatibility with Brics syntax, following the translation mechanism described in [55].*

4.3.2 Schema Collections

To assess the effectiveness of the withess generation algorithm, it was tested on several different schema collections, which are detailed in Table 4.2, including both satisfiable and unsatisfiable schemas.

Collection	**#Total**	**#Sat/#Unsat**	**Size (KB): Avg/Max**
Github (Git) [25]	6,427	6,387/40	8.7/1,145
Kubernetes (K8s) [57]	1,092	1,087/5	24.0/1,310.7
Snowplow (Snw) [4]	420	420/0	3.8/54.8
WashingtonPost (WP) [65]	125	125/0	21.1/141.7
Handwritten (HW) [11, 12]	235	197/38	0.1/109.4
Containment-draft4 (*CC4*) [8]	1,331	450/881	0.5/2.9

Table 4.2: Description of the schema collections

Real-world Schemas. The largest collection was gathered from GitHub, with JSON Schema-related files retrieved using a BigQuery search on the GitHub public dataset. After identifying approximately 80K potential schemas, a thorough process of duplicate removal and data cleaning was applied, reducing the set to 6,427 unique schemas. Among these, 40 were identified as unsatisfiable by the tool and then confirmed through manual inspection. Additionally, occurrences of the `uniqueItems` keyword were renamed, making it a non-validating keyword since the algorithm does not process this keyword.

The other three real-world schema collections came from established standards. These included schemas related to Kubernetes (for application deployment), Snowplow (for regulating system interactions), and The Washington Post (for managing data from content

management systems). Previous studies had already used earlier versions of these collections for inclusion checks. Nearly all schemas in these collections are satisfiable, except for five in the Kubernetes set.

Handwritten Schemas. Real-world schemas, while reflecting practical applications, often focus on commonly used operators and lack more complex interactions needed for stress-testing. To address this, a collection of 235 handcrafted schemas was created to specifically challenge the algorithm by demonstrating intricate interactions between JSON Schema operators. These schemas, while smaller in size, illustrate complex behaviors such as implicit conjunctions in objects caused by overlapping patterns. They also highlight advanced scenarios involving nested logical operators, such as `oneOf` inside `not`.

Synthesized Schemas. In addition to real-world and handwritten collections, synthesized schemas derived from the JSON Schema validation test suite were incorporated. These schemas are designed to systematically cover the full range of JSON Schema operators. They consist of pairs of schemas S and S', along with a boolean value indicating whether S is included in S'. Tests were restricted to schemas compliant with Draft-04, as this version is used by competing tools and thus serves as a common standard for comparison. Schemas containing unsupported features, such as the `format` keyword or external file references, were excluded. Containment or inclusion was checked by generating witnesses for the combined schema $S \wedge \neg S'$, where unsatisfiability indicates that S is contained within S'. Both satisfiable and unsatisfiable schemas were included, providing a broad range of test cases.

4.3.3 Experimental Results

Table 4.3 summarizes the results of the experiments, in which the witness generation tool is compared with the *DG* tool [28] for real-world and handwritten schemas, as *DG* is primarily used for data generation. For synthesized schemas, the comparison includes both the *DG* tool and the *CC* tool [51, 46], the latter of which was developed as a containment checker to verify schema inclusion between two given schemas. As previously stated, the synthesized schemas consist of two schemas for which inclusion is being verified, along with a ground truth specifying whether the inclusion holds or not, making it possible to compare results with the *CC* tool.

When evaluating each tool, three distinct outcomes are identified: *success*, when a correct result is produced; *interruption*, which include timeouts, memory issues, and other runtime errors; and *logical errors*, which are further categorized into errors on satisfiable schemas, where the schema S is satisfiable but the code either returns "unsatisfiable" or provides a witness that does not satisfy S, and errors on unsatisfiable schemas, where the schema is unsatisfiable but a witness is still generated.

Col.	Tool	Success	Interrupt.	Errors sat.	Errors unsat.	Med. Time	95% -tile	Avg. Time
Git	Ours	99.19%	0.81%	0%	0%	0.019 s	0.749 s	4.289 s
	DG	94.2%	2.86%	2.43%	0.51%	0.021 s	0.082 s	0.190 s
K8s	Ours	100%	0%	0%	0%	0.013 s	0.510 s	0.577 s
	DG	99.54%	0%	0%	0.46%	0.023 s	0.069 s	0.031 s
Snw	Ours	99.52%	0.48%	0%	no unsat	0.065 s	3.864 s	2.071 s
	DG	94.76%	0%	5.24%	no unsat	0.024 s	0.078 s	0.032 s
WP	Ours	100%	0%	0%	no unsat	0.042 s	132.690 s	23.349 s
	DG	96.8%	0%	3.2%	no unsat	0.030 s	0.079 s	0.042 s
HW	Ours	100%	0%	0%	0%	0.070 s	3.063 s	2.593 s
	DG	8.51%	34.04%	49.36%	8.09%	0.023 s	0.132 s	0.049 s
CC4	Ours	100%	0%	0%	0%	0.004 s	0.038 s	0.011 s
	DG	28.78%	30.88%	0.07%	40.27%	0.020 s	0.034 s	0.019 s
	CC	35.91%	62.96%	0.15%	0.98%	0.003 s	0.096 s	0.036 s

Table 4.3: Correctness and completeness results, median/95th percentile/average runtime (in seconds)

Correctness and Completeness. Table 4.3 highlights that in the Github collection, the witness generation tool encounters interruption errors in 0.81% of the schemas (52 schemas). Among these interruptions, 0.44% are due to timeouts (28 schemas), 0.01% are caused by ref-expansion issues (1 schema), and 0.36% result from out-of-memory errors when using the automata library (23 schemas). Additionally, 2 timeouts are noted for the Snowplow collection, attributed to the presence of the `maxLength` keyword with a high value. Since Brics constructs an automaton based on this, it results in increased processing time. The tool successfully processes the remaining schemas, with no logical errors observed in any of the schema collections.

The *DG* tool successfully processes 94.20% of the GitHub schemas and achieves a similar level of correctness for other real-world schemas. However, it performs poorly on handwritten schemas, where it shows 49.36% logical errors on the satisfiable schemas of this collection. This issue arises from the tool's inability to handle intricate situations involving negation and complex interactions between object constraints and array constraints, which sometimes involve implicit relationships that must be carefully managed. Additionally, the tool uses a different library for generating strings, which contributes to some logical errors in certain cases. Furthermore, the *DG* tool is not suitable for inclusion checking as it fails to detect unsatisfiability.

Regarding the *CC* tool, the results on the Containment-draft4 collection exhibit a very high rate of interruption errors, which can be attributed to its limited support for only a subset of the JSON Schema language. This low coverage results in frequent runtime issues when encountering unsupported features. Despite this, the tool demonstrates relatively few logical errors, indicating that when it does process a schema successfully, it generally produces correct results. However, the frequent interruptions limit its overall reliability and applicability across a broader range of schemas.

In conclusion, the witness generation tool demonstrates better results, as it was specif-

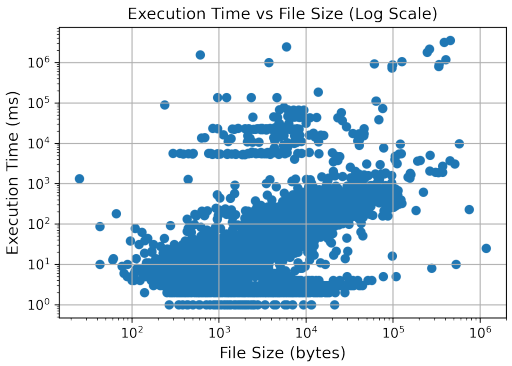
ically designed to target both completeness and correctness. In contrast, the *DG* and *CC* tools were intended for different purposes, with the *CC* tool in particular lacking the capability to fully support the entire JSON Schema language, resulting in higher interruption errors and limited reliability.

Execution Times. The results about execution times in Table 4.3 show that both the *DG* tool and *CC* tool demonstrate low and consistent execution times across the collections, with minimal variance between median, 95th percentile, and average values. On the other hand, while the witness Generation tool exhibits very low median times across all collections, it experiences some outliers that lead to significantly higher 95th percentile and average times. In the Snowplow collection, these outliers with high execution times are due to the extensive presence of the `pattern` keyword, which results in multiple calls to the Brics library during schema preprocessing. Regarding the Washington Post collection, the analysis of the outliers shows intricate combinations of operators, particularly nested logical operators. The GitHub collection contains both categories: the presence of complex patterns and intricate interactions between operators. These observations are further validated by Figure 4.1, which represents the correlation between schema sizes and execution times in a scatter plot on a log-log scale, one of the other aspects analyzed during these experiments. We observe a linear correlation for many schemas, but there are some outliers with small sizes but high execution times. This confirms the previous observation that high execution times are due to the presence of specific operators and their interactions, as well as complex patterns, particularly those involving `maxLength` with high values or intricate patterns.

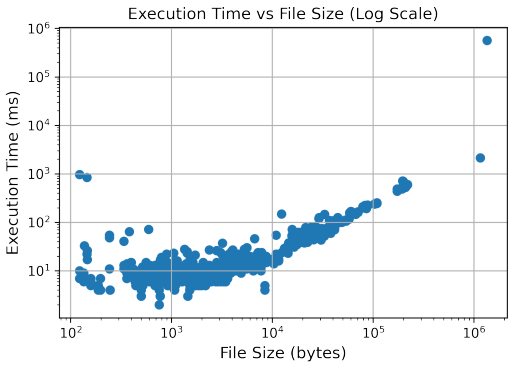
Despite the higher execution times in some schemas, the Witness Generation tool ultimately outperforms the others due to its higher success rate, as it consistently delivers correct results without logical errors. This makes it a more reliable tool in complex cases, where correctness is prioritized over low execution times.

4.4 Conclusion

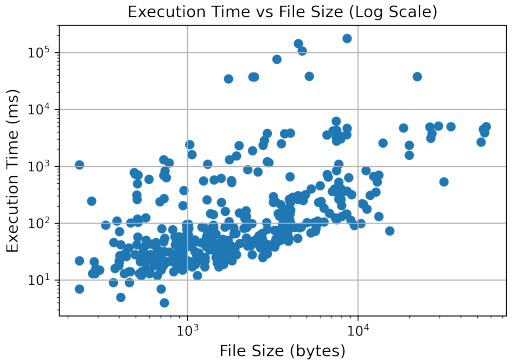
In conclusion, this prior work has introduced a practical and effective algorithm for witness generation for the JSON Schema language, addressing important challenges such as schema satisfiability and containment checking. By combining well-established techniques with novel strategies, such as object and array preparation, the algorithm effectively handles the intricacies arising from the language's operators and their various interactions. The focus on designing a robust algorithm targeting correctness and completeness has led to the development of a tool that efficiently processes both real-world schemas and complex handcrafted ones, where other tools often encounter logical or runtime errors. The experimental analysis attests to the viability of the approach, demonstrating that it can consistently generate correct results, even for complex schemas. While certain schemas result in higher execution times due to factors such as the exponential complexity of the preparation phase, the presence of complex patterns, and the operator `maxLength` with



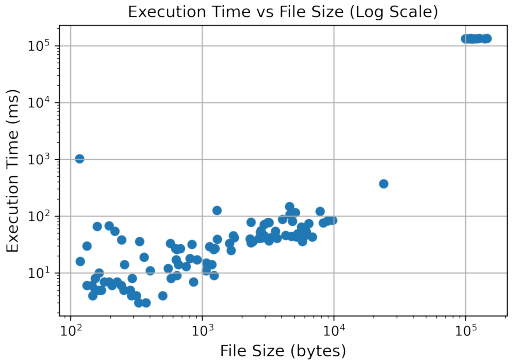
(a) Github collection



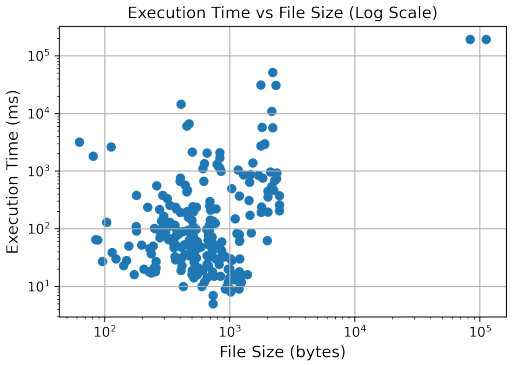
(b) Kubernetes collection



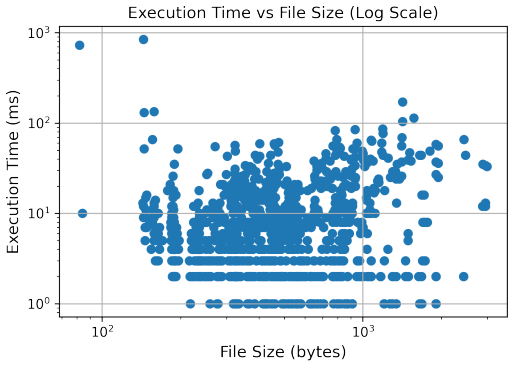
(c) Snowplow collection



(d) WashingtonPost collection



(e) Handwritten collection



(f) Containment-draft4 collection

Figure 4.1: File size vs. runtime on a Log-Log Scale

high values, intricate interactions involving nested logical operators also contribute to this increase. Despite these challenges, the overall performance, correctness, and reliability of the witness generation tool significantly outperform competing solutions, particularly in more complex scenarios.

This approach while being complete still needs to consider the `uniqueItems` operator

which introduces extra complexity to the problem. Addressing this in future work is desirable as it will lead to covering all the JSON Schema language. Additionally, while the current algorithm focuses on generating a single instance, a natural next step would be to develop methods for generating multiple instances with different levels of coverage, and also generating not only synthetic data but real data as well.

Overall, the work provides a solid foundation for data generation for JSON Schema, advancing the state of the art in this domain by offering a reliable and efficient solution to a previously unsolved problem. The results not only affirm the feasibility of handling the complexity of the language within a reasonable timeframe but also open the door to further refinements and broader applications in schema analysis and data validation.

Chapter 5

A Novel Optimistic Approach For Generation

In this chapter, we introduce a novel optimistic approach for JSON Schema data generation that prioritizes efficiency by minimizing schema preprocessing and complex rewritings. Our method focuses on soundness over completeness, providing valid instances for satisfiable schemas while identifying unsatisfiable ones. We will detail the preprocessing phase, which includes reference expansion and schema canonicalization, as well as the generation process for each data type, outlining the specific algorithms that form the core of this technique.

5.1 Introduction

The need for an efficient and correct data generation tool for JSON Schema has become increasingly critical, particularly as JSON Schema continues to evolve with new features, and the complexity of schemas grows with these updates. Existing approaches, while effective in some scenarios, often present significant trade-offs. This chapter introduces a novel optimistic approach for JSON Schema data generation that seeks to strike a balance between completeness, correctness, and performance, addressing the limitations of earlier methods. By focusing on a more streamlined, sound approach that emphasizes speed without sacrificing too much in terms of completeness, this technique offers a practical solution for various generation tasks.

In Chatper 4, we explored a pessimistic approach for instance generation for JSON Schema, and through the experimental analysis in that chapter, we investigated optimistic strategies for data generation. The pessimistic approach, while thorough and accurate, can be computationally expensive and slow due to its exhaustive nature. On the other hand, existing optimistic approaches provide faster results but at the expense of completeness and correctness. These optimistic tools often fail when schemas become too complex or involve challenging constructs like conjunctions and negations, and they also lack mechanisms

to identify unsatisfiable schemas. A common feature of of both the pessimistic and the optimistic techniques, is that they are designed to generate a single instance, meaning that running the generators on a schema S will always return a single instance when successfully processed. Although they can be used to produce multiple instances by constructing an array schema whose `items` sub-schema is S and selecting a minimum number of items to generate, it is uncertain whether the generated instances will be distinct, which is a drawback in scenarios that require unique examples.

This chapter addresses these challenges by proposing a new optimistic approach designed to offer faster data generation while maintaining high coverage with only a slight trade-off in completeness. Our method avoids the complex schema rewritings and exhaustive preparations introduced in previous work, such as the techniques in [11, 12]. Instead, we rely on a minimalistic preprocessing phase that simplifies the schema, followed by an efficient generation phase. We ensure soundness by processing conjunctions and partially handling negations. Additionally, like the pessimistic approach, our method can statically check for schema emptiness, allowing it to identify in most cases when a schema is unsatisfiable.

The primary goals of this approach are to provide a more efficient generator that is both fast and correct (sound), while accepting some loss of completeness. Unlike previous solutions, our method is not limited to generating a single instance; it is also designed to generate multiple distinct instances. Additionally, it is capable of handling the `uniqueItems` constraint in arrays, a feature not supported or addressed naively in most earlier generation methods. This capability is achieved through the use of multiple distinct instances generation.

To achieve these goals, the approach is divided into two main phases. First, in the preprocessing phase, we introduce reference expansion and schema canonicalization. The canonicalization process, adapted from [46] and extended for Draft 2019 of the standard [69], helps prepare the schema for efficient data generation. Reference expansion is straightforward, ensuring that references within the schema are resolved up to a certain depth. Schema canonicalization, on the other hand, transforms the schema into a canonical form, making it easier to generate valid instances in the subsequent phase.

The second phase is the generation phase, which begins with unsatisfiability checking to ensure that the schema admits instances. Once this is confirmed, the schema can then be used to generate the desired number of instances. A dedicated generator is assigned to each data type, including basic types, object types, and array types, ensuring that the correct instances are produced based on the structure and constraints of the schema. Similar to the pessimistic approach introduced in Chapter 4, all generation related to strings is handled using the Brics library [60], which provides robust support for generating string instances in accordance with the specified schema constraints.

In the following sections, we first describe the preprocessing phase in detail, including reference expansion and schema canonicalization. We then present the generation phase, where we outline how each data type is processed, accompanied by the necessary algorithms. The chapter concludes by summarizing the key contributions of our new approach.

5.2 The Preprocessing Phase

In this section, we will discuss the preprocessing phase of our new optimistic approach for JSON Schema data generation, which consists of two main steps: reference expansion and schema canonicalization. This phase is essential for simplifying the schema and preparing it for the generation process by resolving references and applying transformations to standardize its structure.

The first step, reference expansion, involves resolving all `$ref` keywords within the schema. References are replaced with their corresponding schema definitions, ensuring that the entire schema becomes self-contained and easier to process in subsequent phases. This step is straightforward but critical, as unresolved references can complicate both schema understanding and manipulation, potentially leading to errors or incomplete data generation.

The second step, schema canonicalization, plays a pivotal role in preparing the schema for efficient generation. It is based on a set of transformation rules outlined by Habib et al. [46] and extends them to support the Draft 2019 standard [69]. Additionally, we incorporate rules from [11, 12], which help in handling other JSON Schema operators not supported by earlier work. Importantly, this canonicalization process is designed to be minimal and less complex than the exhaustive schema rewritings introduced in Chapter 4. The goal of canonicalization is to restructure the schema to eliminate conjunctions when possible and partially handle negations, applying rewriting rules that reduce schema complexity while preserving its original semantics. Ideally, the schema is transformed into a disjunctive normal form, containing only the `anyOf` keyword, which corresponds to a disjunction of schemas, each representing values of the same JSON type.

After canonicalization, the schema remains semantically equivalent to the original, describing the same set of valid JSON instances, but is in a form that is more amenable to efficient data generation. This transformation is key to ensuring the generation phase operates smoothly without encountering the complexities of the original schema.

In the following subsections, we will provide a detailed explanation of each of these steps, along with examples to illustrate how they are performed on a schema. This will clarify the techniques used and highlight the practical impact of each step in the overall preprocessing phase.

5.2.1 Reference Expansion

References are an important feature of JSON Schema, serving as a mechanism for one schema to point to another. This functionality facilitates the reuse of schema definitions, which allows the same definitions to be reused multiple times, avoiding redundancy and improving maintainability. Reference expansion is a fundamental operation within JSON Schema, which is why we will not delve too deeply into its complexities. Instead, we will focus specifically on local references, which are references that point to other fragments within the same schema document.

While external references can be easily resolved by loading the schema from the specified

URI, they are not included in the schema collections we have been studying. Therefore, we will concentrate on the process of expanding local references, which involves replacing each occurrence of a reference with the specific part it points to within the schema.

Example 10 *To illustrate how references are expanded, consider a recursive schema that defines a person as an object with two properties:* `"name"`*, which is a string, and* `"children"`*, which is an array. Each item in the* `"children"` *array is itself a person, as indicated by the reference pointing back to the entire schema (using* `"$ref":"#"`*, which refers to the root of the current schema). This creates a recursive structure, where each person can have a list of children, and each child is also a person with their own name and potential children.*

```
{ "type": "object",
  "properties": {
    "name": { "type": "string" },
    "children": { "type": "array", "items": { "$ref": "#" } }
  }
}
```

The expansion, limited to one level of depth, produces a schema where the recursive `"children"` *property is terminated by setting the schema of* `items` *to* `false` *instead of continuing to expand the reference further. This marks the end of the recursive branch in the* `items` *assertion and prevents further recursion. While this simplifies the process, it introduces a limitation in our handling of references, which could lead to logical errors in cases where deeper recursion is required.*

```
{ "type": "object",
  "properties": {
    "name": { "type": "string" },
    "children": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "name": { "type": "string" },
          "children": { "type": "array","items": false }
        } } }
  }
}
```

This expansion is described in Figure 5.1, where we focus on the main cases. In this process, S_b denotes the *base schema* that is initially called and serves to resolve local references appearing at any arbitrary nesting level. The function *Expan* is responsible for performing this operation. It takes the schema S_b and recursively resolves the references, ensuring that each reference is substituted with the schema fragment it points to.

We also use a context C, which is simply a list of visited URIs, to keep track of the references that have already been expanded. This helps detect any cycles, preventing

$$Expan(\{\texttt{\$ref} : f\}, C) ::= Expan(gets(S_b, f), C \cup \{f\}) \text{ if } f \notin C$$
$$Expan(\{\texttt{\$ref} : f\}, C) ::= \texttt{false} \text{ if } f \in C$$
$$Expan(\{\texttt{not} : S\}, C) ::= \{\texttt{not} : Expan(S, C)\}$$
$$Expan(\{\texttt{anyOf/allOf/oneOf} : [S_1, .., S_n]\}, C) ::= \{\texttt{anyOf/allOf/oneOf} : [Expan(S_1, C), .., Expan(S_n, C)]\}$$

Figure 5.1: Reference expansion

infinite recursion when a reference points back to a schema that has already been visited. The function *Expan* checks C at each step to see if a reference has been expanded before, ensuring that the process terminates correctly.

To avoid infinite recursion in recursive schemas, we interrupt the expansion by turning the schema into `false` when recursion is detected, as illustrated in the example provided. This approach terminates the recursive branch and prevents the expansion from going further when encountering such cyclic references.

While this one-level expansion is sufficient for the schema collections we have been studying, as they do not include many deeply recursive references, it introduces a limitation. Setting the schema to `false` may lead to logical errors in some scenarios where deeper expansion is necessary. Reference expansion, being a fundamental but straightforward operation in JSON Schema, justifies this simplified approach.

In the future, the depth of expansion could be treated as a hyperparameter, offering more flexibility in handling complex schemas. Additionally, implementing a more robust reference-handling strategy would address the logical errors that may arise from prematurely terminating recursive branches.

5.2.2 Schema Canonicalization

Canonicalization is the step in our approach that aims to simplify and optimize an expanded schema for efficient data generation. The process restructures the schema into a semantically equivalent but more manageable form, ensuring it remains capable of defining the same set of valid instances. This is achieved through a set of transformation rules based on [46], which we have adapted to support Draft-2019 of JSON Schema and extended with additional rules from [25]. The canonicalization process can be broken down into the following key operations:

- **Minimization of redundant operators:** This step removes syntactic sugar by eliminating redundant operators related to the same data type. For example, constraints like `exclusiveMinimum` and `minimum` are merged, the keywords `properties` and `additionalProperties` are combined with `patternProperties`.

  In `string`-typed schemas, the value of the keyword `format` is first replaced with the regular expression corresponding to the specified format. Then, along with the string length bounds, it is integrated into `pattern` using the Brics operators, with the "&" operator being used for conjunction.

Moreover, the `const` operator, which is not specific to any type, is replaced by `enum`, with its value being an array containing only the value of `const`.

Additionally, more complex constructs like `oneOf`, `dependentSchemas`, `dependentRequired`, and `if-then-else` are rewritten into their logical equivalents using disjunctions, conjunctions, and negations. The transformation of `oneOf` follows the rule outlined in [46], while the transformations for `dependentSchemas`, `dependentRequired`, and `if-then-else` are based on the methods presented in [25], as these operators were not supported in the work discussed in [46].

- **Decomposition into homogeneous sub-schemas:** The schema is broken down into smaller sub-schemas, each containing expressions that relate to a specific data type. This decomposition helps simplify the schema structure and facilitates easier manipulation for the generation phase.

- **Filtering:** A small cleaning procedure is performed by removing keywords that appear in a schema but relate to a different type than the one described in the schema. Additionally, the `enum` array is filtered to eliminate values that are not of the same type of the schema in which it appears. This filtering of `enum` can sometimes lead to the detection of schema unsatisfiability when the enumeration becomes empty after the removal of these values.

- **Partial elimination of negation:** We perform partial negation elimination instead of complete negation removal, as discussed in Chapter 4, where negation is fully eliminated in the pessimistic technique. As mentioned in [25], achieving complete negation elimination requires extending the schema language with additional operators. In our approach, negation is only eliminated when dealing with `string` or `number`/`integer` schemas.

String Schemas. After eliminating redundant operators, the only keyword present in a `string` schema is `pattern`. Eliminating negation from this schema produces a disjunction of all the other JSON Schema data types, except for `string`, along with the complement `string` schema of the original schema. In this process, the complement schema is computed by pushing negation inside the pattern keyword, resulting in a pattern that represents the complement of the original. Unlike the pessimistic technique discussed in Chapter 4, the Brics library [60] is not utilized during this preprocessing of string schemas. This decision simplifies the transformation process by avoiding the added complexity of string operations associated with the Brics automata, allowing for a more straightforward handling of the logical structure of the schemas.

Example 11 *If we have a schema* $S =$ `{ "type": "string", "pattern": "p" }`*, the negation elimination in the schema* `{ "not": S }` *produces the following schema:*

```
{ "anyOf": [
    { "type": "null" }, { "type": "boolean" }, { "type": "integer" },
    { "type": "number" }, { "type": "object" }, { "type": "array" },
    { "type": "string", "pattern": np }
  ]
}
```

In this example, `np` *represents the complement pattern of* `"p"`*. In our implementation,* `np` *is denoted by the string* `"NOT(p)"`*. This is the pattern that will be processed during generation when the negation of strings is present.*

Number and Integer Schemas. In schemas of type `number` and `integer`, the elimination of redundant operators removes the keywords `exclusiveMinimum` and `exclusiveMaximum`. After this elimination, the schema will contain the keyword `type`, along with a combination of the keywords `minimum`, `maximum`, and `multipleOf`. Complete negation elimination occurs only when either `minimum` or `maximum` is present, or when both are included.

When the `multipleOf` keyword is present, negation is not eliminated; instead, it is pushed inside the schema, resulting in a sub-schema that includes only the `multipleOf` constraint.

Generally, negating a `number` or `integer` schema results in a disjunction of all other types, with certain exceptions. Specifically, when negating a `number` schema, `integer` is excluded from the list of allowed types. Conversely, negating an `integer` schema does not exclude `number`, as a float value remains a valid instance in this context. In this case, we introduce the additional constraint that the schema does not permit values that are multiple of 1.

Along with the other type-specific schemas, the disjunction contains a sub-schema for each keyword appearing in the original schema. For the keyword `minimum`, the negation elimination produces a sub-schema that contains the dual keyword, `maximum`, with its value being the value of `minimum` minus a small value. In our implementation, we specified $\epsilon = 1e^{-6}$. Similarly, negation elimination of `maximum` produces a sub-schema containing `minimum`, this time with its value being the value of `maximum` plus the epsilon value. Regarding `multipleOf`, the negation elimination produces a sub-schema that contains the `type` keyword along with the `not` keyword, whose corresponding schema includes only the `multipleOf` constraint.

Example 12 *Consider the following schema S:*

```
{ "type": "integer", "minimum": min, "maximum": max, "multipleOf": m }
```

The negation elimination in the schema `{ "not": S }` *produces the following schema, where* $newMax = min - \epsilon$ *and* $newMin = max + \epsilon$*.*

```
{ "anyOf": [
    { "type": "null" }, { "type": "boolean" }, { "type": "string" },
    { "type": "object" }, { "type": "array" },
    { "type": "number", "not": { "multipleOf": 1 } },
    { "type": "integer", "maximum": newMax },
    { "type": "integer", "minimum": newMin },
    { "type": "integer", "not": { "multipleOf": m } }
  ]
}
```

Other Schemas. The remaining cases involve the negation of conjunctions and disjunctions, where negation is distributed according to De Morgan's laws. Additionally, the negation elimination for `null` and `boolean` types is considered. Since there are no constraints specific to these data types, negation elimination in these schemas produces a disjunction of all other types.

For the `object` and `array` types, negation is not eliminated; the result is simply the disjunction of the other types along with the complement schema of the original schema, while retaining the negation. We chose not to eliminate negation in these cases due to the complexity it introduces in these nested structures. While it is possible to address trivial cases, such as the negation elimination of `object` schemas with only the bounds `minProperties` and `maxProperties`, this topic has been thoroughly studied in [25]. Since our work does not introduce additional contributions in this area, we focused primarily on the elimination of negation in the basic types.

Example 13 *Consider the following* `object` *schema:*

```
{ "type": "object", "minProperties": min, "maxProperties": max,
  "patternProperties": { "p1": S1, ..., "pn": Sn }
}
```

Elimination of negation in this schema produces the following schema:

```
{ "anyOf": [
    { "type": "null" }, { "type": "boolean" }, { "type": "integer" },
    { "type": "number" }, { "type": "string" }, { "type": "array" },
    { "not": {
        "type": "object", "minProperties": min, "maxProperties": max,
        "patternProperties": { "p1": S1, ..., "pn": Sn }
      }
    }
  ]
}
```

- **Merging similar assertions:** Assertions related to the same type are combined whenever possible to further simplify the schema. This step also includes identifying cases where the schema might define an empty set of valid instances, allowing for optimization or early termination. Since we rely on this merging process in various aspects of our approach, both during canonicalization and, as we will see later, in the generation phase, we will provide these rules.

  These merge rules are detailed in Figure 5.2. The first rule is straightforward and addresses the merging of schemas that pertain to incompatible types. The remaining rules are tailored to specific types and are designed to combine the constraints associated with each type while preserving their semantic integrity.

  For instance, for number types, the bounds are intersected, and the `multipleOf` arguments are combined using the least common multiple (lcm). For string types, any patterns present are combined using the Brics "&" operator for conjunction. The array rule follows a similar logic for bounds, relying on the combination of the `prefixItems` sub-schemas by invoking mergeItems, which identifies the longest common prefix and applies `items` to the sub-schemas outside that common prefix.

  The `contains` constraints, which are inherently existential, are combined using `allOf` to preserve their semantics. It is essential to emphasize that each schema in `contains` must be satisfied individually. Consequently, combining these schemas may lead to contradictions, resulting in scenarios where the overall conjunction may be unsatisfiable. As a result, we retain these schemas in the `allOf` array, which allows us to maintain the integrity of each constraint. This necessity introduces one of the limitations our approach faces.

  The logic for object types is straightforward, involving the intersection of bounds, the collection of required labels from both schemas, and the merging of properties by combining schemas associated with the same patterns using $mergeProps$.

The canonicalized schemas are generally expressed in disjunctive normal form (DNF). However, in certain cases, schemas cannot be merged due to the presence of negation, particularly when dealing with object and array schemas.

The structure of these canonicalized schemas is captured by the grammar in Figure 5.3, which describes the general schemas S in DNF. The conjuncts of $\mathcal{S}$ are built from positive *typed assertions* (TA) and their negated forms ($notTA$), which primarily apply to object and array assertions. For number assertions, negation can still appear before the `multipleOf` operator as expected, but this does not hinder the merging of assertions.

$$\frac{S_1 = \{\, \texttt{type} : \mathsf{T}, \vec{K} \,\} \qquad S_2 = \{\, \texttt{type} : \mathsf{T}', \vec{K}' \,\} \qquad \mathsf{T} \neq \mathsf{T}'}{\{\, \texttt{allOf} : [\, S_1, S_2 \,] \,\} \rightarrow false} \text{ (heterogenous types)}$$

$$\frac{\begin{array}{c} S_i = \{\, \texttt{type} : \texttt{number}, \texttt{minimum} : m_i, \texttt{maximum} : M_i, \texttt{multipleOf} : mof_i \,\} \\ i = 1, 2 \qquad m = max(m_1, m_2) \qquad M = min(M_1, M_2) \qquad l = lcm(mof_1, mof_2) \end{array}}{\{\, \texttt{allOf} : [\, S_1, S_2 \,] \,\} \rightarrow \{\, \texttt{type} : \texttt{number}, \texttt{minimum} : m, \texttt{maximum} : M, \texttt{multipleOf} : l \,\}} \text{ (intersect number)}$$

$$\frac{S_i = \{\, \texttt{type} : \texttt{string}, \texttt{pattern} : p_i \,\} \qquad i = 1, 2}{\{\, \texttt{allOf} : [\, S_1, S_2 \,] \,\} \rightarrow \{\, \texttt{type} : \texttt{string}, \texttt{pattern} : p_1 \ \& \ p_2 \,\}} \text{ (intersect string)}$$

$$\frac{\begin{array}{c} S_i = \{\texttt{type} : \texttt{array}, \texttt{minItems} : m_i, \texttt{maxItems} : M_i, \texttt{prefixItems} : [\, S^i_1, \ldots . S^i_{n_i} \,], \texttt{items} : S'_i, \\ \texttt{contains} : S_{c_i}, \texttt{minContains} : mc_i, \texttt{maxContains} : Mc_i\} \\ i = 1, 2 \qquad m = max(m_1, m_2) \qquad M = min(M_1, M_2) \\ pItems = mergeItems([\, S^1_1, \ldots . S^1_{n_1} \,], [\, S^2_1, \ldots . S^2_{n_2} \,]) \end{array}}{\begin{array}{c} \{\, \texttt{allOf} : [\, S_1, S_2 \,] \,\} \rightarrow \{\texttt{type} : \texttt{array}, \texttt{minItems} : m, \texttt{maxItems} : M, \\ \texttt{prefixItems} : pItems, \texttt{items} : \{\, \texttt{allOf} : [\, S'_1, S'_2 \,] \,\}, \\ \texttt{allOf} : [\{\, \texttt{contains} : S_{c_1}, \texttt{minContains} : mc_1, \texttt{maxContains} : Mc_1 \,\}, \\ \{\, \texttt{contains} : S_{c_2}, \texttt{minContains} : mc_2, \texttt{maxContains} : Mc_2 \,\}]\} \end{array}} \text{ (intersect array)}$$

$$\frac{\begin{array}{c} S_i = \{\texttt{type} : \texttt{object}, \texttt{minProperties} : m_i, \texttt{maxProperties} : M_i, \texttt{required} : [\, k^i_1, \ldots, k^i_n \,], \\ \texttt{patternProperties} : \{\, p^i_1 : S^i_1, \ldots, p^i_{l_i} : S^i_{l_i} \,\}, \texttt{propertyNames} : S^i\} \\ i = 1, 2 \qquad m = max(m_1, m_2) \qquad M = min(M_1, M_2) \\ pattProps = mergeProps(\{\, p^1_1 : S^1_1, \ldots, p^1_{l_1} : S^1_{l_1} \,\}, \{\, p^2_1 : S^2_1, \ldots, p^2_{l_2} : S^2_{l_2} \,\}) \end{array}}{\begin{array}{c} \{\, \texttt{allOf} : [\, S_1, S_2 \,] \,\} \rightarrow \{\texttt{type} : \texttt{object}, \texttt{minProperties} : m, \texttt{maxProperties} : M, \\ \texttt{required} : [\, k_1, \ldots, k_n \,] \cup [\, k'_1, \ldots, k'_l \,], \\ \texttt{patternProperties} : pattProps, \texttt{propertyNames} : \{\, \texttt{allOf} : [\, S, S' \,] \,\}\} \end{array}} \text{ (intersect object)}$$

Figure 5.2: Conjunction elimination rules

$b \in \{\!|\, false, true \,|\!\}, q \in \mathsf{Num}, i \in \mathbb{N}, k \in \mathsf{Str}, p \in \mathsf{Str}, J \in \mathit{JVal}$

$$
\begin{array}{lcl}
S & ::= & \texttt{true} \mid \texttt{false} \mid \mathcal{S} \mid \{\ \texttt{anyOf} : [\, \mathcal{S}\ (, \mathcal{S})^{+}\,]\ \} \\
\mathcal{S} & ::= & \{\, \texttt{allOf} : [\, \mathit{TA}^{?}\ (, \mathit{NotTA})^{+}\,]\ \} \mid \{\ \mathit{TA}\ (, \texttt{enum} : J)^{?}\ \} \mid \mathit{notTA} \\
\mathit{TA} & ::= & \mathit{NullTA} \mid \mathit{BoolTA} \mid \mathit{NumTA} \mid \mathit{IntTA} \mid \mathit{StrTA} \mid \mathit{ArrTA} \mid \mathit{ObjTA} \\
\mathit{NullTA} & ::= & \texttt{type} : \texttt{null} \\
\mathit{BoolTA} & ::= & \texttt{type} : \texttt{boolean} \\
\mathit{NumTA} & ::= & \texttt{type} : \texttt{number}\ (, \texttt{minimum} : q)^{?} (, \texttt{maximum} : q)^{?} (, \texttt{multipleOf} : q)^{?} \\
 & & (, \texttt{not} : \{\, \texttt{multipleOf} : q\, \})^{?} \\
\mathit{IntTA} & ::= & \texttt{type} : \texttt{integer}\ (, \texttt{minimum} : q)^{?} (, \texttt{maximum} : q)^{?} (, \texttt{multipleOf} : q)^{?} \\
 & & (, \texttt{not} : \{\, \texttt{multipleOf} : q\, \})^{?} \\
\mathit{StrTA} & ::= & \texttt{type} : \texttt{string}\ (, \texttt{pattern} : p)^{?} \\
\mathit{ArrTA} & ::= & \texttt{type} : \texttt{array}\ (, \texttt{minItems} : i)^{?} (, \texttt{maxItems} : i)^{?} (, \texttt{uniqueItems} : b)^{?} \\
 & & (, \texttt{items} : S)^{?} (, \texttt{prefixItems} : [\, S_1, \ldots, S_n\,])^{?} \\
 & & (\mathit{contS} \mid \{\, \texttt{allOf} : [\, \mathit{contS} (, \mathit{contS})^{+}\,]\ \})^{?} \\
\mathit{ObjTA} & ::= & \texttt{type} : \texttt{object}\ (, \texttt{minProperties} : i)^{?} (, \texttt{maxProperties} : i)^{?} \\
 & & (, \texttt{required} : [\, k_1, \ldots, k_n\,])^{?} (, \texttt{patternProperties} : \{\, p_1 : S_1, \ldots, p_m : S_m\, \})^{?} \\
 & & (, \texttt{propertyNames} : S)^{?} \\
\mathit{notTA} & ::= & \{\, \texttt{not} : \{\ \mathit{ObjTA}\ \}\ \} \mid \{\, \texttt{not} : \{\ \mathit{ArrTA}\ \}\ \} \\
\mathit{contS} & ::= & (, \texttt{minContains} : i)^{?} (, \texttt{maxContains} : i)^{?} (, \texttt{contains} : S)^{?}
\end{array}
$$

Figure 5.3: Generation grammar

Example 14 *To illustrate the utility of canonicalization, consider the following schema, which consists of two sub-schemas, each implicitly expressing a disjunction of conditions:*

```
{ "allOf": [
    { "type": ["object","number","array"], "required": ["a","e"],
      "properties": { "a": S1, "e": S2 },
      "patternProperties": { "(a|b)*": S3, "(a|e)+": S4 },
      "multipleOf": 3, "contains": { "type": "string" }
    },
    { "type": ["number","array"], "minimum": 7, "minItems": 3 }
  ]
}
```

During canonicalization, this implicit disjunction is made explicit, allowing assertions to be grouped by type (object, number, array, etc.). The process then distributes conjunctions over disjunctions, transforming the schema into disjunctive normal form (DNF), while attempting to eliminate negation where possible.

```
{ "allOf": [
    { "anyOf": [
      { "type": "object", "required": ["a","e"],
        "properties": { "a": S1, "e": S2},
        "patternProperties": { "(a|b)*": S3, "(a|e)+": S4 },
      },
      { "type":"number", "multipleOf": 3 },
      { "type": "array", "contains": { "type":"string" } }
    ] },
    { "anyOf": [
      { "type": "number", "minimum": 7 },
      { "type": "array", "minItems": 3 }
    ] }
  ]
}
```

Next, syntactic sugar is removed by combining `properties` *into* `patternProperties`*, resulting in the final canonicalized schema:*

```
{ "anyOf": [
    { "type":"object", "required": ["a","e"],
      "patternProperties": { "a": S1, "e": S2, "(a|b)*": S3, "(a|e)+": S4 }
    },
    { "type":"number", "multipleOf": 3, "minimum": 7 },
    { "type": "array", "minItems": 3, "contains": { "type":"string" } }
  ]
}
```

5.3 The generation Phase

In this section, we outline the process of generating multiple distinct valid instances for a given schema, building on the foundations established in the preprocessing phase. Our goal is to extend the problem of single instance generation introduced in Chapter 4 to efficiently produce up to N distinct valid instances for a schema S. Unlike the pessimistic approach described in Chapter 4, which exhaustively explores every possible solution, our optimistic approach prioritizes speed, accepting some incompleteness when processing certain schema constructs, particularly those involving negated object and array sub-schemas.

Despite this incompleteness, correctness remains a key priority. This means that when a schema is *satisfiable*, our method will generate valid instances that adhere to the schema's constraints. Conversely, when a schema is *unsatisfiable*, our method will, in most cases, accurately determine that no valid instances can be generated. By ensuring that satisfiable schemas yield valid outputs and unsatisfiable schemas are flagged, the approach maintains soundness while balancing efficiency with completeness.

The generation phase consists of two key steps, which are detailed in the following sections: unsatisfiability checking and the generation algorithms, which include the main algorithm and the algorithms specific to each JSON Schema type.

5.3.1 Unsatisfiability Checking

Before proceeding to instance generation, we introduce a critical step in the process: checking for unsatisfiability. Once the schema has undergone canonicalization during the preprocessing phase, this step aims to detect inherently unsatisfiable schemas early, thereby preventing unnecessary computation. The detection of unsatisfiability is achieved by *statically* analyzing the canonicalized schema for contradictory specifications. Most cases of schema emptiness can be identified through this static analysis; however, there are certain complex cases where static analysis is insufficient and requires a more involved approach, which highlights a limitation of the method and makes the detection incomplete. In such cases, unsatisfiability detection must be handled *dynamically* during the generation step to avoid increased computational costs, which we do not currently implement.

Specifically, the schemas discarded during this step are those that do not admit even a single valid instance. Determining unsatisfiability when requiring N distinct instances is more challenging and may require additional processing during the generation phase.

This step ensures that our method, while optimistic, maintains correctness by refraining from attempting to generate instances for unsatisfiable schemas, despite its incompleteness and the potential oversight of some cases.

This detection is performed by a recursive function *isUnsat* that determines whether a canonicalized schema is unsatisfiable. This function is defined as a set of rules denoted by the judgment $S \rightarrow r$, taking a schema S as its parameter and returning a boolean value r, where $r = \mathcal{T}\ (true)$ indicates that the schema S is unsatisfiable. The analysis is based on the content of the schema: if it is a typed schema, the function verifies certain conditions specific to that type, as we have defined a set of rules for each data type. If the schema is a

disjunction, the *isUnsat* function is run on each branch. If the function returns $\mathcal{F}$ ($false$) for any branch, we stop the verification process. Conversely, if the schema is a conjunction, we return $\mathcal{F}$, as unsatisfiability cannot be detected in this case.

Universally Satisfiable and Unsatisfiable Schemas. The following rules capture the universally `true` schema, indicating that it admits valid instances, and the universally `false` schema, which indicates that it does not admit any valid instances.

$$\texttt{false} \ \rightarrow \mathcal{T} \qquad (\texttt{false-schema})$$

$$\texttt{true} \ \rightarrow \mathcal{F} \qquad (\texttt{true-schema})$$

Conjunctive and Dijsunctive Schemas. The first rule applies to conjunctive schemas, where the function always returns $\mathcal{F}$, indicating that unsatisfiability cannot be statically determined for such schemas. The second rule handles disjunctive schemas, and checks if all branches of the disjunction are unsatisfiable. If *isUnsat* returns $\mathcal{T}$ for every branch, the schema is deemed unsatisfiable.

$$\{\, \texttt{allOf} : [\, S_1, \ldots, S_n \,] \,\} \ \rightarrow \mathcal{F} \qquad (\texttt{allOf-schema})$$

$$\frac{S = \{\, \texttt{anyOf} : [\, S_1, \ldots, S_n \,] \,\} \qquad r = \wedge_{i=1}^{n} isUnsat(S_i)}{S \ \rightarrow r} \qquad (\texttt{anyOf-schema})$$

Number and Integer Types. The following rules define the conditions under which the function *isUnsat* will determine that a schema of type `number` or `integer` is unsatisfiable. In this context, $\vec{K}$ contains the set of keywords that are permitted after canonicalization; these include those specific to these types or `enum` and are not already present in S.

The first two rules are trivial. The rule (`number-Mof-0`) returns $\mathcal{T}$ when specifying that a number should be a multiple of 0, as no number can be a multiple of 0. The rule (`number-bounds`) checks whether the value of `minimum` is greater than that of `maximum`, resulting in an unsatisfiable schema if this condition holds.

$$\frac{\begin{array}{c} S = \{\, \texttt{type} : \texttt{number}, \texttt{multipleOf} : m, \vec{K} \,\} \\ r = (m = 0) \end{array}}{S \ \rightarrow r} \qquad (\texttt{number-Mof-0})$$

$$\frac{\begin{array}{c} S = \{\, \texttt{type} : \texttt{number}, \texttt{minimum} : min, \texttt{maximum} : max, \vec{K} \,\} \\ r = (min > max) \end{array}}{S \ \rightarrow r} \qquad (\texttt{number-bounds})$$

The following rule checks whether there exists a multiple of m within the interval $[min, max]$. If no such multiple exists, the schema is deemed unsatisfiable. The boolean

variable r is the negation of what the function *mofInInterval* returns; thus, if *mofInInterval* returns $\mathcal{F}$, indicating there are no multiples of m in the specified range, r will be $\mathcal{T}$, indicating that the schema S is unsatisfiable.

The function *mofInInterval* processes this by calculating the smallest multiple of m that is greater than or equal to min and the largest multiple of m that is less than or equal to max. It compares these two values, returning $\mathcal{T}$ if a valid multiple exists within the range. Specifically, the implementation involves computing the ceiling of $\frac{min}{m}$ multiplied by m to find the lower bound, and the floor of $\frac{max}{m}$ multiplied by m for the upper bound. A small epsilon is added to the upper bound comparison to mitigate potential floating-point errors.

$$\frac{S = \{\, \texttt{type} : \texttt{number}, \texttt{minimum} : min, \texttt{maximum} : max, \texttt{multipleOf} : m, \vec{K} \,\} \qquad r = \neg \mathit{mofInInterval}(m, min, max)}{S \ \rightarrow r} \ (\texttt{number-Mof-Interval})$$

The following rule verifies whether m is a multiple of m'. Given that the schema explicitly states that valid instances must not be multiples of m', the function *isUnsat* will return $\mathcal{T}$ if m is indeed a multiple of m', indicating that the schema is unsatisfiable.

The function *isMultiple* processes this by first checking if m' is 0 to prevent division by 0, returning $\mathcal{F}$ in that case. If m' is not equal to 0, it calculates the result of the division $\frac{m}{m'}$ and checks if this result is a whole number. If the division yields a whole number, it confirms that m is a multiple of m', leading to the conclusion that the schema S is unsatisfiable.

$$\frac{S = \{\, \texttt{type} : \texttt{number}, \texttt{multipleOf} : m, \texttt{not} : \{\, \texttt{multipleOf} : m' \,\}, \vec{K} \,\} \qquad r = \mathit{isMultiple}(m, m')}{S \ \rightarrow r} \ (\texttt{number-Mof-notMof})$$

String Type. For `string` schemas, the only rule defined checks whether the language of the pattern is empty, meaning that if no string value matches the pattern, the schema is unsatisfiable.

The verification process is performed using the Brics library [60], which constructs an automaton corresponding to the regular expression defined by the pattern. We then check whether this automaton is empty.

$$\frac{S = \{\, \texttt{type} : \texttt{string}, \texttt{pattern} : p, \vec{K} \,\} \qquad r = (\mathcal{L}(p) = \emptyset)}{S \ \rightarrow r} \ (\texttt{string-empty})$$

Object Type. Detecting emptiness in `object` schemas is more involved than in basic types. In most cases, determining unsatisfiability cannot be done statically; it often requires

dynamic verification of the domains of patterns present in `patternProperties`, among other aspects. However, as stated before, we will focus here on the simpler cases that can be detected statically. The following rules address these cases.

The first two rules are trivial and relate to the verification of the bounds. The rule (`object-bounds`) checks whether the schema contains a minimal bound with a value higher than the maximal bound. The rule (`object-reqMax`) verifies whether the number of required properties exceeds the maximal bound. Any schema that meets these conditions is considered unsatisfiable.

$$\frac{S = \{\, \texttt{type} : \texttt{object}, \texttt{minProperties} : min, \texttt{maxProperties} : max, \vec{K} \,\} \quad r = (min > max)}{S \ \rightarrow r} \ (\texttt{object-bounds})$$

$$\frac{S = \{\, \texttt{type} : \texttt{object}, \texttt{required} : [\, k_1, \ldots, k_n \,], \texttt{maxProperties} : max, \vec{K} \,\} \quad r = (n > max)}{S \ \rightarrow r} \ (\texttt{object-reqMax})$$

The following rule addresses the verification of required properties against the schema defined by `propertyNames`, which describes string values, either as a string schema or an enumeration of string values. This rule checks whether any required property fails to conform to the schema of `propertyNames`. The conformity is assessed using the function LS which builds an automaton for the language defined by the `propertyNames` schema. Each required property is then evaluated against the automaton, and if any property is not accepted, it indicates that the schema is unsatisfiable.

$$\frac{S = \{\, \texttt{type} : \texttt{object}, \texttt{required} : [\, k_1, \ldots, k_n \,], \texttt{propertyNames} : S_{str}, \vec{K} \,\} \quad r = \exists i \in \{\!|1 \ldots n|\!\}.\ k_i \notin LS(S_{str})}{S \ \rightarrow r} \ (\texttt{object-reqPNames})$$

The following rule checks for the existence of a required property that belongs to the language of a pattern whose corresponding schema is unsatisfiable. If such a property is identified, the entire object schema is then unsatisfiable.

$$\frac{S = \{\, \texttt{type} : \texttt{object}, \texttt{required} : [\, k_1, \ldots, k_n \,], \texttt{patternProperties} : \{\, p_1 : S_1, \ldots, p_m : S_m \,\}, \vec{K} \,\} \quad r = \exists i \in \{\!|1 \ldots n|\!\}, j \in \{\!|1 \ldots m|\!\}.\ (k_i \in \mathcal{L}(p_j)) \wedge isUnsat(S_j)}{S \ \rightarrow r} \ (\texttt{object-reqPatt})$$

Array Type. Similar to `object` schemas, detecting schema emptiness for `array` schemas involves additional complexity in many cases. Figure 5.4 outlines the rules for cases where unsatisfiability can be detected statically without complex computations.

Most of these rules address contradictory bounds, where minimal bounds are greater than the maximal bounds. This is captured by rules (array-bounds) and (array-contBounds). In the former, we use `minItems`/`minContains` to avoid repeating the rule twice, as it applies to both `minItems` and `minContains`.

Other rules cover scenarios where the schemas of `prefixItems` are insufficient to meet the minimal bounds, and `items` is present with an unsatisfiable schema (rule (array-itUnsat)), with the minimal bound being either `minItems` or `minContains`.

Further rules address cases related to the `contains` keyword. Rule (array-contUnsat) indicates that a schema S is unsatisfiable when the `contains` keyword contains an unsatisfiable schema. Additionally, rule (array-contMax) specifies that a schema with a maximal bound of 0 in the presence of `contains` is also unsatisfiable.

Finally, the last rule (array-plItemsUnsat) captures unsatisfiability in schemas containing the `prefixItems` keyword, where an unsatisfiable schema appears at an index smaller than the minimal bound.

$$\frac{S = \{\, \texttt{type} : \texttt{array}, \texttt{minItems/minContains} : min, \texttt{maxItems} : max, \vec{K} \,\} \quad r = (min > max)}{S \ \rightarrow r} \ \text{(array-bounds)}$$

$$\frac{S = \{\, \texttt{type} : \texttt{array}, \texttt{contains} : S_c, \texttt{minContains} : minC, \texttt{maxContains} : maxC, \vec{K} \,\} \quad r = (minC > maxC)}{S \ \rightarrow r} \ \text{(array-contBounds)}$$

$$\frac{S = \{\, \texttt{type} : \texttt{array}, \texttt{prefixItems} : [\, S_1, \ldots, S_n \,], \texttt{items} : S_{it}, \texttt{minItems/minContains} : min, \vec{K} \,\} \quad r = ((n < min) \wedge isUnsat(S_{it}))}{S \ \rightarrow r} \ \text{(array-itUnsat)}$$

$$\frac{S = \{\, \texttt{type} : \texttt{array}, \texttt{contains} : S_c, \vec{K} \,\} \quad r = isUnsat(S_c)}{S \ \rightarrow r} \ \text{(array-contUnsat)}$$

$$\frac{S = \{\, \texttt{type} : \texttt{array}, \texttt{contains} : S_c, \texttt{maxItems} : max, \vec{K} \,\} \quad r = (max = 0)}{S \ \rightarrow r} \ \text{(array-contMax)}$$

$$\frac{S = \{\, \texttt{type} : \texttt{array}, \texttt{prefixItems} : [\, S_1, \ldots, S_n \,], \texttt{minimum/minContains} : min, \vec{K} \,\} \quad r = \exists i \in \{\!|1 \ldots n|\!\}.\ (i < min) \wedge isUnsat(S_i)}{S \ \rightarrow r} \ \text{(array-plItemsUnsat)}$$

Figure 5.4: Unsatisfiability rules for array schemas

In addition to all the previously mentioned rules, the verification process for type-

specific schemas that involve the `enum` keyword entails validating each value in the enumeration against the schema. If a valid value is found, the verification process stops, indicating that the schema admits an instance. Conversely, if none of the values in the enumeration satisfy the schema's requirements, then the schema is deemed unsatisfiable. This validation is conducted using the validator described in [3].

Additionally, during the verification of the `enum` items, we filter out and retain only the values that are valid against the schema. This step simplifies the generation process, ensuring that the remaining values in the enumeration are directly usable for instance generation without the need for further checks.

5.3.2 Main Generation Algorithm

Once the schema has been checked for unsatisfiability and determined to be satisfiable, the generation process begins, starting with the main algorithm. This algorithm takes the schema S and the desired number of instances N as input and attempts to generate up to N distinct *valid* instances that conform to the schema.

The goal of this approach is to extend the *witness generation problem* introduced in Chapter 4, by producing multiple distinct valid instances rather than a single one. However, our optimistic solution is inherently *incomplete*, as it may not be able to handle certain schema classes, particularly those involving negated object or array sub-schemas. When this is the case, we return an indication to express this incompleteness through a generation failure signal, meaning that the process cannot proceed with generating instances. When the schema can be successfully processed, the algorithm returns M instances, where $M \leq N$, indicating that it may generate fewer than the desired number of instances.

The generation proceeds through a case-based analysis of the schema, which is managed by two mutually recursive functions: *Gen*, responsible for processing schemas in partially disjunctive normal form (denoted S), and *GenS*, which handles disjuncts (denoted $\mathcal{S}$). Both functions are detailed in Figure 5.5.

The first line of *Gen* handles the universally satisfied schema `true`, generating a default set of values, typically integers from 1 to N. The second line handles disjunctive expressions by generating values from a single disjunct, specifically the one that can produce N instances or a set of values closest to N. Importantly, we do not accumulate instances from multiple disjuncts to avoid generating duplicate values, as this would complicate the process. This choice introduces a minor limitation: we may end up generating fewer than N instances, even when the overall schema could potentially admit N or more distinct instances. If none of the disjuncts can produce a result, the process returns GenFail indicating generation failure.

For *GenS*, the logic is straightforward: generation fails in the presence of negated assertions (as seen in the first and third lines). In cases involving the `enum` keyword, since the enumeration has been cleaned to only contain valid values against the schema (cf. Filtering 5.2.2), *GenS* simply selects N values or the maximum it can generate. For other types, the function relies on type-specific algorithms for generating instances corresponding to basic and complex types.

$$
\begin{array}{lcl}
Gen(\texttt{true}, N) & ::= & \{\!|1, \ldots, N|\!\} \\
Gen(\{\,\texttt{anyOf} : [\,\mathcal{S}_1, \ldots, \mathcal{S}_n\,]\,\}, N) & ::= & GenS(\mathcal{S}_i, N) \text{ s.t.} GenS(\mathcal{S}_i, N) \neq GenFail \text{ and for } i, j = 1..n \\
 & & |GenS(\mathcal{S}_i, N)| = Min(N, Max_j\{\!|\,|GenS(\mathcal{S}_j, N)|\,|\!\}) \\
 & & GenFail \text{ otherwise} \\
 & & \\
GenS(\{\,\texttt{allOf} : [\,TA, NotTA\,]\,\}, N) & ::= & GenFail \\
GenS(\{\,TA, \texttt{enum} : [\,J_1, \ldots, J_n\,])\,\}, N) & ::= & trunc(N, \{\!|J_i \mid J_i \;?\; TA \mapsto \mathcal{T}|\!\}) \\
GenS(NotTA, N) & ::= & GenFail \\
GenS(TA, N) & ::= & \text{see specific algorithms}
\end{array}
$$

Figure 5.5: Instance generation: main algorithm

5.3.3 Generation of Basic Types

In this section, we describe the methodology employed for generating instances of basic data types, including `null`, `boolean`, `number`/`integer`, and `string`. The generation of `null` type is straightforward, as it can only yield the single value `null`. Similarly, for the `boolean` type, the generation process returns the values $\mathcal{F}$, $\mathcal{T}$, or both when the desired number of distinct instances $N \geq 2$.

The main focus lies in generating `number` and `string` types, where the approach becomes more nuanced. For `number` (generation for `integer` is similar), we generalize the function defined in the pessimistic approach [11, 12], aiming to produce N distinct values. This is achieved through an iterative process that updates the interval bounds based on previously generated values, ensuring that the values fall within the specified `minimum` and `maximum` constraints, while satisfying the argument of `multipleOf` and violating the argument of a negated `multipleOf`.

For `string`, the generation of N distinct values relies on the Brics library [60]. This involves constructing an initial automaton that accepts the language defined by the regular expression of the `pattern` constraint. To avoid regenerating previously created values, we iteratively update the automaton by intersecting it with a complement automaton that excludes the newly generated value.

In the following we delve into the specifics of these generation processes, highlighting their underlying logic and addressing the challenges encountered.

Number/Integer Types. The generation of numerical values proceeds through an iterative process aimed at producing N distinct values whenever possible while adhering to the specified constraints. The approach varies depending on the presence or absence of the following constraints: `multipleOf`, `not` (with its corresponding schema containing `multipleOf`, noted here as `notMof`), `minimum`, and `maximum`.

With `multipleOf` constraint:

1. If `minimum` is defined: The algorithm identifies the nearest multiple of the value of the `multipleOf` constraint that is greater than `minimum` and that it is not a multiple

of `notMof` if present.

2. If only `maximum` is defined: The search focuses on the nearest multiple of the value of the `multipleOf` that is less than `maximum` and does not violate the `notMof` constraint.

3. If neither `minimum` nor `maximum` is defined: The algorithm selects the `multipleOf` value directly, as it has been verified to not be a multiple of `notMof` during unsatisfiability checking. Here, ϵ is set to $1e^{-6}$.

After generating a valid number v_i, if `minimum` is defined, it is updated to $v_i + \epsilon$; if only `maximum` is present, it is adjusted to $v_i - \epsilon$. If neither is specified, a new `minimum` constraint is introduced with a value of $v_i + \epsilon$ during the first iteration. This update aims at excluding the previously generated value from being generated again.

When `multipleOf` is not present but `notMof` is present:

1. If `minimum` is defined: The algorithm identifies the nearest value greater than `minimum` that is not a multiple of the value of the `notMof` constraint.

2. If only `maximum` is defined: The algorithm looks for the nearest value less than `maximum` that is not a multiple of the value of the `notMof` constraint.

3. If neither `minimum` nor `maximum` is defined: The algorithm starts from zero and moves upwards, ensuring that the generated values are not multiples of the value of the `notMof` constraint.

The updates of `minimum` and `maximum` are performed similarly to the previous case. If neither bound constraint is present, a new `minimum` constraint is introduced during the first iteration.

When neither `multipleOf` nor `notMof` is present: When neither the `multipleOf` nor `notMof` constraints exist, the generation starts with `minimum` (if defined) and moves upwards, or with `maximum` (if only that is specified) and moves downwards. If neither `minimum` nor `maximum` is provided, the algorithm begins from zero and moves upwards. It continually updates the values of `minimum` and `maximum`, ensuring that a new `minimum` constraint is introduced with a value of the generated value plus ϵ after the first iteration if no bound constraints are present.

String Type. After canonicalization, string schemas contain only the `pattern` keyword, with potential negations expressed using the notation `"Not(p)"`, where `p` represents the negated pattern. This may also involve conjunctions of negations, nested negations, and other combinations. To address this and facilitate generation with Brics, we use a function, *patternToAutomaton*, that creates an automaton starting from a pattern. This function decomposes the entire pattern if it consists of multiple components, builds separate automatons for each part, and combines them to create a single automaton that defines the

language of the full pattern. For negated patterns, the automaton returned is the complement of the automaton built from the regular expression of the pattern. For instance, in the case of the negation `"Not(p)"`, where `p` is a single pattern, the function first constructs the automaton for `p` and then returns its complement automaton.

The function also makes minimal adjustments to the patterns, aiming to align them with Brics' requirements as closely as possible. As mentioned in Remark 3 of Chapter 4, the pessimistic approach performs a complete translation from the regular expression syntax of JSON Schema to Brics. However, for our optimistic approach, we opt for a simpler strategy, translating only essential JSON metacharacters. For instance, the metacharacter `"\d"` is replaced with `"[0-9]"`, and `"\w"` is replaced with `"[a-zA-Z0-9_]"` to ensure compatibility with Brics.

The generation of the N distinct string values is then performed iteratively, starting with the automaton returned by the *patternToAutomaton* function. This automaton, denoted as $\mathcal{A}$, accepts the language defined by the regular expression of the `pattern` constraint. In each iteration, we generate a new string value w and exclude it from the automaton $\mathcal{A}$ to prevent regeneration. This exclusion is achieved by updating the automaton such that $\mathcal{A} = \mathcal{A} \cap \mathcal{A}'$ where $\mathcal{A}'$ is the complement of the automaton that accepts only w.

Given our incomplete translation, some scenarios may result in potential inaccuracies or false results. We do not opt for a full translation to Brics because it does not provide any substantial contribution to our work. Instead, we focus on covering the limited set of patterns commonly observed in the usage of JSON Schema. This targeted approach allows us to effectively address the most relevant scenarios without unnecessary complexity or computational overhead, ensuring that our generation process remains efficient and manageable.

5.3.4 Object Types

Given an object schema

$$
\begin{aligned}
&ObjTA ::= \\
&\{\ \texttt{type}\text{: object}, \texttt{minProperties}\text{: min}, \texttt{maxProperties}\text{: max}, \\
&\quad \texttt{patternProperties}\text{:}\{p_1 : S_1, ..., p_n : S_n\}, \\
&\quad \texttt{required}\text{:}[l_1, ..., l_k] \\
&\quad \texttt{propertyNames}\text{: } S_{str}\ \ \}
\end{aligned}
$$

and the desired number N of distinct objects, generation is performed by the following Algorithm 1. It first invokes SatReq (cf. Algorithm 2) in order to generate the values of the *required* fields. In case it is not possible to generate a value for at least one of the required fields, generation *fails*. Otherwise, generation goes on by calling SatMinProp (cf. Algorithm 3) which checks whether `minProperties` is satisfied and, if not, it generates additional fields by attempting to produce enough values for each additional field to increase the number of possible objects that can be obtained. Then, if the sets of generated values for fields satisfying `minProperties` are sufficient to obtain at least N objects, we return the

generated instances; Otherwise, if it is possible to add new fields or replace non-required existing ones (as verified by $|$ `required` $|<$ `maxProperties`), MoreInstances function (cf. Algorithm 4) is called to generate the missing instances. This is achieved by adding more fields while adhering to the `maxProperties` constraint in order to reach N objects.

Algorithm 1: main object generation algorithm

Data: An object assertion $ObjTA$, an integer N
Result: $M \leq N$ instances $\mid$ $GenFail$

```
1  Res = {}
2  objMap = SatReq(ObjTA, N)
3  if objMap ≠ GenFail then
4      objMap = SatMinProp(ObjTA, objMap, N)
5      if objMap ≠ GenFail then
6          Res = objProduct(objMap)
7          M = |Res|
8          if M < N and |required| < maxProperties then
9              return Res ∪ MoreInstances(ObjTA, objMap, N−M)
10         else
11             return Res
12 return GenFail
```

Example 15 *To illustrate the generation of objects, consider the following normalized object schema which requires every valid object to have at least 4 properties, two of which must have the keys* `"a"` *and* `"b"`.

```
{ "type": "object", "minProperties": 4, "maxProperties": 5,
  "patternProperties": {
    "a": { "type": "integer", "minimum": 1, "maximum": 4 },
    "b": { "type": "integer", "minimum": 1, "maximum": 1 },
    "c": { "type": "integer", "minimum": 1, "maximum": 5 },
    "d": { "type": "integer", "minimum": 1, "maximum": 4 },
    "e": { "type": "integer", "minimum": 1, "maximum": 4 },
    "a.*": { "type": "integer", "multipleOf": 2 }
  },
  "required": ["a","b"]
}
```

Suppose we want to generate 100 distinct objects that satisfy this schema. First, we deal with required fields (Algorithm 2 - SatReq) and we try to generate 100 distinct values for the key `"a"` *but there are only two values possible, hence* $V_a = [2, 4]$. *For the next required key, which is* `"b"` *we try to generate* $\lceil \frac{100}{2} \rceil = 50$ *values, but the generation returns one value,* $V_b = [1]$. *We continue by satisfying* `minProperties` *(Algorithm 3 - SatMinProp)*

and for `"c"` *we try to generate* $\lceil \frac{50}{1} \rceil = 50$ *but we get* $V_c = [1, 2, 3, 4, 5]$*, finally, we try to generate* $\lceil \frac{50}{5} \rceil = 10$ *for* `"d"` *by we only get* $V_d = [1, 2, 3, 4]$*. The number of instances we can build using the pairs key-set of values is:* $M = |V_a| * |V_b| * |V_c| * |V_d| = 2 * 1 * 5 * 4 = 40$ *instances.*

The `minProperties` *assertion is now met, but we are still missing* $L = N - M = 60$ *instances, and since the maximum number of properties per object is not reached yet which is* 5*, we can introduce a new property, and to this end (Algorithm 4 - MoreInstances) we pick* `"e"`*, which has not yet been considered by generation. Observe that in order to obtain new records we can use values for required attributes* `"a"` *and* `"b"` *already dealt with, and combine with values for* `"e"` *either those for* `"c"` *or those for* `"d"` *or values for both* `"c"` *and* `"d"`*. Each of these combinations yields a record satisfying both* `minProperties` *and* `maxProperties`*. In more details, we have* $req = \{\!|\texttt{"a"}, \texttt{"b"}|\!\}$*, hence we set* $P_{req} = |V_a| * |V_b| = 2 * 1 = 2$*, and* $nonReq = \{\!|\texttt{"c"}, \texttt{"d"}|\!\}$*, hence the subsets of* $nonReq$ *that we can associate to the new key* `"e"` *and to the set* req *of required properties is:* $subs = \{\!|\{\!|\texttt{"c"}|\!\}, \{\!|\texttt{"d"}|\!\}, \{\!|\texttt{"c"}, \texttt{"d"}|\!\}|\!\}$*, hence we have:* $\sum_{i=1}^{|subs|} P_i = P_1 + P_2 + P_3 = |V_c| + |V_d| + |V_c| * |V_d| = 5 + 4 + 20 = 29$*. Finally, concerning the number of values for* `"e"` *we have* $N_e \geq \lceil \frac{60}{2*29} \rceil$*, i.e. we need two values for the new property* `"e"` *in order to reach the 100 instances, so that the generation of values for* `"e"` *returns* $V_e = [1, 2]$*. We perform then a cartesian product between the sets of values associated to each key in order to obtain the 100 instances.*

We present now in more details the three algorithms invoked by the main object generation algorithm just illustrated.

Algorithm 2 - SatReq

Algorithm 2: $SatReq(ObjTA, N)$

Result: A map from strings to sets of JSON values | $GenFail$

```
1  objMap : Map[Str, Set[J]] = Map();  frac = N
2  for each l ∈ required do
3      CS = {|}
4      for each (p_i, S_i) ∈ patternProperties s.t. l ∈ L(p_i) do
5          CS = CS ∪ S_i
6      Ŝ = merge(CS); V_l = Gen(Ŝ, frac)
7      if V_l ≠ GenFail then
8          objMap[l] = V_l; frac = ⌈frac/ |V_l|⌉
9      else
10         return GenFail
11 return objMap
```

Algorithm 2 aims at satisfying the `required` constraint by generating a set of key-value pairs. For each required property l, we consider all sub-schemas S_i associated to patterns p_i whose language contains l and which, therefore, need to be merged into $\widehat{S}$ using the function *merge* defined in sub-section 5.2.2.

Going back to Example 15 we have that values generated for the key `"a"` must satisfy the schemas corresponding to both `"a"` and to `"a.*"`, whereas values generated for the `"b"` key must only satisfy the schema corresponding to `"b"`. As illustrated before, in generating those field values, we generate multiple values for each key so as to increase towards N the number of objects that we can form by combining `"a"` and `"b"` values. The pairs `"a"` : V_a and `"b"` : V_b, where V_a (resp. V_b) collects the set of generated values for `"a"` (resp. `"b"`), are then returned in the map $objMap$. Note that if the generation returns $GenFail$ for a required key, the generation for the whole schema aborts and returns $GenFail$.

Algorithm 3 - SatMinProp

The goal of Algorithm 3 is to fulfill the `minProperties` constraint, if not already satisfied by the previous generation step. It resorts at generating the *missing* keys from a set of candidate patterns obtained by considering the specification of `propertyNames`: S_{str}, if any, using $Comb(S_{str}, p_i)$ which combines the schema S_{str} with a pattern p_i expressed in `patternProperties`:$\{p_1 : S_1, ..., p_n : S_n\}$, resulting into a pattern that accepts strings matching both S_{str} and p_i. So we build the set P of patterns in `patternProperties` having non empty intersection with S_{str} (line 5) and then try to generate attributes using these patterns (lines 9-18). For any $p \in P$ we first try to generate a label l from p by means of $GenKey(p,\ failP)$, which returns a string value that matches p and does match any pattern in $failP$. This last one includes patterns p_i for which we cannot generate an instance for S_i. We initially set $failP$ to the empty set (line 4). Then, with $l = GenKey(p,\ failP)$ we recover all $p_i : S_i$ such that $l \in L(p_i)$, by including of course also $p : S$ in `patternProperties` (lines 4-6), and we add S_i to the sets of schemas CS, whose schemas are then merged in order to obtain $\widehat{S}$. We generate then instances V_l for $\widehat{S}$ and by definition of the merge operation this implies that the instances in V_l satisfy all the S_i in CS (lines 14-20). So we update $objMap[l]$ and the number $frac$ of values to generate for the next key in order to reach N instances, and the remaining number m of key-value pairs to generate (lines 14-20). In case we are not able to generate any instances for $\widehat{S}$, and CS only contains the schema S (related to the pattern p of the current iteration of the loop starting at line 6, we add p to $failP$ (lines 20-21) so that we do not generate keys and related values for this pattern in subsequent iterations. After all patterns have been used to generate the missing m properties, if $m > 0$ then this means that we could not generate more properties so we return $fail$, otherwise, the current mapping $objMap$ including the generated instances for a number of properties meeting `minimum` is returned.

Algorithm 3: $SatMinProp(ObjTA, objMap, N)$

Result: A map from strings to sets of JSON values | $GenFail$

$P_{req} = \prod_{(l_i,V_i)\in objMap} |V_i|$; $m = min- |required|$; $frac = \lceil N/P_{req} \rceil$
if $m > 0$ **then**
  $failP = \emptyset$
  $P = \{p \mid (p, S) \in \texttt{patternProperties} \wedge \mathcal{L}(Comb(S_{str}, p)) \neq \emptyset\}$
  for *each* $p \in P$ **do**
    for $i = 1$ *to* m **do**
      if $p \notin failP$ **then**
        $l = GenKey(p, failP)$
        if $l == null$ **then**
          break
        else
          $CS = \{|\}$
          for *each* $(p_i, S_i) \in$ `patternProperties` *s.t.* $l \in L(p_i)$ **do**
            $CS = CS \cup S_i$
          $\widehat{S} = merge(CS), V_l = Gen(\widehat{S}, frac)$
          if $V_l \neq GenFail$ **then**
            $objMap[l] = V_l$; $frac = \lceil frac/\ |V_l| \rceil$; $m = m - 1$;
          else
            if $|CS| == 1$ **then**
              $failP = failP \cup p$
      else
        break
    if $m == 0$ **then**
      break
if $m > 0$ **then**
  return $fail$
return $objMap$

Algorithm 4 - MoreInstances

Algorithm 4: $MoreInstances(ObjTA, objMap, N)$

Result: $M \leq N$ instances

```
1  Res = ∅
2  failP = ∅
3  P = {p | (p, S) ∈ patternProperties ∧ ℒ(Comb(S_str, p)) ≠ ∅}
4  for each p ∈ P do
5      while true do
6          if p ∈ failP then
7              break
8          else
9              l = GenKey(p, failP)
10             if l == null ∨ |Res| == N then
11                 break
12             else
13                 CS = {|}
14                 for each (p_i, S_i) ∈ patternProperties s.t. l ∈ L(p_i) do
15                     CS = CS ∪ S_i
16                 (n, subs) = nbValuesToGen(ObjTA, objMap, N)
17                 Ŝ = merge(CS), V_l = GenS(Ŝ, n)
18                 if V_l ≠ GenFail then
19                     objMap[l] = V_l;
20                     Res = Res ∪ getNewInstances(l, objMap, subs, N)
21                 else
22                     if |CS| == 1 then
23                         failP = failP ∪ p
24     if |Res| == N then
25         break
26 return Res
```

The number of instances that can be built from the map $objMap$ once the previous phase has finished is $M = \prod_{(l_i,V_i)\in objMap} |V_i|$. These instances are constructed by taking the cartesian product of the sets V_i of values from $objMap$. The final phase of the generation, defined by Algorithm 4, is invoked when N has not been reached yet (i.e. $M < N$), and is meant to generate new instances, when possible.

Algorithm 4 is somewhat similar to Algorithm 3. The main difference is that Algorithm 4 terminates when the desired number of instances is reached or, otherwise, there are no more patterns to exploit for generating the new properties.

Like Algorithm 3, we use P and $failP$ in order to determine the pattens to use to generate new keys l. The main differences w.r.t. Algorithm 3 are the following ones. For each new generated key l we stop considering the pattern p_i from which l has been generated in case $l = null$ or in case we have a sufficient number of instances (lines 13-14). Otherwise we invoke $nbValuesToGen$ which returns the number n of values we need to generate for this key l so as to reach the desired number of instances N, and in addition it returns the set of subsets of non-required generated keys with which new values for l will be combined to in order to form new object instances (as illustrated in Example 15).

To this end $nbValuesToGen$ considers, from its input parameters, the current $M = N - \prod_{(l_i,V_i)\in objMap} |V_i|$, req and $nonReq$ as the sets of required and non-required properties appearing in $objMap$ respectively, min and max as the values of the constraints `minProperties` and `maxProperties` in the object schema $ObjTA$, and returns $subs$ as the set of subsets of $nonReq$ keys s.t. $\forall s \in subs$: $|s| \in [\, min - |req| - 1, max - |req| - 1\,]$. In other words, $subs$ is the set of all the possible combinations of non-required properties that will be associated to the new key l and to the required keys in order to generate new instances while respecting `minProperties`, `maxProperties` and `required` constraints. Given a new key l, the generation of N_l values for l ensures the generation of $P_{req} * N_l * \sum_{i=1}^{|subs|} P_i$ new instances, where: $P_{req} = \prod_{i=1}^{|req|} |V_i|$, where V_i is the set of values associated to the ith required key in $objMap$, and $\forall i \in [1, |subs|]$ we have $P_i = \prod_{j=1}^{|subs_i|} |V_j|$, where V_j is the set of values associated to the jth non-required key in the ith subset of $subs$ in $objMap$. To reach the N instances, we need to generate M new instances, hence, the number of values N_l to generate for the new key l should satisfy the following inequality: $P_{req} * N_l * \sum_{i=1}^{|subs|} P_i \geq M$, thus: $N_l \geq \lceil \frac{M}{P_{req} * \sum_{i=1}^{|subs|} P_i} \rceil$. If the generation produces valid JSON values for a given key l, we use $getNewInstances$ to combine the pair $l : V_l$ with the pairs that were stored in $objMap$ in order to build the new instances.

5.3.5 Array Types

The general form of array schemas that we consider is defined below:

$ArrTA$ ::=
{ `type`: `array`, `minItems`:$minIt$, `maxItems`:$maxIt$,
`prefixItems`:$[S_1, ..., S_n]$, `items`:S,
`contains`:S_c, `minContains`:$minC$, `maxContains`:$maxC$ }

Dealing with arrays in the general case is slightly more involved than dealing with objects due to the upper bound assertion `maxContains` and to the uniqueness constraint `uniqueItems`. The upper bound assertion requires the use of negation and poses a serious limitation. Luckily enough, in our collections of schemas representing typical real-world schemas, no schema uses this assertion, so we ignore `maxContains` in the current version of the work and postpone it to future work. The situation is different for the uniqueness constraint which is used in practice and which we can deal with by posing a restriction, that of considering the single instance generation, where the goal is to generate one instance

only, i.e $N = 1$. We will present multiple instances generation in the absence of `uniqueItems` then we show how we deal with `uniqueItems` in a restricted case where we only generate a single instance.

Remark 4 *While in the generation grammar of Figure 5.3, an array schema may contain a conjunction of* `contains` *assertions as consequence of applying merge, we decided to keep our presentation simple by focusing on the case of a single* `contains` *assertion, as our solution naturally generalizes to the broader case involving multiple such assertions. Additionally, in practice, we have never encountered such a scenario.*

Multiple array instance generation

The generation of N array instances is captured by Algorithm 5 which proceeds by calling $SatMinItMinC$ (cf. Algorithm 6) to generate $M \leq N$ array instances of homogenous size s satisfying both `minItems` and `minContains`, then, in case $M < N$, it attempts to generate the missing $N - M$ instances by calling $MoreInstances$ (cf. Algorithm 8) to combine the already generated M instances with extra values, if the array specification allows so, that is, if $s <$ `maxItems`; otherwise, it returns the instances already generated by $SatMinItMinC$.

Algorithm 5: main array generation algorithm

Data: An array assertion $ArrTA$, an integer N
Result: $M \leq N$ instances | $GenFail$

```
1  Res = {|}
2  arrMap = SatMinItMinC(arrTA, N)
3  if arrMap ≠ GenFail then
4  |   Res = arrProduct(arrMap);
5  |   M =| Res |; s = Size(Res[0]);
6  |   if M < N and s < maxItems then
7  |   |   return Res ∪ MoreInstances(ArrTA, Res, N−M, s)
8  |   else
9  |   |   return Res
10 return GenFail
```

Algorithm 6 - SatMinItMinC The general idea of generating a set of N arrays from an array specification is based on generating a sequence of k fractions of N using the sub-schemas of the specification then to cross-product these k fractions in order to build the N arrays. This generation relies on preparing a candidate array CA of sub-schemas by considering the schemas of `prefixItems` up to *minit* and, in case `prefixItems` has less schemas than `minItems`, CA is extended with as many copies of `items` as needed to meet the `minItems` constraint (lines 4-5). This preparation also serves for preparing a potentially

empty array called *tail* obtained by considering the `prefixItems` schemas not part of CA and which may be used for satisfying the `minContains` constraint (line 7).

Generation proceeds iteratively by consuming the schemas of CA in a sequential order while passing, at each iteration, the fraction $frac$ of values that remain to be generated. During this process, the `contains` schema is also considered towards exhausting `minContains`. In case combining this schema with the candidate schema from CA yields a valid instance, the counter j is decremented when calling $GenWithContIfPoss$ (line 9). All generated values are collected in an indexed map M and the fraction value is updated by considering the cardinality product of the already generated values, this is captured by $DomProd$ used in line 13 and defined as follows

$$DomProd(M) = \prod_{V \in values(M)} |V|$$

In case `minContains` is not completely satisfied, the generation continues by considering, this time, the *tail* schemas, if any (line 17) until they are completely examined, then resorting to the `items` schema if its combination with `minContains` schema yields a valid value (line 19). In either cases, when generation succeeds, the generated values are appended to the map, otherwise, the whole generation fails due to failing to meet `minContains`.

Example 16 *To illustrate $SatMinItMinC$, consider the following array schema and let S_1 (resp. S_2) denote the first (resp. last sub-schema) of* `prefixItems`*, and let S_i (resp. S_c) denote the sub-schema of* `items` *(resp.* `contains`*).*

```
{ "type": "array",
  "minItems": 3,
  "prefixItems": [
    { "type": "integer", "minimum": 1, "maximum": 2 },
    { "type" : "integer", "minimum": 4, "maximum": 5 }
  ],
  "items": { "type": "integer", "minimum": 2, "maximum": 10 },
  "contains": { "type": "integer", "multipleOf": 3 },
  "minContains": 2
}
```

The generation of 100 instances satisfying this schema is achieved as follows: First, CA is built as described before and yields $[S_1, S_2, S_i]$. Then generation proceeds by attempting to generate 100 *distinct values from S_1 while considering S_c. However, since S_1 and S_c are incompatible, generation falls back to considering only S_1 whose maximum domain is $V_0 = \{\!|1, 2|\!\}$. Now, we consider S_2 alone (after noticing that it is also incompatible with S_c) and attempt to generate $\lceil \frac{100}{|V_0|} \rceil = \lceil \frac{100}{2} \rceil = 50$ values but only succeed in producing $V_1 = \{\!|4, 5|\!\}$ which contains two values. The last schema in CA is S_i which can be combined with S_c and is requested to generate $\lceil \frac{100}{|V_0|*|V_1|} \rceil = \lceil \frac{100}{4} \rceil = 25$ values while it yields $V_2 = \{\!|3, 6, 9|\!\}$. In order to fulfill* `minContains`*, we still need to generated the same set of elements $V_3 = \{\!|3, 6, 9|\!\}$ one more time.*

Algorithm 6: $SatMinItMinC(ArrTA, N)$

Result: a map containing pairs (index, set of values) | $GenFail$

```
1  M : Map[Int, Set[J]] = Map(); frac = N; j = minC;
2  /* satisfy minItems, satisfy minContains if possible */
3  m = Min(minit, size(prefixItems)); CA = [S_1, .., S_m]; tail = [];
4  if size(CA) < minit then
5  |  CA.concat(repeat(S_it, minit − m));
6  else
7  |  tail = [S_{m+1}, ..., S_n]
8  for i in 1..size(CA) do
9  |  (V, j) = GenWithContIfPoss(CA[i], S_c, frac, j);
10 |  if V ≠ GenFail then
11 |  |  M[i] = V; frac = ⌈frac/DomProd(M)⌉
12 |  else
13 |  |  return GenFail
14 /* satisfy remaining minContains */
15 while j > 0 do
16 |  if size(tail) > 0 then
17 |  |  (V, j) = GenWithContIfPoss(tail.removeFirst(), S_c, frac, j);
18 |  else
19 |  |  V = Gen(merge({|S_it, S_c|}), frac); j−−;
20 |  if V ≠ GenFail and size(M) < maxItems then
21 |  |  M[i] = V; frac = ⌈frac/DomProd(M)⌉
22 |  else
23 |  |  return GenFail
24 return M
```

Algorithm 8 - MoreInstances The generation of missing array instances exploits the set of already generated arrays *Res* by extending them with new values generated by considering the array specification, to produce new instances of larger size, which are thus, distinct from those of *Res*. The main task of MoreInstances is to produce $\lceil N/\,|Res|\rceil$ distinct values so that, when they are used for extending the *Res* arrays, we are able to obtain N distinct instances. These new values are first obtained from schemas of `prefixItems` in case not all sub-schemas of `prefixItems` have been used by SatMinItMinC (lines 5-7) then by considering `items` in case all `prefixItems` sub-schemas have been used by SatMinItMinC or that the combination of the generated values does not produce N (lines 8-9). Lastly, the newly generated values are combined with the arrays from *Res* (lines 11 and 12), and in case the combinations exceed the requested value N, only N such combinations are returned (using an arbitrary *trunc* function).

Algorithm 7: $GenWithContIfPoss(S_h, S_c, N, j)$

Data: Head schema S_h, 'contains' schema S_c, # instances N, # contains j
Result: A pair (V, j), V is the generated value and updated j

```
if j > 0 then
    V = Gen(merge({|S_h, S_c|}), N);
    if V ≠ GenFail then
        return (V, j − 1)
return (Gen(S_h, N), j)
```

Example 16 (continued)

Now, we construct the array instances by combining elements from all V_i's whose total combination amounts to $\prod_{i=0}^{3} |V_i| = 36$; hence it remains to generate $100-36 = 64$ by using S_i which is requested to return $\lceil \frac{64}{\prod_{i=0}^{3} |V_i|} \rceil = 2$ values and it indeed produces $V_4 = \{|2, 3|\}$. These values are built and added to the previous ones, making the total number of instances generated $36 + \prod_{i=0}^{4} |V_i| = 36 + 72 = 108$ instances, and only 100 instances are returned.

Algorithm 8: $MoreInstances(ArrTA, Res, N, m)$

Result: $M \leq N$ instances

```
M : Map[Int, Set[J]] = Map(); CA = []; i = 0; frac = ⌈N/ |Res|⌉;
if size(prefixItems) > m then
    CA = [S_{m+1}, ..., S_n]
/*Generate new values*/
while i < size(CA) and frac > 0 do
    V = Gen(CA[i], frac); i++;
    if V ≠ GenFail then
        M[i] = V;
    frac = ⌈frac/DomProd(M)⌉;
if frac > 0 then
    V = Gen(S_it, frac);
    if V ≠ GenFail then
        M[i] = V;
/*combine new values with Res*/
New = arrProduct(M);
return trunc(N, concat(Res, New))
```

Single instance generation in the presence of `uniqueItems`

A naive solution for generating an array of distinct values involves iterating over the candidate array CA, generating a value for the first candidate, and continuing this process for each subsequent candidate. However, ensuring distinct values for each candidate quickly becomes problematic. For instance, when generating a value for the second candidate, it may result in the same value as that of the first candidate. In such cases, we would need to regenerate a new value, leading to two potential strategies for resolving the issue.

The first strategy is to repeatedly generate new values randomly until a distinct one is obtained. However, this approach may never terminate, particularly when dealing with schemas that define infinite domains or have overlapping constraints within CA. The second strategy involves introducing negation to discard previously generated values. This adds significant complexity, as it requires schema canonicalization to properly handle negations. In our case, negation elimination is only partially supported, which could lead to generation failures when schemas involve negations that are difficult or impossible to resolve.

To avoid these challenges, we propose a solution that does not adopt either of the previous strategies. Instead, it aims for efficiency by performing a single pass over the candidate array CA, while guaranteeing termination. Generating a single array instance that ensures uniqueness is a specific case of multiple array instance generation. While both processes involve satisfying `minItems` and `minContains` by invoking SatMinItMinC, they differ in the number of values generated at each iteration. In multiple array instances generation, we start with N and progressively decrease the number of values generated as the process continues. In contrast, when generating a single instance that satisfies `uniqueItems`, we begin with $N = 1$ for the first candidate of CA and incrementally increase the number of generated values for each subsequent candidate.

The purpose of incrementing N is to prevent the generation of the same value for different candidates in previous iterations. To ensure that each new candidate of CA can be assigned a distinct value, the approach generates $N = i + 1$ values for the candidate at index i of the array CA (where $i = 0$ for the first candidate). When the generation of the N distinct values is successful, it guarantees that there is always a distinct value available for each candidate, maintaining the uniqueness constraint. However, even when generating fewer than N distinct values in a particular iteration, the array generation can still succeed. This is because the overall process does not strictly rely on producing a fixed number of values at each step, but rather focuses on ensuring that the final solution satisfies the uniqueness constraint by considering all candidates collectively.

Graph-theoretic approach to uniqueness verification. To explain why generating fewer values can still result in an array that respects the uniqueness constraint, we adopt a graph-theoretic approach. Specifically, we construct a *bipartite graph* $G = (L, R, E)$, where the left-hand side L vertices represent the indices of the array, and the right-hand side R vertices represent the values generated for the candidates of CA. The edges E connect the indices to the generated values. To ensure uniqueness, we have two possible approaches for verification.

A first exponential solution with perfect matching. Ensuring uniqueness at each iteration is equivalent to guaranteeing the existence of an L-perfect matching in G, defined as a set of disjoint edges that covers every vertex in L. In other words, this ensures that each vertex in L is connected to a distinct vertex in R, preventing any two vertices in L from sharing the same value from R.

The key to this approach lies in Hall's Marriage Theorem [47], which provides a necessary and sufficient condition for the existence of an L-perfect matching. According to the theorem, there is an L-perfect matching in G if, for every subset X of L, the following condition holds: $|X| \leq N_G(X)$, where $|X|$ represents the number of vertices in the subset X, and $N_G(X)$ is the neighborhood of X in G (i.e., the set of vertices in R connected to at least one vertex in X). This condition is evaluated only when the desired number of distinct values N to generate is not met.

Specifically, we start with an empty graph G, where $L = R = E = \emptyset$. During the first iteration, we generate one value v_0 for the first index of the array. If the generation fails, the process stops and returns GenFail. Otherwise, we add the vertices to both L and R , resulting in $L = \{\!|0|\!\}$, $R = \{\!|v_0|\!\}$ and $E = \{\!|(0, v_0)|\!\}$. The process continues in this manner. At each iteration, we need to compute the pairs (X,$N_G(X)$) of subsets of L and their neighborhoods in G. As stated before, the verification of the Hall's Marriage condition only intervenes when the desired number of distinct values is not generated; nevertheless, the subsets are computed at each iteration (not all subsets are recomputed, only the new ones by accumulation). This reduces the complexity, which is initially $O(2^{|L|} \cdot |E|)$ at each iteration, but it is still exponential.

Maximum cardinality matching for improved efficiency. Given the impracticality of the previous solution when the size of the arrays to generate becomes large due to the exponentiality of the algorithm, we address this with a second approach, which is also graph-theoretic, using the same graph G and construction process. However, the verification of the uniqueness changes as follows: at each iteration when we generate fewer distinct values than desired, we find the maximum cardinality matching in G. This is a subset of the edges E such that vertices from both L and R are adjacent to at most one edge. After finding the maximum matching, we verify whether it contains all the vertices of L,which means all the indices of the array are covered. There exist many algorithms for finding the maximum cardinality matching; among them, the Hopcroft-Karp algorithm [50] is one of the most efficient, with a complexity of $O(\sqrt{|L| + |R|} \cdot |E|)$.

Once the complete graph G is built, we need to retrieve the L-perfect matching or the maximum matching in the case of the second approach.

Example 17 *To illustrate the process of building the bipartite graph, consider the following schema, which includes various types of assertions that can be expressed in arrays while requiring the values to be distinct. The verification will be conducted using the first approach, which focuses on ensuring uniqueness through L-perfect matching in the bipartite graph.*

```
{ "type": "array", "minItems": 2, "maxItems": 4, "minContains": 2,
  "prefixItems": [
    { "type": "integer", "maximum": 2 },
    { "enum" : [3,12,18] }
  ],
  "items": { "type": "integer", "multipleOf": 2, "maximum": 20 },
  "contains": { "minimum": 10, "multipleOf": 3 },
  "uniqueItems": true
}
```

To satisfy this schema, an array must have a minimum of two items, the first of which adheres to `prefixItems`*, and it must contain at least two items satisfying the* `contains` *schema. In our case, this requirement can only be fulfilled by generating an additional item using* `items`*, since the constraints of the first schema* `prefixItems` *contradict those of* `contains`*.*

Figure 5.6 depicts the iterations for building the graph $G = (L, R, E)$.
In the following we describe the iterations for building the graph G, where the neighborhood of a subset X of L in G is represented as $X \to Y$, with Y being a subset of R that includes the vertices connected to at least one vertex of X.

1. *Initially, only the first schema of* `prefixItems` *at index 0 is considered, resulting in the generation of $\{\!|2|\!\}$. G is updated to contain $L = \{\!|0|\!\}$, $R = \{\!|2|\!\}$, and $E = \{\!|(0, 2)|\!\}$. The neighborhood of the subsets of L at this iteration is*

$$N_G = \{\!|\{\!|0|\!\} \to \{\!|2|\!\}|\!\}$$

2. *Next, the second schema is considered together with* `contains`*, leading to the generation of $\{\!|12, 18|\!\}$. At this stage, $L = \{\!|0, 1|\!\}$, $R = \{\!|2, 12, 18|\!\}$, and $E = \{\!|(0, 2), (1, 12), (1, 18)|\!\}$. The neighborhood is updated to:*

$$N_G = \{\!|\{\!|0|\!\} \to \{\!|2|\!\}, \{\!|1|\!\} \to \{\!|12, 18|\!\}, \{\!|0, 1|\!\} \to \{\!|2, 12, 18|\!\}|\!\}$$

3. *Then,* `items` *is considered and combined with* `contains` *to fulfill* `minContains`*, resulting in the generation of the same set of values $\{\!|12, 18|\!\}$. The updated graph has $L = \{\!|0, 1, 2|\!\}$, $R = \{\!|2, 12, 18|\!\}$, and $E = \{\!|(0, 2), (1, 12), (1, 18), (2, 12), (2, 18)|\!\}$. The neighborhood from the previous iteration is then* ***augmented*** *with the following:*

$$\{\!|\{\!|2|\!\} \to \{\!|12, 18|\!\}, \{\!|0, 2|\!\} \to \{\!|2, 12, 18|\!\}, \{\!|1, 2|\!\} \to \{\!|12, 18|\!\}, \{\!|0, 1, 2|\!\} \to \{\!|2, 12, 18|\!\}|\!\}$$

   At this iteration, we observe that the generation process resulted in only two values: $\{\!|12, 18|\!\}$. This presents an interesting scenario, as the desired number of distinct values was $N = 3$, corresponding to $i + 1$, where $i = 2$.

To verify the uniqueness constraint, we examine the existence of an L-perfect matching in the bipartite graph G. The L-perfect matching requires that each vertex in L is connected to a distinct vertex in R. Given the current state of the graph, $L = \{\!|0, 1, 2|\!\}$ and $R = \{\!|2, 12, 18|\!\}$, we need to check if we can establish a matching that covers all vertices in L.

The verification of an L-perfect matching reveals that it is indeed possible to cover all vertices in L with the available values in R, as the condition $| X | \leq N_G(X)$, where X represents a subset of L, is satisfied. For instance, one valid matching could be $\{\!|(0, 2), (1, 12), (2, 18)|\!\}$. This outcome confirms that, despite generating only two values, the conditions for uniqueness can still be satisfied by appropriately matching the candidates with the generated values.

The graph admits two L-perfect matchings (which are also maximum matchings that contain all the vertices of L): $\{\!|(0, 2), (1, 12), (2, 18)|\!\}$ and $\{\!|(0, 2), (1, 18), (2, 12)|\!\}$ and allows for generating two arrays of unique items: $[2, 12, 18]$ *and* $[2, 18, 12]$.

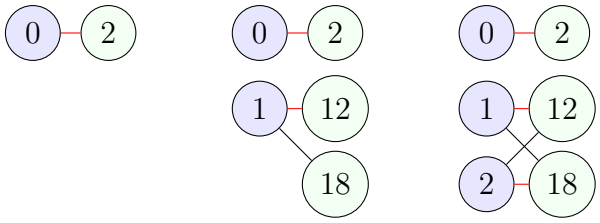


Figure 5.6: Graph G at iterations: 0, 1 and 2

Algorithm 9 outlines the approach for constructing the graph G, represented as the map M. As stated previously, the process of generating a single array instance in the presence of `uniqueItems` is similar to generating multiple arrays; it only differs in the number of values to generate for each candidate CA and in verifying the existence of an L-perfect matching (or a maximum matching that covers all the indices of the array), with the function *existsMatching* implemented to support both versions of the verification process. It is important to note that we have omitted the instructions for handling `contains` and `minContains` from the algorithm, as their satisfaction is managed similarly to the case of generating multiple instances. Once the graph is built without errors, the function *Matching* returns the matching that will be used to construct the array.

In conclusion, while the algorithm for generating a single instance in the presence of `uniqueItems` demonstrates notable efficiency, it does have certain limitations. A primary concern lies in the order in which the candidates from the array CA are processed. For example, if a candidate schema with an infinite domain is evaluated first, causing the algorithm to generate only one value for this schema, and the subsequent candidate schema has a finite domain limited to only accepting the value generated from the first, this can lead to generation failure.

Algorithm 9: UniqueItems generation

```
Data: An array of schema CA
Result: An instance J_a | GenFail
M : Map[int, Set[J]] = Map(); i = 0
while i <|CA| do
    V_i = GenS(CA[i], i + 1)
    if V_i == GenFail then
        return GenFail
    else
        if |V_i|< i + 1 then
            if ¬existsMatching(M) then
                return GenFail
        M[i] = V_i; i = i + 1
return Matching(M)
```

To address this issue, it is essential to adopt a more sophisticated approach that employs a heuristic to determine the order in which candidates are evaluated. By strategically reordering candidates, it becomes possible to mitigate the risk of generation failures and enhance the overall effectiveness of the solution. This adjustment not only improves the algorithm's robustness but also paves the way for generating valid instances more reliably in complex scenarios.

Furthermore, it is possible to generalize the problem of single instance generation in the presence of `uniqueItems` to the task of generating N different instances. The main idea here is to ensure the presence of N distinct L-perfect matchings (or N distinct maximum matchings) at each iteration. This generalization not only broadens the applicability of the algorithm but also provides a framework for generating distinct multiple valid instances while maintaining the uniqueness constraints required by the `uniqueItems`.

5.4 Conclusion

In this chapter, we introduced a novel optimistic approach for JSON Schema data generation, designed to address the limitations of existing techniques by being efficient while maintaining correctness and a high level of coverage. The two main contributions of this approach are: its ability to generate multiple distinct instances and its handling of arrays with the `uniqueItems` constraint.

The first major contribution, multiple instances generation, allows for the generation of distinct, valid instances for the same schema. This feature is especially useful in testing and validation scenarios, where having a variety of examples is required. By utilizing efficient generation algorithms tailored to each data type, our approach ensures that each instance conforms to the schema while being unique. This marks a significant improvement

over previous methods that were designed to generating a single instance and where the generation of multiple distinct instances is not guaranteed.

The second contribution, the ability to handle arrays with the `uniqueItems` constraint, ensures that generated arrays contain unique items, a feature often overlooked or not well handled by previous generation tools. While we provided an efficient solution for this problem, it does have some limitations, primarily related to the order in which schemas are processed. These limitations can be addressed with more sophisticated solutions that use heuristics to optimize the order of evaluation. Additionally, we discussed a potential approach to generalize this problem for generating multiple arrays while respecting the `uniqueItems` constraint, ensuring that the algorithm produces distinct arrays, each containing values that are unique.

The preprocessing phase of our approach, consisting of reference expansion and schema canonicalization, is crucial for simplifying the schema and preparing it for efficient generation. Reference expansion resolves internal references, making the schema self-contained, while canonicalization restructures it to facilitate generation. Designed to be minimal, this phase reflects the optimism in our approach. Unlike the complex preprocessing techniques employed in the pessimistic method outlined in Chapter 4, we focus on straightforward procedures that do not involve intricate operations. This phase, although minimal, also differentiates us from other optimistic solutions, which often neglect preprocessing and suffer from errors due to inadequate preparation.

Our approach is designed for efficiency and correctness, which makes it inherently incomplete. Its incompleteness arises from several factors: partial negation elimination, incomplete translation of patterns from the regular expression language used by JSON Schema to that used by the Brics library [60], and static unsatisfiability checking that leaves cases requiring dynamic verification undetected. Despite these limitations, the method aims for high coverage: satisfiable schemas generate valid instances, and the approach can flag unsatisfiable schemas early, reducing unnecessary computation. The trade-off in completeness is acceptable for many real-world applications where quick and correct results are more valuable than exhaustive instance coverage.

In conclusion, this chapter presented an efficient and sound solution for JSON Schema data generation, designed to address the limitations of existing techniques by balancing efficiency, correctness, and completeness. With support for generating multiple distinct instances and correctly handling the `uniqueItems` constraint in arrays (even while limiting to single-instance generation due to complexity), this approach achieves a balance between performance and accuracy. Its streamlined preprocessing and focus on soundness make it a practical tool for a wide range of tasks requiring JSON Schema data generation, particularly in scenarios where speed and correctness take precedence over completeness.

Chapter 6

Experimental Analysis

In this chapter, we present a comprehensive experimental analysis of our instance generation approach for JSON Schema. The experiments aim to achieve several objectives: first, to compare our method with existing data generation tools in terms of correctness and execution times; second, to assess the scalability of our optimistic approach and its ability to generate multiple valid instances; and third, to investigate our handling of uniqueItems.

6.1 Introduction

The goal of this chapter is to evaluate the performance and effectiveness of our instance generation approach for JSON Schema through a series of targeted experiments. The experiments are designed to address three key aspects of our method. First, we aim to compare our approach with existing data generation tools, focusing on both the correctness of the generated instances and the time required to generate them. This comparison will help us understand how our solution measures up against other state-of-the-art tools in terms of accuracy and efficiency.

Second, we assess the scalability and efficiency of our optimistic approach in generating multiple valid instances across real-world schemas. This involves evaluating not only the ability of our method to handle individual instance generation but also its effectiveness in producing sets of instances with varying cardinality. By examining how well our approach scales, we can determine if it maintains low execution times, ensuring that it remains efficient and practical for real-world applications that require large-scale instance generation.

Finally, we evaluate the method's ability to successfully handle the `uniqueItems` constraint, which is a key contribution of this work. This experiment assesses the robustness and correctness of our approach in managing this challenging schema feature.

To conduct these experiments, we used the schema collections previously employed in the pessimistic method introduced in Chapter 4 (cf. Table 4.2 for a detailed description of

the schema collections), ensuring consistency in our evaluations. Our optimistic approach tool was implemented in Scala, leveraging pattern matching to encode schema rewriting rules for preprocessing the input schema. The generation algorithms were implemented in a more imperative style to maximize efficiency. Additionally, we relied on the Brics library [60] for generating strings from patterns, following the same design principles as in our previous work on witness generation [11, 12]. The experiments were performed on a Precision 7550 laptop with a 12-core Intel i7 2.70GHz CPU and 32 GB of RAM, running Ubuntu 23.04.

6.2 Single Instance Generation Experiments

To evaluate the performance of our optimistic generation tool, we compared it with the tools JSON-Schema-faker (*JSF*) [1] and JSON-everything (*JE*) [36] presented in Section 2.3.2. Since these other tools are designed to generate only a single instance, we configured our optimistic tool to also produce just one instance by setting $N = 1$. Additionally, to ensure a fair comparison, we ignored the `uniqueItems` constraint, as it is not handled by the other tools.

It is important to note that the experiments involving the *witness generation* tool introduced in Chapter 4, the *DG* tool [28], and the *CC* tool [51] have already been discussed in Section 4.3, with the full results presented in Table 4.3. Therefore, we will not reference these tools again in this context.

6.2.1 Experimental Results

Table 6.1 presents the results of our evaluation of our approach's ability to generate valid instances. We executed our tool alongside the two other tools[1] for each schema collection and subsequently collected the generated instances, which were validated against the original schema (when satisfiable) using an external validator [3]. The correctness ratios for each combination of tool and schema collection are reported, detailing three distinct outcomes: *success*, which includes cases where a valid instance is produced as well as scenarios where the schema is unsatisfiable and the tool correctly indicates emptiness; *logical errors*, which occur when a non-valid instance is generated or an instance is generated while the schema is unsatisfiable; and *interruptions* (*Interrupt.*), which include any processing errors such as timeouts. Additionally, we report the ratio of schemas that our tool intentionally discards to prevent the generation of potentially invalid instances (*GenFail*). To ensure the timely completion of our experiments, we set a processing timeout of 5 minutes for each schema.

Correctness and Completeness. The experimental results indicate that the optimistic tool encounters interruption errors in 0.37% of the schemas (24 schemas), with 22 of these

[1] *JSF* was tested using its Command Line Interface, while *JE* was tested by directly using its source code.

Col.	Tool	Success	Interrupt. (+GenFail)	Logical Error	Time (ms) med./avg.
Git	Ours	98.17%	0.37% +0.81%	0.65%	1/54
	JSF	82.64%	5.80%	11.56%	117/323
	JE	58.68%	20.77%	20.55%	9/10
K8s	Ours	100%	0% +0%	0%	2/15
	JSF	89.84%	0.18%	9.98%	122/125
	JE	69.41%	8.43%	22.16%	7/8
Snw	Ours	99.04%	0.48% +0%	0.48%	2/223
	JSF	95.24%	2.62%	2.14%	119/549
	JE	72.38%	18.10%	9.52%	2/5
WP	Ours	100%	0% +0%	0%	5/65
	JSF	87.20%	0%	12.80%	121/134
	JE	29.60%	25.60%	44.80%	6/11
HW	Ours	18.72%	5.53% +71.07%	4.68%	5/95
	JSF	9.36%	17.45%	73.19%	221/3,806
	JE	3.83%	1.70%	94.47%	2/7
CC4	Ours	78.96%	0.45% +19.69%	0.90%	1/2
	JSF	27.20%	2.78%	70.02%	206/219
	JE	0.23%	3.91%	95.86%	1/508

Table 6.1: Correctness results and execution times

due to timeouts where the tool fails to return a result within the 5-minute limit. The remaining 2 schemas experience errors related to unhandled patterns. Additionally, there are 2 timeouts in the Snowplow collection, which correspond to the same schemas that resulted in timeouts for the *witness generation* tool. As previously mentioned, this issue arises from high values of the `maxLength` constraint. In the Handwritten and Containment-draft4 collections, interruptions stem from patterns. However, the errors are not solely due to the patterns being unsupported by the Brics library [60] but also arise from the way patterns are transformed during the negation elimination process in string schemas, as introduced in 5.2.2. Specifically, this process involves pushing the negation inward to complement the pattern, and the implementation struggles when dealing with conjunctions of negated patterns. This leads to interruption errors, revealing a bug in the system that needs to be addressed.

Regarding logical errors, these primarily arise from unhandled patterns in the GitHub, Snowplow, and Containment collections. In the Handwritten collection, the errors result from very intricate schemas where logical operators are nested and interact in complex ways, indicating the presence of bugs related to the processing of these operators.

Generation failures are almost nonexistent in the real-world schemas, with the tool returning *GenFail* for only 52 schemas in the GitHub collection. An examination of

these schemas revealed the presence of negations applied to object and array schemas. In contrast, we observe a high incidence of generation failures in the Handwritten and Containment-draft4 collections, as these synthetic schemas make excessive use of object and array negations.

In terms of success rates, the optimistic tool shows a high success rate, outperforming both other tools and the *DG* tool. It also demonstrates comparable success rates to the *witness generation* tool, except for the synthetic schema collections.

Regarding the other tools, in the GitHub collection, the *JSF* tool shows a notable level of success, but it still experiences numerous interruptions and logical errors, particularly when handling synthetic schemas. These issues indicate its difficulties in consistently generating valid instances, which makes it not effective in practice. The *JE* tool, in particular, struggles even more, exhibiting a high rate of interruptions alongside a significant number of logical errors. This clear disparity highlights the superior performance and reliability of our approach in generating valid instances across the different schema collections.

Execution Times. The execution times reported in Table 6.1 reveal strong performance of the optimistic tool across the various schema collections. The tool demonstrates consistently low median and average execution times, with median times as low as 1-5 milliseconds and average times always under 100 milliseconds, except for the Snowplow collection, where some schemas exhibit higher execution times. For example, in the GitHub collection, 66.31% of the successfully processed schemas (4,246 out of 6,403) were completed in 1 millisecond or less. As noted previously in Chapter 4, the longer execution times in the Snowplow collection can be attributed to the presence of complex patterns, where string generation significantly contributes to increased processing times.

In comparison to the *witness generation* tool, the optimistic tool is less affected by the preprocessing phase, requiring minimal preprocessing. Table 6.2 further illustrates the median and average times for both the canonicalization and generation phases. While the execution times for canonicalization are generally higher than those for generation in most collections, with both the median and average times being greater for canonicalization, it is noteworthy that when considered individually, generation times are typically higher. This trend is illustrated in Figure 6.1, a scatter plot of generation versus canonicalization times, where this observation is more clearly visualized for the GitHub and Snowplow collections.

Canonicalization is influenced by the elimination of conjunctions and the partial removal of negation, particularly when these operations are performed within nested structures (i.e., inside object and array constraints), which can lead to increased execution times. Conversely, generation is impacted by string generation, relying on the Brics library for generating strings and object properties, whereas canonicalization does not utilize Brics at all. It is also important to note that generation depends on canonicalization; for instance, when an object property matches multiple patterns, generation requires the merging of the schemas of the matching patterns, which can sometimes result in longer execution times.

These findings underscore the effectiveness of our optimistic approach across the majority of schema collections, demonstrating its ability to deliver efficient processing while maintaining low execution times.

Col.	Canoncicalization		Generation	
	Median	**Average**	**Median**	**Average**
Git	0	8.86	0	41.84
K8s	1	10.89	1	1.16
Snw	0	0.61	1	222.41
WP	1	37.94	1	4.80
HW	4	46.31	0	8.68
CC4	1	1.69	0	0.2

Table 6.2: Median and average times for the canonicalization and the generation phases

Regarding the other tools, the *JSF* tool consistently shows higher execution times across all collections, particularly in the HW collection, where the average time exceeds 3800 milliseconds. Regarding the *JE* tool, although it shows very low execution times, a fair comparison must take into account the success rate of each tool. Given that *JE* encounters a very high number of interruptions and logical errors, it proves unreliable, and its execution times are of limited relevance.

Moreover, as highlighted in earlier experiments, the relationship between schema size and execution time remains a significant factor. Figure 6.2, in which the plots of execution time against schema size on a log-log scale, continues to show a linear correlation for many schemas while identifying a few outliers across the different schema collections.

The results are consistent with the previous experimental analysis discussed in Chapter 4, where outliers in execution times were attributed to complex schema features such as intricate patterns, nested logical operators, and the `maxLength` keyword with high values.

6.2.2 Preliminary Experiments with *HJS*

While the primary focus of the experimental analysis centered on other tools and methods, Hypothesis-jsonschema (*HJS*) [48] had not been initially included in the testing procedures. To address this, a set of preliminary experiments was later conducted specifically for *HJS*. This subsection presents those findings, highlighting its performance and behavior under similar conditions to the previously analyzed tools.

Table 6.3 highlights the results of the experiments of *HJS* on the schema collections listed in Table 4.2. The execution times of the Github schema collection were not recorded, hence they are not presented in the table.

The comparison between the results of *HJS* and our technique (cf. Table 6.1) reveals that our tool achieves higher success rates across all schema collections, with the sole exception of the Handwritten collection. This difference arises because, in our approach, we do not eliminate negation when encountering object and array schemas, whereas *HJS* addresses some of those cases.

The results indicate that *HJS* suffers from a high number of interruption errors, primarily due to its inability to handle recursive references effectively. In terms of logical errors, most arise from the generation of invalid string values, particularly in the Github

and Snowplow collections. In the Snowplow collection, 16 out of 18 logical errors are due to patterns, while 2 are due to not satisfying the constraint `multipleOf`. In the Github collection, 337 out of 379 errors arise from not satisfying `pattern`. For the Handwritten collection, *HJS* struggles with logical errors mainly due to its failure to generate values that are valid with respect to schemas involving negation and conjunction.

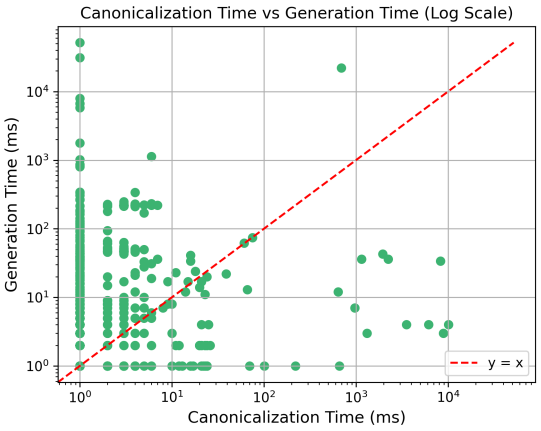
Regarding execution times, measured in **seconds**, *HJS* exhibits significantly long execution times. This was expected due to its underlying structure, as it relies on the property-based testing framework *Hypothesis* [59]. Notably, it is the tool with the highest execution times across all the different tools tested in our experiments. This increased duration may impact its practical usability, especially in scenarios requiring rapid data generation.

Col.	**Success**	**Interrupt.**	**Logical Error**	**Time (seconds) med. / avg.**
Git	86.56%	7.34%	6.10%	-
K8s	97.89%	2.11%	0%	5.47 / 13.71
Snw	95.48%	0.24%	4.28%	1.99 / 4.48
WP	80.8%	19.2%	0%	2.77 / 4.57
HW	36.60%	58.72%	4.68%	14.88 / 29.65
CC4	77.99%	21.64%	0.37%	0.02 / 5.69

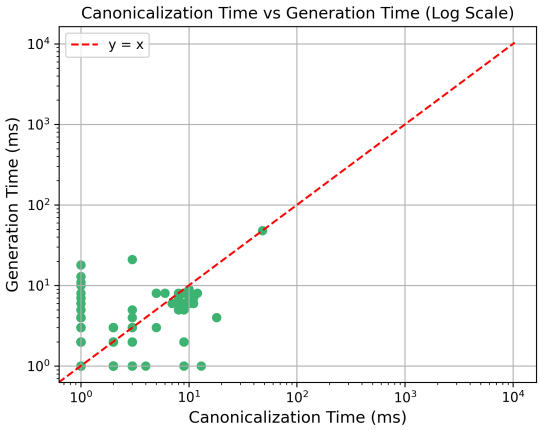
Table 6.3: Correctness results and execution times of *HJS*

6.2.3 Conclusive Remarks

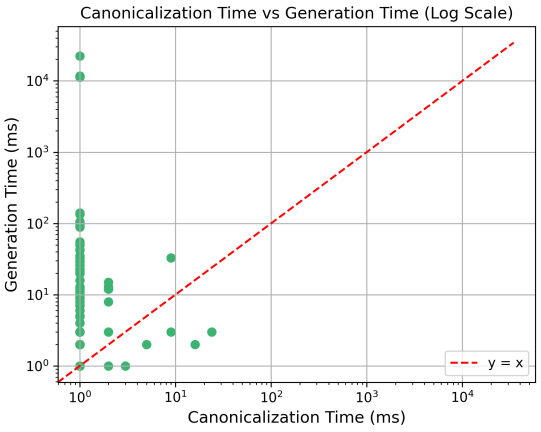
The experimental analysis of our method for single instance generation demonstrates that it outperforms other tools. Although the *witness generation* tool (Chapter 4) demonstrates higher accuracy, our approach yields comparable results for real-world schemas. A key advantage of our tool is its very low execution times, making it particularly suitable for applications requiring fast data processing. While we faced some challenges with specific schema collections, the overall performance of our tool underscores its reliability and effectiveness in practical scenarios.



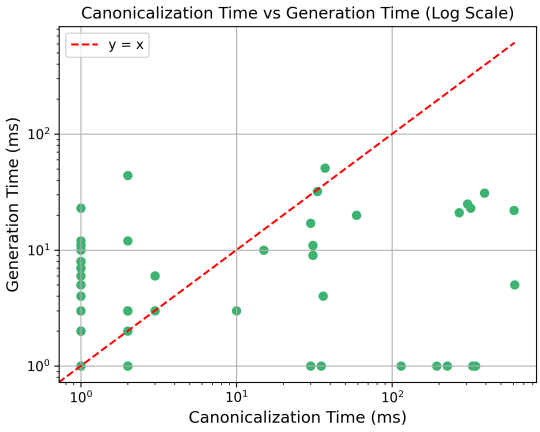
(a) Github collection



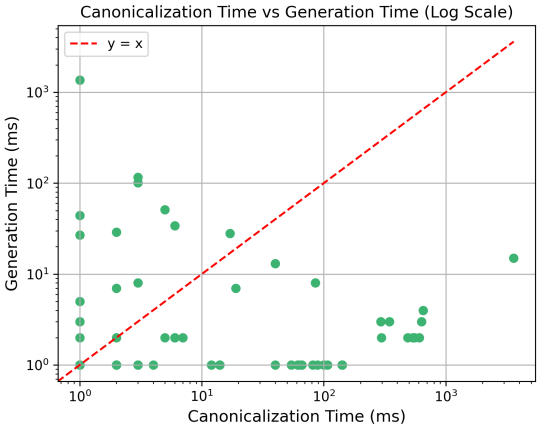
(b) Kubernetes collection



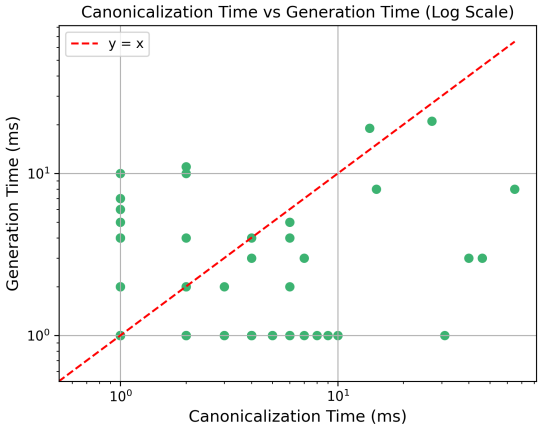
(c) Snowplow collection



(d) WashingtonPost collection

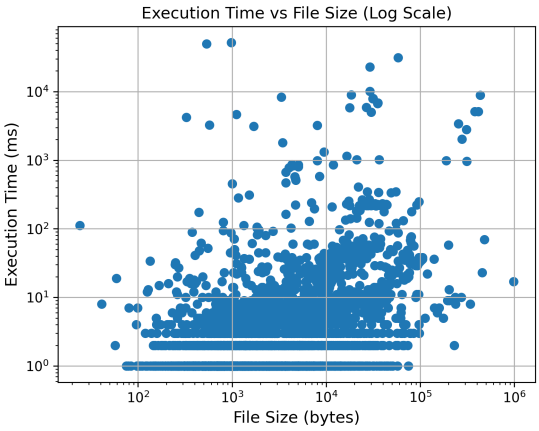


(e) Handwritten collection

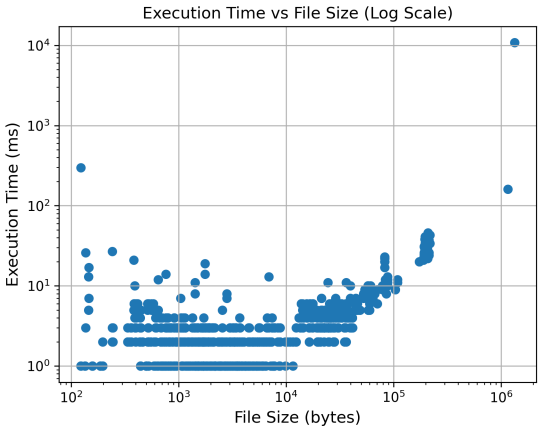


(f) Containment-draft4 collection

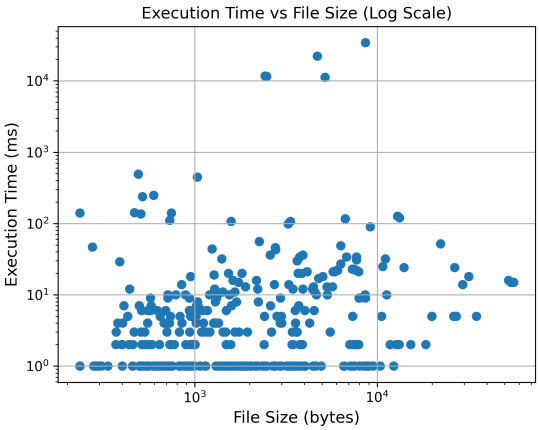
Figure 6.1: Canoncilization time vs generation time on a Log-Log Scale



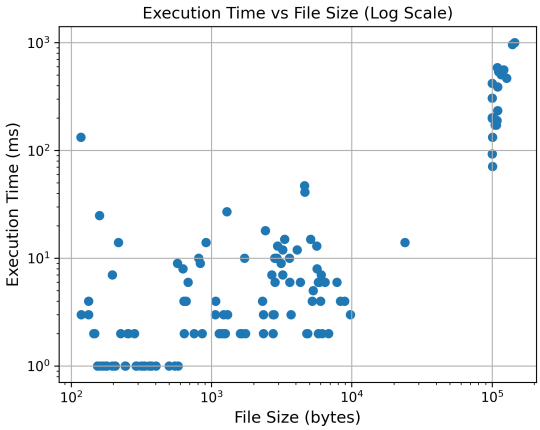
(a) Github collection



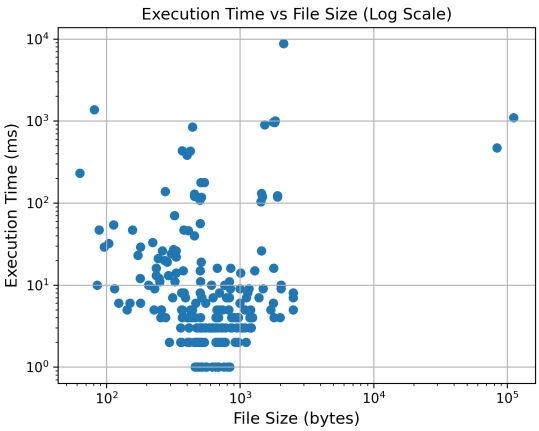
(b) Kubernetes collection



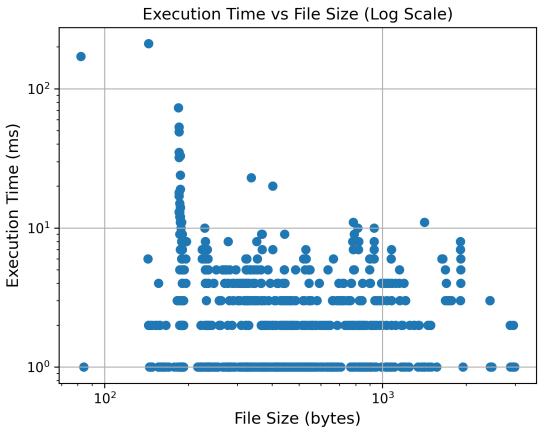
(c) Snowplow collection



(d) WashingtonPost collection



(e) Handwritten collection



(f) Containment-draft4 collection

Figure 6.2: Single instance generation: file size vs. runtime on a Log-Log Scale

6.3 Multiple Distinct Instances Generation Experiments

To evaluate the scalability of our instance generation approach, we conducted experiments focused on generating multiple distinct instances. This aspect is crucial for assessing how well our method can handle generating a large number of instances and ensuring efficiency in practical applications. For this purpose, we utilized the GitHub schemas that were successfully processed during our single instance generation experiments, creating a new dataset specifically for this evaluation.

Given a well-processed Github schema S, we constructed an array schema in the following format: `{"type":"array","items":S',"minItems":N,"uniqueItems":true}`, where `S'` is the resulting schema after performing the reference expansion process introduced in Section 5.2.1 on S. We varied N to examine the performance across different sizes, including values such as 5, 10, 20, and larger values of N up to 1000. We also included the `uniqueItems` constraint set to *true* to ensure that every instance amongst the N instances to be generated is unique. This approach thoroughly tests our method's capability to generate multiple valid instances efficiently and correctly.

For each value of N, we excluded schemas that inherently could not generate the specified number of instances (i.e., schemas that are unsatisfiable by design). This cleaning phase was carried out by manually inspecting the schemas. As in our previous experiments on single instance generation, each generation run here was conducted with a timeout of 5 minutes per schema.

We measured the various rates of success, interruptions, generation failures, and logical errors as in the previous experiments. Additionally, we included the average size of the instances produced.

The results of these experiments are reported in Table 6.4, providing a comprehensive overview of the performance metrics.

6.3.1 Correctness and Completeness

After analyzing the different rates measured, we can make the following observations: (i) the success ratio remains relatively stable, with a minor decline for higher values of N, accompanied by a slight increase in interruption and logical errors, and (ii) the generation failure ratio remains consistent, around 1%. As in our previous single instance generation experiments, we identified the same primary causes for these issues: interruptions were largely due to timeout errors and pattern-related issues, while logical errors resulted from generating invalid strings from patterns containing special characters not natively supported by the Brics library.

Upon further investigation, we observed that interruptions for lower values of N were mostly timeout-related, while pattern-related errors became more prominent as N increased. For instance, at $N = 5$, we encountered 12 timeouts, 6 interruptions caused by patterns, and 2 interruptions due to memory problems linked to two large schemas.

N	#Schemas	Success	Interrupt.	GenFail	Logical Error	Time (ms) Avg / Med	Avg Size (KB)
5	6,228	95.59%	0.35%	0.93%	3.13%	136 / 2	1
10	6,217	95.37%	0.47%	0.98%	3.18%	125 / 2	1.35
20	6,209	95.18%	0.47%	0.98%	3.37%	147 / 3	2.74
30	6,208	95.12%	0.55%	0.96%	3.37%	89 / 2	4.15
40	6,208	94.36%	1.32%	0.97%	3.35%	98 / 3	5.54
50	6,208	94.35%	1.32%	0.98%	3.35%	176 / 6	6.94
60	6,208	94.32%	1.32%	0.98%	3.38%	205 / 7	8.34
70	6,207	94.25%	1.32%	0.98%	3.45%	155 / 6	9.76
80	6,207	94.19%	1.37%	0.98%	3.46%	173 / 8	11.17
90	6,205	94.18%	1.37%	0.98%	3.47%	197 / 9	12.58
100	6,205	94.13%	1.42%	0.98%	3.47%	222 / 11	14
150	6,203	94.10%	1.50%	0.98%	3.42%	144 / 23	21.12
250	6,200	94.08%	1.55%	0.95%	3.42%	181 / 60	35.44
500	6,196	94.08%	1.57%	0.95%	3.40%	373 / 237	71.72
1000	6,194	94.04%	1.60%	0.95%	3.41%	1,174 / 974	145.98

Table 6.4: Multiple instances generation results

These schemas were the only ones across all N values to experience heap errors, and no additional memory issues were observed. At $N = 100$, we recorded 20 timeouts and 63 interruptions related to patterns, and for $N = 1000$, 27 timeouts and 67 pattern-related interruptions occurred.

As with the single instance experiments, the generation failures were primarily tied to schemas involving negations, in the presence of `not` or `oneOf`. These negations affected required branches of the schema, making them difficult to handle with our generation approach. Additionally, some failures were linked to limitations in the Brics library, which, for certain patterns, struggled to generate the required number of distinct strings, despite the input pattern meeting the necessary cardinality constraints.

6.3.2 Execution Times

When examining the execution times for different values of N, several notable trends emerge. For smaller values of N (from $N = 5$ to $N = 150$), the median times increase in a relatively linear fashion, indicating that most schemas are processed efficiently. In contrast, the average times are significantly higher, suggesting the presence of a few slow outliers. As N increases, both the average and median times steadily rise, with a noticeable increase in median times starting at $N = 250$ and a substantial jump at $N = 500$. This increase can be attributed to two factors: first, generating more instances inherently requires longer execution times; second, generating those instances may necessitate traversing additional branches of the schemas compared to those traversed for the previous value of N, which can also extend processing times. The decrease observed in average times at $N = 30$ and $N = 40$ can be attributed to a rise in interruption errors.

Furthermore, as N increases, both the average and median times continue to grow, indicating a trend of longer execution times for generating instances. Even at $N = 1000$, the average and median times remain distinct, though the gap between them narrows, suggesting that while the impact of outliers lessens, they still exist. Overall, the data shows that although a few slow schemas inflate the average times for smaller values of N, processing times for most schemas increase as N rises, reflecting a general slowdown in instance generation for larger values of N.

Despite this growth in execution times, generation using this optimistic approach remains highly efficient, capable of generating 1000 instances in approximately 1 second, making it suitable for real-world applications.

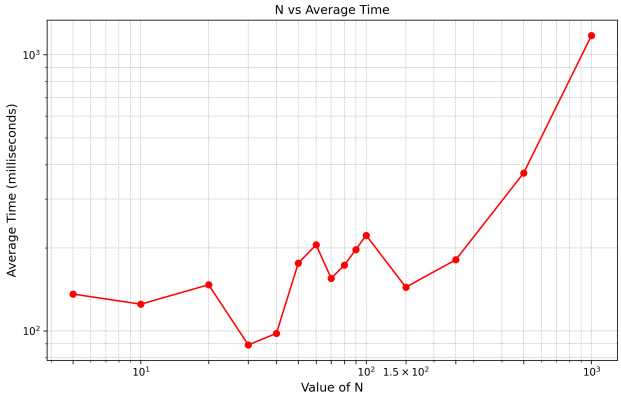
Figures 6.3a and 6.3b, which illustrate the evolution of average and median execution times as N varies, effectively help visualize and confirm these observations. The trends depicted in these figures reinforce our analysis, highlighting the differences in processing times for various schemas as the number of instances increases. By clearly displaying the changes in both average and median times, these figures provide a comprehensive understanding of the impact of N on execution efficiency.

6.3.3 Conclusive Remarks

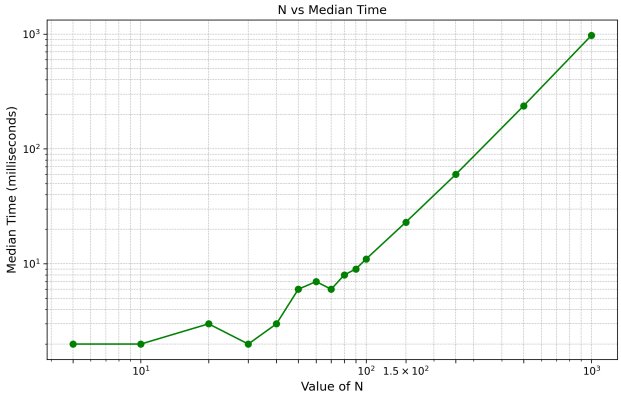
In conclusion, our analysis has highlighted the overall efficiency and effectiveness of the tool for generating multiple instances. The findings confirm that, despite some fluctuations in execution times and the presence of a few outlier schemas, the tool remains capable of generating 1000 instances in approximately 1 second. This efficiency is particularly noteworthy, as it suggests the tool's applicability in real-world scenarios requiring rapid data generation.

Additionally, Figure 6.3a and Figure 6.3b provide valuable insights into the trends of average and median execution times as N varies, confirming the patterns we observed. The linear growth illustrated in Figure 6.3c, depicting the average size of the generated instances, aligns with our expectations, given the deterministic nature of our optimistic approach. However, it's important to note that the growth can sometimes exceed linear, depending on the complexity of the schema. This occurs when, at certain points, we switch branches within the schema to generate the required number of instances. This behavior highlights the adaptability of the approach, although it can lead to increased size variability in some cases. This flexibility remains a strength of the tool, contributing to its reliability and effectiveness.

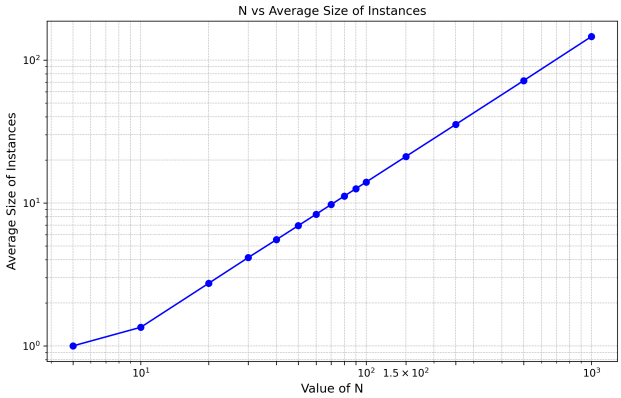
Overall, these insights reinforce the potential of our instance generation tool for practical applications, setting a solid foundation for future developments and improvements aimed at enhancing both accuracy and efficiency.



(a) N vs Average time



(b) N vs Median Time



(c) N vs Average size of the generated instances

Figure 6.3: Overview of Execution Metrics with Varying N

6.4 UniqueItems Experiments

To evaluate our approach's ability to handle the `uniqueItems` keyword and test the capabilities of the algorithm designed for this purpose, we constructed a specialized collection of synthetic schemas where `uniqueItems` is always set to *true*. The previous schema collections lacked sufficient schemas featuring `uniqueItems` as they were primarily intended to test other generators that do not support this feature.

6.4.1 Experimental Setup

To build the schemas, we defined four distinct meta-schemas: S_{Int}, S_{Str}, S_{Obj}, and S_{Arr}. These meta-schemas describe array schemas defining four different types of arrays:

- S_{Int} defines arrays containing items of type `integer`
- S_{Str} defines arrays where items are of type string
- S_{Obj} defines arrays with items of type `object`
- S_{Arr} defines arrays with items of type `array`

When running our optimistic generator on these meta-schemas, instead of generating instances that are JSON values, the generator produced instances that are themselves JSON schemas. The meta-schemas effectively guided the generation process, resulting in schemas representing the various data types. The meta-schemas that were defined have the following form:

```
{ "type": "object",
  "required": ["type","minItems","prefixItems","items",
    "contains","minContains","uniqueItems"],
  "properties": {
    "type": { "type": "string", "const": "array" },
    "minItems": { "type": "integer", "minimum": 5, "maximum": 10 },
    "prefixItems": S_pIt,
    "items": S_it,
    "contains": S_C,
    "minContains": { "type": "integer", "minimum": 2, "maximum": 8 },
    "uniqueItems": { "type": "boolean", "const": true }
  },
 "additionalProperties": false
}
```

Here, S_{pIt} is an array of schemas, while both S_{it} and S_C are schemas. These schemas are used to describe the values of the four data types. For example, when using the meta-schema S_{Str} to generate schemas, the schemas within S_{pIt}, S_{it}, and S_C define items that are of type string. Specifically, the schema S_C of the meta-schema S_{Str} is defined as follows:

```
{ "type": "object", "required": ["type","pattern","minLength"],
  "properties": {
    "type": { "type": "string", "const": "string" },
    "pattern": { "type": "string", "const": "a+b*.*" },
    "minLength": { "type": "integer", "minimum": 5, "maximum": 10 }
  },
 "additionalProperties": false
}
```

To construct a large collection of schemas, and given that our generator is deterministic and processes the properties within the `required` array in the order they appear in the schemas, we aimed to avoid duplicate schemas while ensuring diverse configurations with various values for all constraints. To achieve this, we implemented a randomization strategy for processing the `required` array. For each meta-schema, we conducted 500 runs of generation, with each run producing $N = 1000$ schemas. This approach allows for greater diversity in the combinations of values, effectively covering the constraints with different configurations.

In total, after duplicates removal and the exclusion of the unsatisfiable schemas, the collection comprises 35,366 unique schemas, distributed as follows based on the content of their array-schema: 5,962 schemas with arrays of integers (AInt), 5,610 with arrays of strings (AStr), 7,688 with arrays of objects (AObj), and 16,106 with arrays of arrays (AArr).

For instance, when the generator is executed on the meta-schema S_{Int}, it might produce a schema like the following, which defines arrays containing integer values with the specific constraints:

```
{ "type": "array", "minItems": 5, "uniqueItems": true,
  "items": { "maximum": 100, "type": "integer", "minimum": -10 },
  "contains": { "type": "integer", "multipleOf": 13 },
  "prefixItems": [
    { "maximum": 100, "type": "integer", "minimum": 15 },
    { "maximum": 100, "type": "integer", "minimum": -5 },
    { "maximum": 100, "type": "integer", "minimum": -10 },
    { "maximum": 100, "type": "integer", "minimum": 30 },
    { "maximum": 100, "type": "integer", "minimum": 17 }
  ],
  "minContains": 6
}
```

6.4.2 Experimental Results

Table 6.5 presents the standard metrics measured throughout all the experiments conducted, concerning correctness and execution times for each schema collection, along with the average size of the generated instances.

Coll.(size)	Success	Interrupt.	GenFail	Logical Error	Avg time (ms)	Avg Size (KB)
AInt (5,962)	100%	0%	0%	0%	0.85	0.04
AStr (5,610)	100%	0%	0%	0%	4.91	0.12
AObj (7,688)	100%	0%	0%	0%	4.85	0.47
AArr (16,106)	100%	0%	0%	0%	13.07	0.38

Table 6.5: `uniqueItems` experiments

The results presented in the table about the performance metrics show that the generator achieved a success rate of 100 for all the four schema collections. There were no interruptions or logical errors reported, reflecting the robustness of the generation process, nor generation failures, since we have not included negation in these schemas.

The observations we make regarding execution times are as follows: the collection with arrays of integers (AInt) exhibited the shortest average time at 0.85 milliseconds, as integers do not present any complexities in terms of generation. In contrast, the collections with arrays of strings (AStr) and arrays of objects (AObj) showed slightly longer average times of 4.91 milliseconds and 4.85 milliseconds, respectively. This increase in execution time can be attributed to the involvement of the Brics library, which adds some overhead during the generation process.

The collection containing arrays of arrays (AArr) had the longest average execution time at 13.07 milliseconds. This is expected due to the increased complexity associated with nested array structures. Specifically, the presence of a non-empty `contains` schema, combined with a minimum value of 3 for `minContains`, necessitates a merging of the schemas from `prefixItems` and the `contains` schema during generation. This additional requirement contributes to the increased execution time, making it higher than that of the other collections.

Overall, these results demonstrate the effectiveness of our approach in generating valid instances for schemas that contain the `uniqueItems` keyword while maintaining high performance and accuracy across schemas that describe different data types. Although the four schema collections are synthetic and do not involve very complex schemas, they feature various combinations of constraints and their values, which necessitates interactions that may require merging the schemas. These preliminary experiments were sufficiently robust to test the generation algorithm for `uniqueItems`, including its various components, such as verifying the presence of maximum matching and other related processes.

6.5 Conclusion

This chapter has provided a comprehensive evaluation of our instance generation approach for JSON Schema, demonstrating its performance and effectiveness across multiple experimental scenarios. We systematically analyzed our method through single instance and multiple instances generation experiments, as well as specific tests focusing on the

`uniqueItems` constraint.

In the single instance generation experiments, we compared our approach with existing tools, highlighting its competitive edge in terms of execution speed without sacrificing accuracy. This efficiency is crucial for real-world applications that require rapid and reliable data generation. Despite some difficulties encountered with particular schema collections, the overall results affirm our tool's capability to produce valid instances swiftly.

The multiple instances generation experiments further underscored the scalability of our method. We observed that our approach efficiently handled the generation of large sets of instances, maintaining low execution times even as the size of the desired outputs increased. The data generated during these tests showcased the tool's adaptability and performance, confirming its suitability for practical applications that demand high-volume data generation.

Lastly, the dedicated experiments on the `uniqueItems` constraint revealed the robustness of our approach in managing this intricate schema feature. The successful generation of valid instances underlined the method's reliability and its ability to navigate complex interactions within schemas.

Overall, the findings from this chapter reinforce the potential of our JSON Schema data generation tool. Its design effectively balances speed and correctness, paving the way for future enhancements that could further optimize accuracy and performance. This solid foundation encourages continued exploration and development, promising significant advancements in the domain of schema-based data generation.

Chapter 7

Hybrid Approach

In this chapter, we introduce a hybrid approach for JSON Schema data generation, which combines the strengths of both the pessimistic and optimistic strategies. The goal of this approach is to ensure a balance between accuracy and efficiency, addressing the limitations of each individual method. We provide a detailed evaluation of its effectiveness, comparing it with previous techniques, and present an optimized workflow that enhances both the correctness and performance of the generation process.

7.1 Introduction

In this chapter, we introduce an advanced generation technique that harnesses the strengths of the two previously introduced generation strategies—the pessimistic approach 4, and the optimistic approach 5—aiming at delivering a robust solution that ensures completeness, correctness, and overall efficiency for JSON Schema data generation.

The pessimistic approach, defined in Chapter 4, represents a rigorous method for generating JSON data. It guarantees that the generated data adheres meticulously to the schema's requirements. This approach ensures that every constraint and specification present in the schema is satisfied, thereby providing a high level of assurance regarding the correctness of the generated data. However, this thoroughness comes with a significant computational cost. For complex schemas, the pessimistic approach can be particularly resource-intensive, resulting in longer processing times and higher computational overhead. Additionally, this method does not handle uniqueness constraints in arrays, which may limit its applicability in certain scenarios.

To address these drawbacks, we introduced the optimistic approach, which will be the second key component of our hybrid strategy. The optimistic approach is designed to overcome the limitations of the pessimistic approach by focusing on efficiency and speed. This method sacrifices some degree of completeness but is capable of covering a much

broader spectrum of schemas. As shown in the experimental analysis section, the optimistic approach efficiently handles a wide variety of schemas, even though it does not guarantee a result for every possible schema. The trade-off between completeness and execution speed makes the optimistic approach particularly valuable for applications where rapid results are crucial, despite potential failure in generating a result.

The hybrid approach combines these two techniques, leveraging their respective strengths to create a versatile and effective solution for JSON Schema data generation. By integrating the thoroughness of the pessimistic approach with the efficiency of the optimistic approach, our hybrid methodology provides a balanced solution that can be adapted to a wide range of use cases. This dual-strategy system allows us to optimize the generation process, ensuring both the accuracy of the results and the efficiency of the computation. The following sections will delve into the details of how these techniques are combined and applied, illustrating the benefits and applications of our hybrid approach in various scenarios.

7.2 Comparative Analysis: Quantitative and Qualitative Evaluation

In this section, we provide an in-depth comparative analysis of the pessimistic approach versus the optimistic approach, aiming to highlight their respective strengths and limitations. This discussion extends the findings presented in the experimental analysis chapter 6 by offering a clearer examination of both quantitative and qualitative aspects. Our goal is to thoroughly evaluate the potential benefits and improvements that arise from the hybrid generation technique, which integrates elements of both approaches. We will explore how this combined methodology enhances success rates, efficiency, and overall effectiveness in practical applications.

To begin, we will quantitatively assess the performance of both approaches by analyzing the number of schemas where each method encounters an interruption error, fails to generate a valid witness, or determines the unsatisfiability of the schema. By comparing the outcomes between the pessimistic approach and the optimistic approach, we aim to discern any improvements or advantages that the hybrid method may offer.

In addition to the quantitative analysis, we will conduct a qualitative evaluation to gain deeper insights into the practical implications of each approach. This involves examining the contexts in which each method encounters difficulties, as well as exploring the trade-offs associated with each approach. For instance, we will investigate whether the hybrid technique provides more reliable results in scenarios where the optimistic approach alone might fall short, or if it delivers efficiency gains that outweigh the completeness sacrificed by the optimistic approach.

Our analysis will also consider various factors such as computational cost, execution time, and the complexity of schemas. By evaluating these aspects, we aim to understand how effectively the hybrid approach balances the thoroughness of the pessimistic method

with the efficiency of the optimistic approach. This comprehensive assessment will help identify areas where the hybrid technique excels and where further improvements may be needed.

Ultimately, the objective of this comparative analysis is to provide a clear understanding of how the integration of these two generation techniques impacts the overall performance and applicability of the JSON Schema data generation process. By thoroughly examining both quantitative metrics and qualitative factors, we seek to determine whether the hybrid approach offers significant advantages over the individual methods and to outline the scenarios in which it is most beneficial.

7.2.1 Github Collection

In the following, we will discuss the differences between the results of the experimental analysis of both approaches on the GitHub collection, as this particular collection presents nuances where the differences are not immediately clear and require an in-depth study and comparison.

First, we will identify the number of schemas for which one approach fails to generate a valid result and examine the outcomes of those schemas in the other approach, and vice versa, recalling, as discussed previously, that the pessimistic approach only fails due to interruption errors. Then, we will identify the reasons behind these differences in outcomes to better understand the strengths and limitations of each method. This comparison aims to determine whether combining both approaches leads to an improvement in the number of schemas for which a valid instance is generated, thereby increasing the overall success rate.

Table 7.1 provides a detailed comparison of schemas that encounter interruption errors, generation failures, and logical errors using the optimistic approach, and shows the outcomes of these schemas when evaluated with the pessimistic approach. This analysis highlights the complementary nature of the two methods: the pessimistic approach effectively resolves many issues where the optimistic approach fails, thereby improving the overall generation success rate. However, it also reveals limitations, as there are a total of 22 schemas that are not successfully handled by either method. Specifically, in the pessimistic approach, all 22 schemas encounter timeout errors, while in the optimistic approach, 18 result in timeout errors, and the remaining 4 experience generation failures.

The reasons why some schemas are successfully processed by the pessimistic approach but not by the optimistic approach are as follows:

Interrupt. For two schemas, the interruption errors in the optimistic approach occur due to unhandled patterns in JSON Schema regular expressions (cf. String generation in Section 5.3.3). In contrast, the pessimistic approach successfully processes these schemas by fully translating the regular expressions into the language supported by the Brics library, as outlined in [12, 55]. Another schema encounters issues in the optimistic approach due to its naive reference expansion mechanism, which can lead to multiple expansions of the same

schema and result in a timeout. The pessimistic approach avoids this by using a variable mechanism that ensures each schema is expanded only once, as mentioned earlier. The remaining three schemas are complex and require more processing time. As noted before, the optimistic approach is limited to a 5-minute timeout, whereas the pessimistic approach allows for up to an hour. This additional time in the pessimistic approach enables the successful processing of these schemas. Extending the timeout for the optimistic approach could potentially lead to a successful processing as well.

GenFail. The schemas that resulted in generation failures using the optimistic approach can be attributed to the inherent limitations of the method when dealing with negation. On the other hand, the pessimistic approach avoids these issues by entirely eliminating negation, as previously mentioned in Section 5.2.2. This allows it to generate valid results for these schemas.

Logical Error. Similarly to the two errors in the *Interrupt.* category, the logical errors in the optimistic approach are due to limitations in the Brics library [60, 55], whereas the pessimistic approach addresses these issues effectively through the regular expressions translation mechanism.

Optimistic		**Pessimistic Result**	
Error	**# Schemas**	**Success**	**Interrupt.**
Interrupt.	24	6	18
GenFail	52	48	4
Logical Error	42	42	-

Table 7.1: Optimistic errors VS the pessimistic results

Table 7.2 highlights the failures encountered by the pessimistic approach and their corresponding outcomes when assessed with the optimistic approach. The table shows that the optimistic approach successfully resolves the majority of issues that the pessimistic method encounters, as evidenced by the high number of successful results. However, it also confirms that there are 22 schemas that remain unresolved by both approaches, as noted in table 7.1. Among the 52 schemas that encounter errors in the pessimistic approach, 28 are due to timeouts, 23 are due to memory issues and 1 schema encounters a ref-expansion error. The memory problems arise during the translation of patterns from JSON Schema to Brics in the pessimistic approach. In contrast, the optimistic approach does not perform this translation. Since these memory issues do not occur during generation (where Brics is invoked), it indicates that the problematic patterns are not used in the generation phase. Additionally, the optimistic approach avoids the complex preparation phases present in the pessimistic approach, contributing to its better performance in handling timeouts.

To conclude the assessment of the hybrid approach's success rate on this collection, we find that combining both methods improves the overall success rate. The breakdown of

Pessimistic		Optimistic Result			
Error	**# Schemas**	**Success**	**Interrupt.**	**GenFail**	**Logical Error**
Interrupt.	52	30	18	4	-

Table 7.2: Pessimistic errors VS the optimistic results

outcomes is as follows: *success* for 6,405 schemas (99.66%), *GenFail* for 4 schemas (0.06%), and *Interrupt.* for 18 schemas (0.28%).

After confirming the improvement in the success rate, we will now analyze the execution times to determine whether the optimistic approach consistently outperforms the pessimistic method in terms of speed or if there are cases where the pessimistic approach is faster. This analysis will help us understand how to integrate both methods effectively into a hybrid approach, potentially optimizing the workflow to leverage the strengths of each method for better overall performance.

Table 7.3 provides a summary of schema execution times, comparing the number of schemas where the optimistic approach is faster, the pessimistic approach is faster, and those with identical execution times. This comparison focuses exclusively on schemas for which a correct result was returned, meaning either a valid instance was generated if the schema was satisfiable, or it was correctly identified as unsatisfiable. Schemas that were not successfully covered by either method have been excluded. The count for the optimistic approach includes schemas where it succeeded and was faster, as well as those where it succeeded while the pessimistic approach returned an interruption error. Similarly, the count for the pessimistic approach includes schemas where it succeeded and was faster, as well as those where it succeeded while the optimistic approach either returned a generation failure or encountered a logical error. The table highlights that the optimistic approach is faster for the majority of schemas (98.02%), while the pessimistic approach is faster for a smaller subset (1.94%), and a few schemas (0.04%) have equal execution times for both methods.

# Schemas	**Same Exec. Time**	**Optimistic Faster**	**Pessimistic Faster**
6,405	3	6,278	124

Table 7.3: Schema counts based on execution times

The plots in Figures 7.1 and 7.2 provide a visual confirmation of the results presented in Table 7.3, offering additional insights into the comparison of execution times between the optimistic and pessimistic approaches. The analysis covers a total of 6,279 schemas, focusing only on those for which both methods returned valid results and excluding cases where at least one method yielded an invalid result, as comparisons in such cases are meaningless.

Figure 7.1 illustrates the count of schemas processed within different time intervals, with execution times measured in milliseconds (ms). The intervals range from 0-1 ms to

over 1000 ms, providing a clear comparison between the two methods. As expected, the optimistic approach is significantly faster, dominating the lower time intervals and processing 85.47% of the schemas in the 0-5 ms range, which highlights its efficiency. Conversely, although the pessimistic approach is slower overall, it still processes a considerable number of schemas in the 0-100 ms range, demonstrating that it is still efficient for many cases.

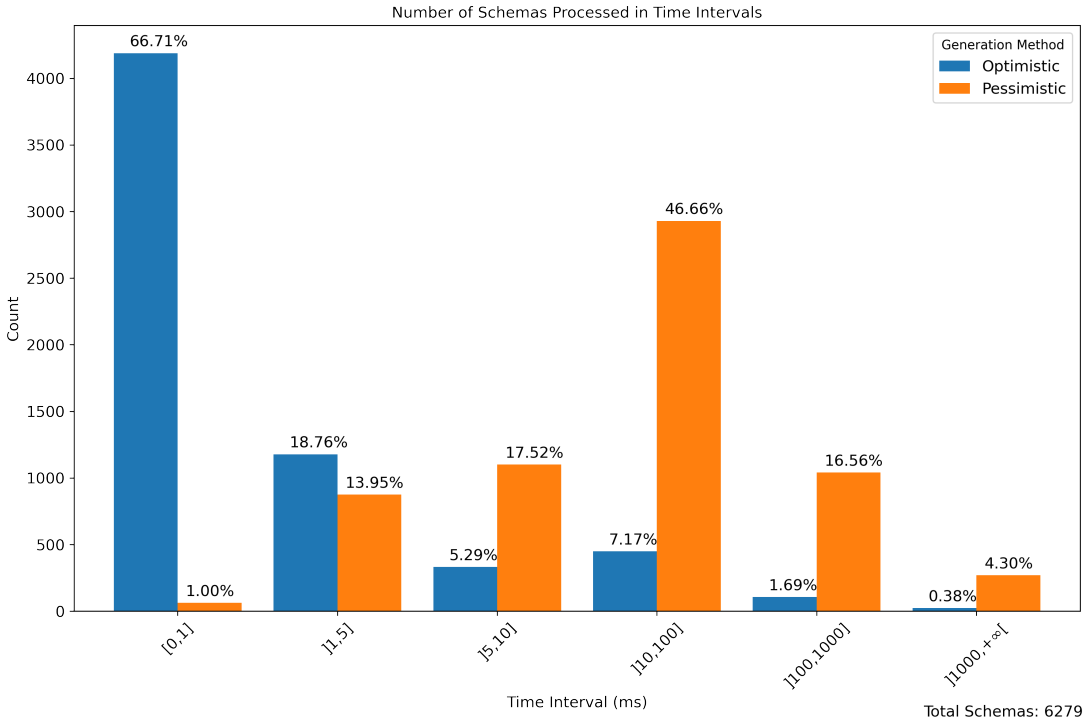


Figure 7.1: Distribution of Schemas Processed Across Time Intervals for Optimistic and Pessimistic Approaches

Figure 7.2 presents a detailed distribution of execution times for the two methods on a log-log scale. Each blue dot represents the execution time of a schema under both the optimistic and pessimistic approaches, with the x-axis indicating the optimistic approach's execution time and the y-axis indicating the pessimistic approach's execution time. The red line, labeled "Equal Times," represents the line y = x, where both approaches have the same execution time for a given schema. The dense clustering of points above the "Equal Times" line, where the x-axis (optimistic approach) values are lower than the y-axis (pessimistic approach) values, visually confirms that the optimistic approach generally outperforms the pessimistic approach in terms of speed. For many schemas, the difference in performance is substantial, with the optimistic approach being significantly faster, as illustrated by the points located at the top left of the plot. The few points below the line indicate cases where the pessimistic approach is faster than the optimistic approach. However, these points are close to the y = x line, suggesting that, despite being slower, the execution times of the optimistic approach are not far from those of the pessimistic

approach, except for the few points under the green line.

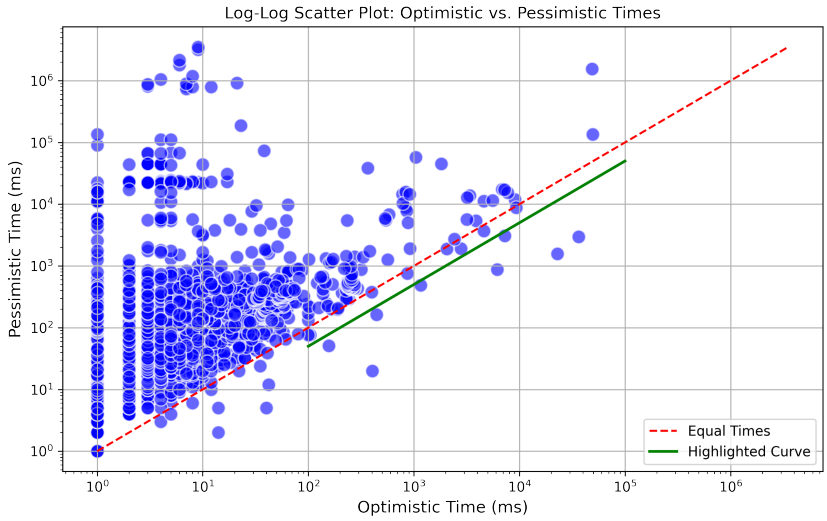


Figure 7.2: Comparison of Execution Times Between Optimistic and Pessimistic Approaches on a Log-Log Scale

Together, these plots illustrate the same trend highlighted in the table: the optimistic approach dominates in terms of speed for the majority of schemas, while the pessimistic approach is faster only in a few cases. The visualizations emphasize the distribution of execution times, with the optimistic approach consistently performing more efficiently across a wide range of schemas.

7.2.2 Other Collections

Applying the same comparison approach used for the GitHub collection, we will now analyze the results for the Snowplow, Kubernetes, WashingtonPost, and Containment-draft4 collections. The Handwritten schema collection will be excluded from the analysis due to the very low success rate of the optimistic approach compared to the pessimistic approach, as no additional gains can be achieved.

In the case of the Snowplow collection, the pessimistic solution encounters only two *Interrupt.* errors, which are also observed in the optimistic approach. For the other collections, the pessimistic approach demonstrates a 100% success rate, making further comparison to check for potential improvements unnecessary. The outcomes for these collections remain consistent with those of the pessimistic approach, maintaining the following success rates: Kubernetes (100%), Snowplow (99.52%), WashingtonPost (100%), and Containment-draft4 (100%).

Following the success rate analysis, we now turn our focus to execution times across the different schema collections. Table 7.4 summarizes this comparison by showing the number

of schemas where each approach is faster or where their execution times are identical within the four collections. Specifically, the optimistic approach significantly surpasses the pessimistic approach in three collections. In the WashingtonPost collection, it is always faster (100%). In the Kubernetes and Snowplow collections, it is quicker for almost all schemas (99.27% and 98.56%, respectively). In contrast, the Containment-draft4 collection shows a more balanced distribution, with the optimistic approach being faster in 63.41% of schemas and the pessimistic approach in 31.03%. This balance is due to the optimistic approach returning a *GenFail* for 19.69% of the schemas, as detailed in Table 6.1, leading the pessimistic approach to be considered faster in those cases, as it successfully returns a valid result.

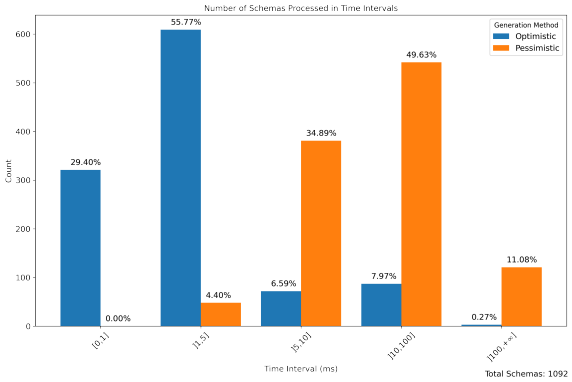
collection	# Schemas	Same Exec. Time	Optimistic Faster	Pessimistic Faster
K8s	1,092	3	1,084	5
Snw	418	0	412	6
WP	125	0	125	0
CC4	1,331	74	844	413

Table 7.4: Schema counts based on execution times

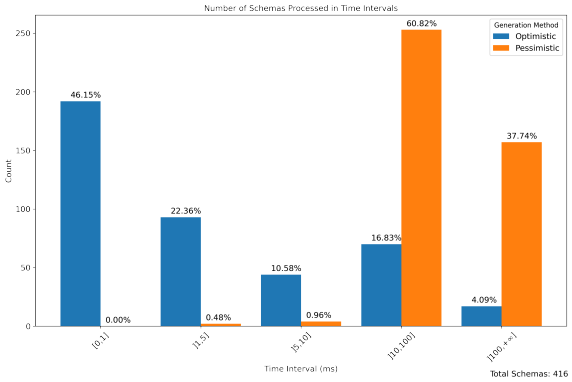
Figure 7.3 shows the number of schemas processed within different time intervals for the various collections. As observed in the GitHub collection, the optimistic approach predominantly occupies the lower time intervals, except in the WashingtonPost collection, where Figure 7.3c shows a more balanced distribution across time intervals. In contrast, the pessimistic approach is more concentrated around the middle and higher time intervals for all schema collections.

Figure 7.4 provides a detailed log-log scale distribution of execution times for the two methods across all schema collections. The different plots confirm our previous observations from the comparison of the GitHub collection; here, too, we see that the optimistic approach is faster for the majority of schemas. Specifically, in the plots of the Kubernetes, WashingtonPost, and Containment-draft4 collections (Figures 7.4a, 7.4c, and 7.4d, respectively), the differences in execution times are generally small, except for some outliers in the WashingtonPost collection. These outliers are represented by points at the top middle of the plot, where the difference is significant (around 10^2ms and 10^3ms for the optimistic approach, and over 10^5ms for the pessimistic approach). The substantial difference in these cases is explained by the object and array preparations steps that these schemas undergo in the pessimistic method, while the optimistic approach uses simpler preprocessing, suggesting that some parts of the schemas that were prepared are not crucial for generation. Regarding the Snowplow collection, represented by Figure 7.4b, we observe large differences in many schemas. This is due to the preprocessing of strings in the pessimistic approach, where the use of the Brics library results in additional manipulations that are sometimes not essential for generation. In contrast, the optimistic approach, although it also uses the Brics library, utilizes it only during the generation phase and not during canonicalization,

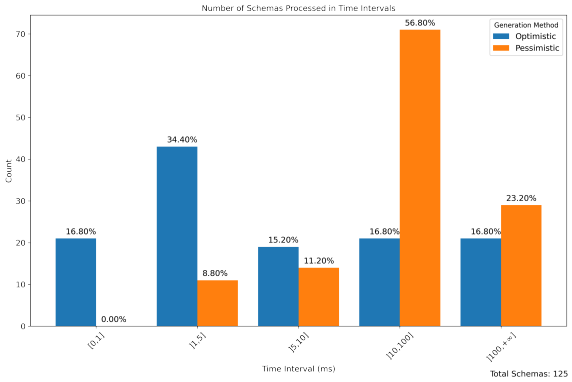
resulting in a more straightforward process without any extra overhead.



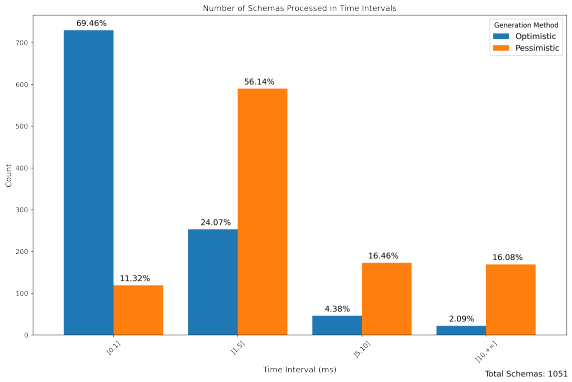
(a) Kubernetes collection



(b) Snowplow collection



(c) WashingtonPost collection



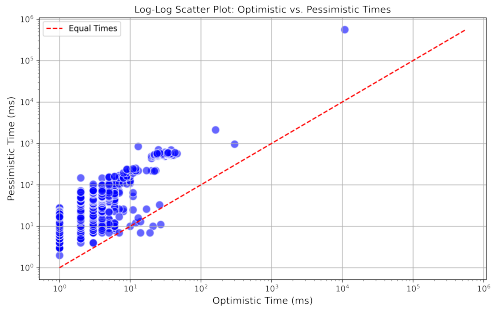
(d) Containment-draft4 collection

Figure 7.3: Distribution of Schemas Processed Across Time Intervals for Optimistic and Pessimistic Approaches

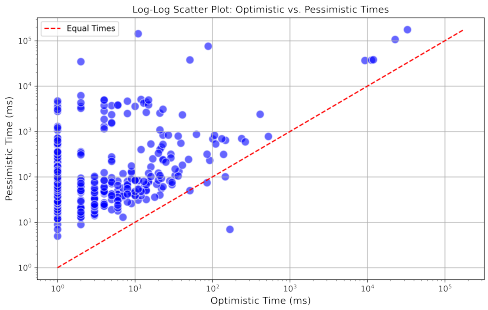
7.2.3 Conclusive Remarks

In conclusion, the refined analysis and detailed comparison between the optimistic and the pessimistic approaches suggest that developing a hybrid method integrating both approaches is both valuable and promising. Combining these methods can significantly enhance the overall success rate of generating correct JSON instances and determining schema unsatisfiability. Additionally, it ensures high coverage for JSON Schema data generation by addressing the limitations inherent in each method. For example, the hybrid approach can address the pessimistic approach's lack of handling the `uniqueItems` constraint and the optimistic approach's incomplete handling of negation, thus providing a more comprehensive and robust solution.

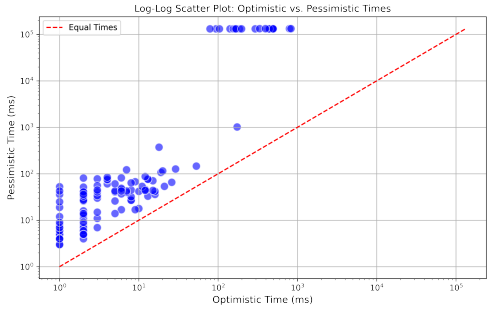
Our evaluation of execution times reveals that the optimistic approach generally outperforms the pessimistic approach in terms of speed for most schemas across all schema



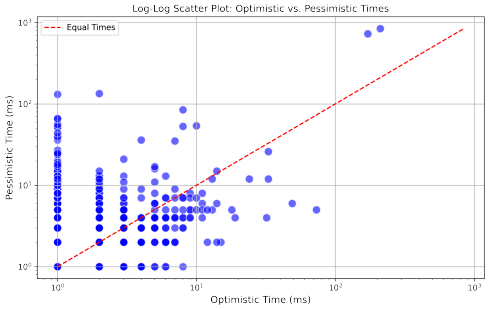
(a) Kubernetes collection



(b) Snowplow collection



(c) WashingtonPost collection



(d) Containment-draft4 collection

Figure 7.4: Comparison of Execution Times Between Optimistic and Pessimistic Approaches on a Log-Log Scale

collections. The figures presented in the previous sections visually confirm this trend, illustrating that the optimistic approach dominates the lower time intervals and shows superior efficiency across a wide range of schemas. Specifically, in the GitHub collection, the optimistic approach processes the majority of schemas in the fastest time intervals, while the pessimistic approach is more concentrated in the middle and higher time intervals. Further analysis of the Snowplow, Kubernetes, WashingtonPost, and Containment-draft4 collections reaffirms these findings. The optimistic approach consistently surpasses the pessimistic approach in three collections, demonstrating superior speed. The WashingtonPost collection is notable for its balanced distribution across time intervals, while in the Containment-draft4 collection, the performance is more balanced due to the optimistic approach returning a *GenFail* for some schemas.

Overall, while the optimistic approach proves to be faster and more efficient in the majority of cases, the detailed examination of execution times and the specific characteristics of each schema collection highlight the nuanced performance differences between the two methods. These insights are valuable for optimizing the hybrid approach and tailoring it to leverage the strengths of both methods effectively.

7.3 Hybrid Approach Workflow

In this section, we discuss the adaptive strategies that form the foundation of the hybrid approach for JSON Schema data generation. The analysis conducted in the previous section will guide us in developing various methodologies for integrating the optimistic and pessimistic approaches. These methodologies aim to intelligently combine the strengths of both approaches to achieve an optimal balance between efficiency and accuracy. We will first introduce a naive strategy and then present more sophisticated strategies that dynamically adapt based on the characteristics of the schema being processed.

7.3.1 Naive Approach

Since the analysis performed in the previous section and the detailed comparison between the two methods revealed that the optimistic approach is faster in most cases, a straightforward strategy for building the hybrid system involves initially attempting to generate JSON data using this method. The remarkable speed and efficiency of the optimistic approach make it an ideal starting point. If this initial attempt fails, the hybrid system then falls back on the pessimistic method. By adopting this two-step process, the naive strategy for the hybrid system effectively combines the strengths of both methods: leveraging the speed of the optimistic approach for the majority of cases while ensuring completeness through the pessimistic approach when necessary.

Table 7.5 summarizes the correctness results and execution times for the naive hybrid approach across across the various schema collections. It includes columns for the success rate, interruption errors (*Interrupt.*), generation failures (*GenFail*), logical errors, and execution times, with both median and average values reported in milliseconds. The success rates for each collection have already been discussed in the previous section, so we will not revisit them here. Instead, we will focus on analyzing the execution times. This analysis aims to assess the efficiency of the naive approach and identify both its advantages and drawbacks. By examining its performance across different schema collections, we seek to uncover trends and areas where optimizations and improvements may be needed, ultimately refining the approach to achieve a better balance between speed and correctness.

collection	Success	Interrupt.	GenFail	Logical Error	Time (ms) med./avg.
Git	99.66%	0.28%	0.06%	0%	1/813
K8s	100%	0%	0%	0%	2/15
Snw	99.52%	0.48%	0%	0%	2/225
WP	100%	0%	0%	0%	5/65
CC4	100%	0%	0%	0%	1/7

Table 7.5: Naive hybrid approach: correctness results and execution times

The time measurement for the naive hybrid approach is structured to account for the combined execution of both the optimistic and pessimistic methods, depending on the

outcome of the optimistic approach. Specifically, if the optimistic method successfully generates a valid result, the recorded time is solely the time taken by the optimistic approach. However, if the optimistic approach produces an invalid result, such as a logical error or a *GenFail*, the time measured is the cumulative duration of the optimistic attempt plus the time required by the pessimistic method to generate a valid result. In cases where the optimistic approach encounters an interruption error, the time measurement varies based on the type of error. For a timeout error, the recorded time includes the 5-minute timeout duration plus the time the pessimistic method takes to generate a valid result. If the interruption error is of a different nature, the time measured combines the duration needed to detect the error, along with the subsequent time taken by the pessimistic method to produce a valid outcome.

The table reveals that the median execution times for all schema collections align precisely with the median times of the optimistic approach. This outcome was expected, given the results of the analysis conducted in Chapter 6, which highlighted the optimistic approach's high success rate and speed. Since the naive hybrid system predominantly relies on the optimistic method, the consistency in median times suggests that the system effectively leverages the strengths of the optimistic approach in the majority of cases.

Another clear observation from the table is that, for the Kubernetes and WashingtonPost collections, the execution times match those of the optimistic approach exactly. This result is due to the 100% success rate of the optimistic approach in these collections, meaning there was no need to resort to the pessimistic approach.

In contrast, the other collections show an increase in average execution time. However, this increase is almost insignificant in the Snowplow and Containment-draft4 collections. In Snowplow, only 2 schemas required reprocessing with the pessimistic method because the optimistic approach initially produced invalid results. Similarly, in the Containment-draft4 collection, despite the hybrid approach need to fall back on the pessimistic method for 20.59% of the schemas, the average execution time increase is minimal. This is attributed to the low execution times of the pessimistic method in this collection. Compared to the two collections, the GitHub collection exhibits a noticeable increase in the average execution time. This increase can be attributed partly to the reprocessing of 90 schemas that initially yielded incorrect results with the optimistic approach, requiring a fallback to the pessimistic method. Additionally, a major contributor to the extended execution time is the processing of 6 schemas that encountered interruption errors. Specifically, 4 of these errors were timeout errors in the optimistic approach, which also resulted in lengthy processing times with the pessimistic method. The remaining 2 errors were of other types, detected immediately.

In conclusion, the naive hybrid approach offers several key advantages:

- **Simplicity in Setup:** It requires neither additional computational resources nor complex schema manipulations and verifications.
- **Efficiency:** By leveraging the speed of the optimistic approach, the hybrid method is highly efficient, particularly for schemas that the optimistic method handles correctly.

- **Execution Time:** As observed across various collections, the execution time of the naive hybrid approach is nearly identical to that of the optimistic approach, with the exception of the GitHub collection.

- **Performance:** It outperforms the pessimistic method, and its success rates are at least as good as, if not better than, those of both methods when run individually.

However, the approach does have limitations. In particular, as seen in the GitHub collection, it can experience significant increases in execution time, primarily due to handling timeout errors or schemas that the optimistic approach fails to process correctly. These drawbacks highlight areas where the naive hybrid approach can be optimized. Addressing these issues could further enhance its efficiency and effectiveness, making it an even more robust solution for JSON Schema data generation.

7.3.2 Fallback Mechanism Optimization

While the naive hybrid approach provides a solid foundation, there are opportunities to refine its fallback mechanism for better performance. In this section, we explore ways to optimize the process of switching between the optimistic and pessimistic methods. These optimizations aim to reduce the time overhead that occurs during fallback scenarios, thereby improving the overall efficiency of the hybrid approach. By enhancing the decision-making process and introducing smarter fallback strategies, we aim to create a more efficient and effective system for JSON Schema data generation.

Post-Canonicalization Switch

A first optimization strategy we propose involves shifting the fallback decision point to after the schema has been canonicalized. This strategy aims to switch to the pessimistic approach earlier in the process. Instead of relying on failures during generation, logical errors, or interruption issues to trigger the fallback, this strategy evaluates the schema's canonical form to determine if the optimistic approach should proceed.

The primary motivation for this strategy is that, once a schema is in its canonical form, potential issues become easier to identify due to its normalized structure, making generation-related problems more straightforward to detect. Additionally, previous experiments have shown that across various schema collections, the generation phase is generally more time-consuming than the canonicalization phase for many schemas, particularly in the GitHub collection. By implementing the fallback decision after canonicalization, we can bypass the generation phase for schemas that are unlikely to produce successful results, thus saving both computational resources and time.

The primary focus of this optimization is to address generation failures. For interruption errors, timeout errors occur during the canonicalization phase, making it impractical to address them post-canonicalization. Other interruption errors are detected instantaneously, so the optimization does not significantly impact their handling, as these errors

are identified quickly during the generation phase. Regarding logical errors, the optimization offers minimal benefit. These errors primarily result from current limitations in the implementation of the optimistic approach and would require a translation mechanism for regular expressions from JSON Schema to Brics [60] to be effectively resolved

In the following, we will detail how this post-canonicalization strategy functions, present some results, and conclude by discussing its advantages and limitations.

Methodology. In the post-canonicalization decision strategy, the process begins with executing the optimistic method until the schema has been canonicalized. Once the schema is in its normalized form, our static verification checks whether the optimistic approach can successfully generate JSON data. If this verification indicates a potential failure, the hybrid system bypasses the optimistic generation attempt and directly invokes the pessimistic method. This approach ensures that the optimistic method is only used when it has a reasonable chance of success. The verification process on the canonicalized schema must be efficient and less costly compared to the actual generation process.

The decision to either continue using the optimistic approach for generation or switch to the pessimistic method is based on a static analysis of the canonicalized schema. Given the incompleteness of the optimistic method, negation and conjunction can still be present in the schema, potentially complicating the generation process. The static verification involves traversing the schema to check for these operators in fragments that are essential for generation. This process is not costly and helps determine the likelihood of success for the optimistic approach. By ensuring that the optimistic approach generation step is applied only when appropriate, this static analysis optimizes the overall generation process.

Before describing how the static verification is performed, let us introduce examples of schemas for which generation will fail. These examples will help clarify the types of schema structures that pose challenges during generation, particularly when they include object or array constraints essential to fulfilling the schema's overall specifications. Once these examples are presented, we will move on to explaining the static verification process and how it determines when generation is likely to fail.

Example 18 *Consider the schema* `{ "not": { "required": ["b"] } }`. *After applying the canonicalization process to this schema, it is transformed into the normalized schema:* `{ "not": { "type": "object", "required": ["b"] } }`.

Now, consider the following schema in its canonical form that contains the previous schema as a fragment. This schema restricts the structure of an object such that `"a"` *must exist and must not be an object containing a property* `"b"`. *Since the schema corresponding to the property* `"a"` *is a schema containing the negation operator, the generation of a value for* `"a"` *will not proceed and returns* GenFail, *and because* `"a"` *is a required property, this failure leads to the overall schema generation failing as well.*

```
{ "type": "object",
  "properties": { "a": { "not": { "type": "object", "required": ["b"] } } },
  "required": ["a"]
}
```

Example 19 *Consider the following canonical schema that defines object instances where the property* `"a"` *must exist and its value must be an array with at least two items. Additionally, the keyword* `patternProperties` *specifies that the value of any property that starts with* `"a"` *followed by any number of occurrences of a given character must be an array with at least one item. Furthermore, these arrays must not contain any objects that have a property* `"b"`*. Given these constraints, optimistic generation will fail for this schema because it is not possible to generate valid value for the property* `"a"`*. Consequently, generating a valid instance for the entire schema is impossible.*

```
{ "type": "object", "required": ["a"],
  "properties": { "a": { "type": "array", "minItems": 2 } },
  "patternProperties": {
    "^a.*$": {
      "type": "array", "minItems": 1,
      "items": { "not": { "type": "object", "required": ["b"] } }
    }
  }
}
```

Following the examples, we now present the verification process for detecting schema structures that lead to generation failures. As previously mentioned, our goal is to maintain a verification process that is efficient and not costly. To achieve this, we focus on schema structures characterized by straightforward conditions that do not involve complex computations or intricate verification.

To detect these schema structures, we define a recursive function *isFail* that determines whether generation for a schema will fail. This function is defined as a set of rules denoted by the judgment $S \ \rightarrow r$, taking a schema S as its parameter and returning a boolean value r, where $r = \mathcal{T}$ $(true)$ indicates that generation will fail for S.

The rules (not-object-fail) and (not-array-fail) describe fundamental failure cases that illustrate the smallest fragments of schemas where generation will fail. These cases cover the generation process for negation (i.e. *notTA*) as outlined in the main generation algorithm 5.5. Specifically, these rules apply to schemas resulting from the canonicalization of `{(objKW)`$^+$`}` and `{(arrKW)`$^+$`}`, where *objKW* and *arrKW* denote an object and an array keyword, respectively. These canonical forms represent the basic scenarios where schema generation is inherently problematic due to their structure. By addressing these fundamental fragments, we establish the minimal conditions under which schema generation fails, providing a foundation for understanding more complex failure cases.

$$\frac{S = \{\, \texttt{not} : \{\, \texttt{type} : \texttt{object}, (objKW)^+ \,\} \,\}}{S \ \rightarrow \mathcal{T}} \quad \text{(not-object-fail)}$$

$$\frac{S = \{\, \texttt{not} : \{\, \texttt{type} : \texttt{array}, (arrKW)^+ \,\} \,\}}{S \ \rightarrow \mathcal{T}} \quad \text{(not-array-fail)}$$

The following rules (`allOf`-fail) and (`anyOf`-fail) describe generation failures for conjunctions and disjunctions. These cases correspond to the generation process for conjunction and disjunction outlined in the main generation algorithm 5.5. Generation will fail for the conjunction if it contains at least one sub-schema for which generation fails. Regarding disjunction, generation will fail if it fails for all of its sub-schemas.

$$\frac{S = \{\, \texttt{allOf} : [\, S_1, \ldots, S_n \,]\,\} \qquad r = \vee_{i=1}^{n} isFail(S_i)}{S \;\rightarrow r} \quad (\texttt{allOf-fail})$$

$$\frac{S = \{\, \texttt{anyOf} : [\, S_1, \ldots, S_n \,]\,\} \qquad r = \wedge_{i=1}^{n} isFail(S_i)}{S \;\rightarrow r} \quad (\texttt{anyOf-fail})$$

The rule (`required`-fail) specifies that generation will fail for the schema S if a required property matches a pattern whose corresponding schema leads to generation failure. This rule covers the scenario outlined in Example 18.

$$\frac{\begin{array}{c} S = \{\texttt{type} : \texttt{object}, \texttt{patternProperties} : \{\, p_1 : S_1, \ldots, p_m : S_m \,\}, \\ \texttt{required} : [\, k_1, \ldots, k_n \,], \vec{K}\} \\ r = \exists i \in \{\!|1 \ldots n|\!\}, j \in \{\!|1 \ldots m|\!\}.\ (k_i \in L(p_j)) \wedge isFail(S_j) \end{array}}{S \;\rightarrow r} \quad (\texttt{required-fail})$$

The second rule for objects, denoted (`minProperties`-fail), addresses the schema structures illustrated in Example 19. It asserts that if the schema requires instances to have at least one key-value pair, and if `patternProperties` contains patterns whose corresponding schemas lead to generation failures, then generation will fail for S.

$$\frac{\begin{array}{c} S = \{\texttt{type} : \texttt{object}, \texttt{patternProperties} : \{\, p_1 : S_1, \ldots, p_m : S_m \,\}, \\ \texttt{minProperties} : min, \vec{K}\} \\ r = (min \geq 1) \wedge (\wedge_{i=1}^{m} isFail(S_i)) \end{array}}{S \;\rightarrow r} \quad (\texttt{minProperties-fail})$$

The rule (`contains`-fail) states that in the presence of the `contains` constraint in an array schema, generation will fail if generation for the schema of `contains` results in *GenFail* and if the function $minCinS(S)$ returns $\mathcal{T}$ $(true)$. This function checks the condition $(\texttt{minContains} \notin keys(S)) \vee (\texttt{minContains} \in keys(S) \wedge \mathrm{S}(\texttt{minContains}) \geq 1)$, which determines whether the `contains` constraint must be satisfied. Here, $keys(S)$ returns the keywords present in S, and $S(\texttt{minContains})$ returns the value of `minContains` in S.

$$\frac{\begin{array}{c} S = \{\texttt{type} : \texttt{array}, \texttt{contains} : S_c, \vec{K}\} \\ r = minCinS(S) \wedge isFail(S_c) \end{array}}{S \;\rightarrow r} \quad (\texttt{contains-fail})$$

The rule (contains-prefixItems-fail) indicates that if the `contains` constraint is present in the schema S and must be satisfied, and if the schema also contains the `prefixItems` array, then generation will fail for the whole schema if it fails for the first schema in `prefixItems`. This is because any instance that satisfies the schema must include at least one item that meets the `contains` constraint. Therefore, if generation fails for the first schema in `prefixItems`, denoted S_1, the generation process for the entire schema S will fail.

$$\frac{S = \{\texttt{type} : \texttt{array}, \texttt{contains} : S_c, \texttt{prefixItems} : [\, S_1, \ldots, S_n \,], \vec{K}\} \quad r = \mathit{minCinS}(S) \wedge \mathit{isFail}(S_1)}{S \ \rightarrow r} \ (\texttt{contains-prefixItems-fail})$$

In the presence of the `contains` constraint in an array schema, where it must be satisfied, and with `items` included but `prefixItems` absent, meaning that the items of any valid instance must conform to the specifications of `items`, the rule (contains-items-fail) states that if the generation process results in *GenFail* for the schema of `items`, then the generation for the entire schema S will fail.

$$\frac{S = \{\texttt{type} : \texttt{array}, \texttt{contains} : S_c, \texttt{items} : S_{it}, \vec{K}\} \quad r = \mathit{minCinS}(S) \wedge \texttt{prefixItems} \notin \mathit{keys}(S) \wedge \mathit{isFail}(S_{it})}{S \ \rightarrow r} \ (\texttt{contains-items-fail})$$

The rule (minItems-prefixItems-fail) specifies that generation for the schema S will fail under two conditions: first, if the schema requires instances to have at least min items, and second, if generation fails for any schema in `prefixItems` where the index is less than or equal to min. When both conditions are met, the generation process for the entire schema S will fail.

$$\frac{S = \{\texttt{type} : \texttt{array}, \texttt{minItems} : \mathit{min}, \texttt{prefixItems} : [\, S_1, \ldots, S_n \,], \vec{K}\} \quad r = (\mathit{min} \geq 1) \wedge (\exists i \in \{\!|1 \ldots n|\!\}.\ (i \leq \mathit{min}) \wedge \mathit{isFail}(S_i))}{S \ \rightarrow r} \ (\texttt{minItems-prefixItems-fail})$$

Similar to the rule (contains-items-fail), the rule (minItems-items-fail) indicates that in the absence of `prefixItems` and the presence of `items`, generation will fail for S if S requires the instances to have at least one item and the generation for the schema of `items` results in failure. This rule covers the scenario involving the array sub-schema within `patternProperties` in Example 19.

$$\frac{S = \{\texttt{type} : \texttt{array}, \texttt{minItems} : \mathit{min}, \texttt{items} : S_{it}, \vec{K}\} \quad r = (\mathit{min} \geq 1) \wedge \texttt{prefixItems} \notin \mathit{keys}(S) \wedge \mathit{isFail}(S_{it})}{S \ \rightarrow r} \ (\texttt{minItems-items-fail})$$

Finally, rule (`minItems-prefixItems-items-fail`) addresses scenarios where both `prefixItems` and `items` are present. If the schema requires more items than are specified in `prefixItems`, and if the schema for `items` leads to a generation failure, then the generation for the entire schema S will also fail

$$\frac{S = \{\texttt{type} : \texttt{array}, \texttt{minItems} : min, \texttt{prefixItems} : [\, S_1, \ldots, S_n \,], \texttt{items} : S_{it}, \vec{K}\} \qquad r = (min > n) \wedge isFail(S_{it})}{S \ \rightarrow r} \ (\texttt{minItems-prefixItems-items-fail})$$

Results. To assess the benefits of this optimization, we analyzed the execution times for schemas where generation resulted in failures. By focusing on these cases, we aimed to determine whether the optimization effectively reduces processing time and improves performance. This examination allows us to verify if the post-canonicalization switch successfully identifies problematic schemas earlier, thereby avoiding costly generation attempts and contributing to a more efficient overall process.

Figure 7.5 displays the distribution of execution times for the generation phase of schemas in the GitHub collection that resulted in generation failures. Out of the 52 schemas that encountered generation failures, 45 had execution times of 0 milliseconds, as shown in the plot. For the remaining schemas, while the execution times were not zero, they were still relatively insignificant. This distribution highlights that the generation phase generally completed almost immediately for most schemas, indicating that the proposed optimization does not significantly impact or improve the handling of these specific schemas. This outcome is attributed to the fact that the majority of these schemas triggered generation failures through the basic cases addressed by the main generation algorithm 5.5, with their canonical forms commonly featuring either a negation or conjunction operator at the root level. Regarding the other collections, generation using the optimistic approach resulted in failures only for the Containment-draft4 collection, where, similarly, these failures were detected immediately.

Even though the proposed optimization doesn't target logical errors, we are still interested in examining the execution times of schemas that encountered these errors. By analyzing these times, we aim to gain further insights into how the generation phase performs in these cases. This analysis will provide a more complete picture of the overall execution times, even for scenarios not directly improved by the optimization.

Figure 7.6 highlights the execution times for schemas encountering logical errors in the GitHub collection. The generation phase completes almost immediately for the majority of schemas, with only a few outliers showing longer execution times, typically due to complex patterns. Similarly, for the other collections, Snowplow and Containment-Draft4, the analysis of schemas with logical errors reveals the same behavior, as generation also concludes quickly across these collections.

In conclusion, while the post-canonicalization fallback optimization represents a well-considered effort to improve the hybrid approach, it comes with notable limitations. The

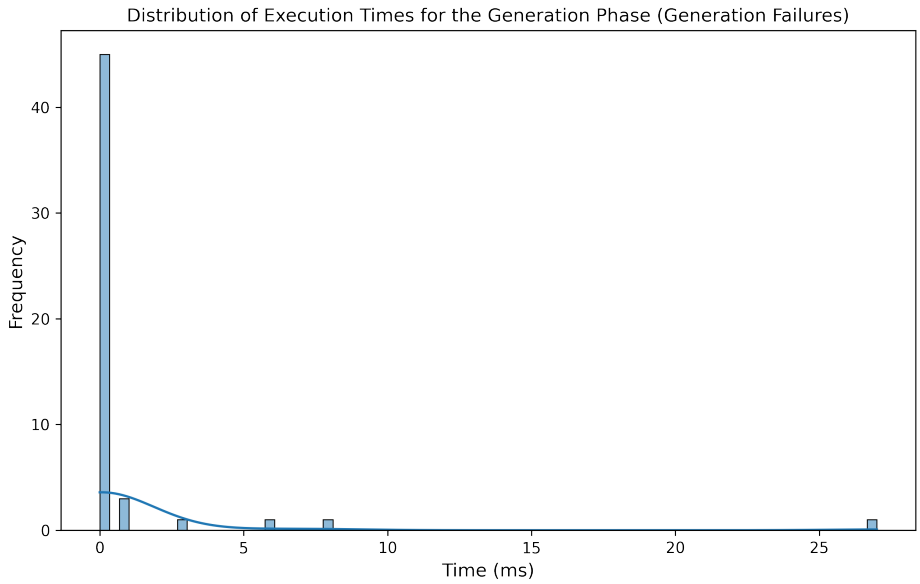


Figure 7.5: Distribution of Execution Times for the Generation Phase of Failed Schemas in the GitHub Collection

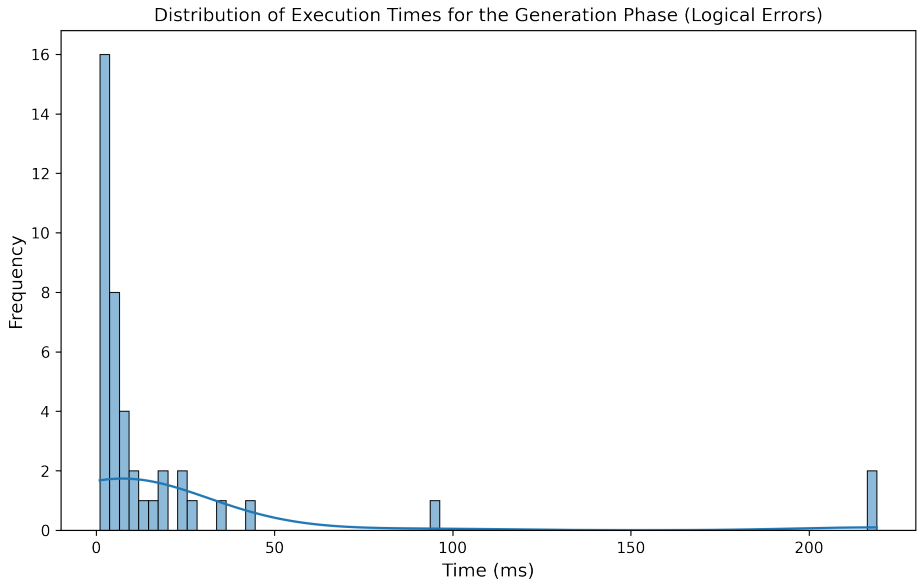


Figure 7.6: Distribution of Execution Times for the Generation Phase of Schemas with Logical Errors in the GitHub Collection

primary drawback is that this optimization focuses solely on generation failures, leaving interruption errors unresolved. Since timeout errors typically arise during the canonicalization phase, optimizing post-canonicalization provides no solution for these cases. Additionally, logical errors remain unaddressed by this strategy due to inherent limitations in the optimistic approach, particularly in processing complex regular expressions.

Furthermore, the analysis of execution times for schemas encountering generation failures indicates that this optimization yields minimal gains in practice. For both the GitHub and Containment-Draft4 collections, generation failures were detected almost immediately, offering no significant performance improvements. These observations suggest that while the optimization reduces some overhead, its overall impact may be limited for certain schema collections.

Despite these observations, it is important to note that the schema collections used in this study are specific examples. In more diverse or complex scenarios, this optimization has the potential to enhance JSON Schema data generation, particularly for schemas where the generation phase is time-intensive and prone to failure. Therefore, while the current benefits are limited, this strategy might still offer advantages in practical scenarios involving more complex schema structures.

Pre-Canonicalization Switch

In continuation of our efforts to improve the hybrid approach and address the limitations of the previous optimization strategy, we introduce the pre-canonicalization switch optimization. This new strategy proposes moving the fallback decision point to before the canonicalization phase begins. The goal is to address timeout errors more effectively and enhance the overall efficiency of the hybrid method by avoiding both the canonicalization and generation phases for schemas that are likely to result in generation failures.

The core idea behind this approach is to assess the schema's potential for failure before starting canonicalization. By identifying problematic schemas earlier in the process, we can avoid triggering canonicalization and generation phases for schemas that are expected to fail. This proactive strategy aims to save computational resources and reduce processing time, thereby enhancing the overall efficiency of the hybrid approach pipeline.

In the sections that follow, we will detail the mechanics of the pre-canonicalization switch strategy, present results from its implementation, and discuss its advantages and limitations. This strategy is designed to address the inefficiencies observed with the previous optimization by focusing on early detection of issues and reducing unnecessary processing.

Methodology. In the pre-canonicalization switch strategy, the evaluation of the schema starts immediately after the reference expansion phase. This early assessment aims at identifying schemas that are likely to fail, thereby allowing us to bypass both the canonicalization and generation phases for such schemas and directly call the compete and correct approach.

To implement this strategy, we use a recursive function named *isFail2*. Similar to the *isFail* function used in the post-canonicalization switch, *isFail2* determines the likelihood of failure for a schema based on its structure following reference expansion. Defined by a set of rules, this function serves the same purpose of early failure detection. As with our previous approach, this early detection process is designed to be efficient and less costly than the canonicalization and generation phases.

The rules used in the function *isFail2* to capture failures for schemas featuring disjunction, as well as those handling schemas that describe arrays and objects, are similar to those defined for the *isFail* function in the post-canonicalization switch. For disjunction, failure is determined when generation fails for all its sub-schemas. Similarly, for schemas describing arrays and objects, the failure conditions follow the same rules established for *isFail*, ensuring consistency in the evaluation of these schema structures across both optimization strategies.

The rule (not-fail2) is used to detect generation failures in specific schema structures that contain the keyword `not`. The schema S may include additional keywords, represented by the list of assertions $\vec{K}$. Regardless of the other keywords present, under certain conditions, the sub-schema S' associated with the keyword `not` is sufficient to determine whether generation will result in a *GenFail* outcome for the entire schema S. The function *isDisjunct* takes the schema S' and returns $\mathcal{T}$ if those specific conditions are met, indicating that generation will fail.

$$\frac{S = \{\, \texttt{not} : S', \vec{K} \,\} \qquad \texttt{type} \notin \mathit{keys}(S) \qquad r = \mathit{isDisjunct}(S')}{S \;\rightarrow r} \qquad \text{(not-fail2)}$$

Specifically, the function *isDisjunct* verifies whether a schema is of the form `{(type:`$allTypes$`)`$^?$`,(objKW | arrKW)`$^+$`}`. Here, *allTypes* is an array containing all the JSON Schema data types, and the presence of the keyword `type` is optional. The terms *objKW* and *arrKW* denote an object and an array keywords, respectively, with the condition that at least one of these keywords must be included. *isDisjunct(S')* returns $\mathcal{T}$ if:

- $S' = \{\, KW_1, \ldots, KW_n \,\}$ where $\forall i \in \{\!|1 \ldots n|\!\}$. KW_i is an object or an array keyword.
- $S' = \{\, \texttt{type} : \mathit{allTypes}, KW_1, \ldots, KW_n \,\}$ where $\forall i \in \{\!|1 \ldots n|\!\}$. KW_i is an object or an array keyword.
- $S' = \{\, \texttt{anyOf} : [\, S_1, \ldots, S_n \,] \,\}$ where $\forall i \in \{\!|1 \ldots n|\!\}$. KW_i is an object or an array keyword and $\nexists i, j \in \{\!|1 \ldots n|\!\}$. $j \neq i$ where $S_i = \{\, objKW_1, \ldots, objKW_m \,\}$ and $S_j = \{\, arrKW_1, \ldots, arrKW_l \,\}$

In addition to the rule used to detect failures in schemas containing conjunctions, as defined in the previous optimization strategy, which specifies that a generation failure in a sub-schema leads to the failure of the root schema, the function *isFail2* incorporates this

rule as well. However, it also includes additional rules to handle a wider range of schema structures.

The following rules, (`allOf-fail-obj`) and (`allOf-fail-arr`), are designed to identify generation failures in schemas describing conjunctions involving objects and arrays, respectively. In these rules, none of the individual sub-schemas within the conjunctions causes generation failure on its own. For instance, in the rule for objects, the schema with negation effectively represents a disjunction of all other types combined with the negation of the object schema. Since any value of a different type is valid, this schema alone does not result in generation failure. However, when combined with other schemas that exclusively describe objects, the values of other data types become invalid. Because the negation for objects cannot be removed, the schema ends up with persistent negation, leading to generation failure for the entire schema.

$$\frac{\begin{array}{c} S = \{\, (\texttt{type} : \texttt{object})^?, \texttt{allOf} : [\, S_1, \ldots, S_n \,], \vec{K} \,\} \qquad n \geq 2 \\ r = \exists i \in \{\!|1 \ldots n|\!\}.\ S_i = \{\, \texttt{not} : \{\, \texttt{type} : \texttt{object}, (objKW)^+ \,\} \,\} \\ \wedge\ \forall j \in \{\!|1 \ldots n|\!\}.\ j \neq i, S_j = \{\, \texttt{type} : \texttt{object}, (objKW)^* \,\} \end{array}}{S \ \rightarrow r} \ (\texttt{allOf-fail-obj})$$

$$\frac{\begin{array}{c} S = \{\, (\texttt{type} : \texttt{array})^?, \texttt{allOf} : [\, S_1, \ldots, S_n \,], \vec{K} \,\} \qquad n \geq 2 \\ r = \exists i \in \{\!|1 \ldots n|\!\}.\ S_i = \{\, \texttt{not} : \{\, \texttt{type} : \texttt{array}, (arrKW)^+ \,\} \,\} \\ \wedge\ \forall j \in \{\!|1 \ldots n|\!\}.\ j \neq i, S_j = \{\, \texttt{type} : \texttt{array}, (arrKW)^* \,\} \end{array}}{S \ \rightarrow r} \ (\texttt{allOf-fail-arr})$$

In addition, since the verification process is conducted before canonicalization, the boolean operator `oneOf` is also included in this failure detection strategy. We use the following rule to detect schema structures that cause a schema containing this operator to result in generation failures. Given the complexity of this operator, our focus is on covering basic schema structures that do not require complex verification. These schema structures were discovered after analyzing schemas for which generation fails, ensuring that our detection strategy is targeted and relevant. The failure of a schema containing `oneOf` can be effectively detected depending on the structure of the sub-schemas within the `oneOf` operator and their interactions. To detect these interactions that lead to failure, we use the function *oneOfCond*, which returns $\mathcal{T}$ if certain conditions are met by the sub-schemas within `oneOf`.

$$\frac{S = \{\, \texttt{oneOf} : [\, S_1, \ldots, S_n \,], \vec{K} \,\} \qquad r = \mathit{oneOfCond}([\, S_1, \ldots, S_n \,])}{S \ \rightarrow r} \ (\texttt{oneOf-fail})$$

Before delving into the explanation of how the function *oneOfCond* works, it is important to note that during canonicalization, the `oneOf` operator is translated into its logical equivalent: a disjunction of conjuncts, where each conjunct contains negation. In these conjuncts, all sub-schemas within `oneOf` are negated except for one. The schema structures

we aim to cover here are those describing objects and arrays, regardless of whether the `type` keyword is present or not.

Table 7.6 presents the schema structures we are analyzing, the schemas produced after applying the canonicalization process to these patterns, and their corresponding negations. In this context, `Obj` represents schemas describing objects that include at least one object-related keyword, represented by $\{\texttt{type:object}, (\texttt{objKW})^+\}$. Similarly, `Arr` refers to schemas describing arrays, represented by $\{\texttt{type:array}, (\texttt{arrKW})^+\}$. The function `co` is used to denote schemas that capture instances not covered by its schema parameter. For example, `co(Obj)` captures instances that are not valid against the schema `Obj`. Moreover, `otherTypes` captures all JSON schema types except the one appearing in the disjunction. For instance, in `otherTypes` $\vee$ `co(Obj)`, `otherTypes` is equivalent to the disjunction of all types except objects.

Schema structure	Canonicalization	Negation
`Obj`	`Obj`	`otherTypes` $\vee$ `co(Obj)`
`Arr`	`Arr`	`otherTypes` $\vee$ `co(Arr)`
`{not:Obj}`	`otherTypes` $\vee$ `co(Obj)`	`Obj`
`{not:Arr}`	`otherTypes` $\vee$ `co(Arr)`	`Arr`
$\{(\texttt{objKW}\mid\texttt{arrKW})^+\}$	`otherTypes` $\vee$ `Obj` $\vee$ `Arr`	`co(Obj)` $\vee$ `co(Arr)`
$\{\texttt{not:}\{(\texttt{objKW}\mid\texttt{arrKW})^+\}\}$	`co(Obj)` $\vee$ `co(Arr)`	`otherTypes` $\vee$ `Obj` $\vee$ `Arr`

Table 7.6: Schema structures with corresponding canonicalized forms and negations

Since each conjunct resulting from the canonicalization of the `oneOf` operator contains exactly one positive schema, with all the others negated, we will analyze the outcome of the conjunction of the canonicalized version of each schema structures from the previous table with the negations of each schema structures.

The matrix $\texttt{M}^*$ represents the outcome of these conjunctions, where the rows represent the canonicalized schemas and the columns their negations. To simplify the notation, we substituted $\{(\texttt{objKW}\mid\texttt{arrKW})^+$ with $\{(\texttt{KW})^+\}$. A value 0 in the matrix indicates that the conjunction of the two schemas will not proceed, meaning that generation will fail. Conversely, a value of 1 signifies that the conjunction is successful and the generation will proceed.

$$\texttt{M}^* = \begin{bmatrix} & \texttt{Obj} & \texttt{Arr} & \texttt{\{not:Obj\}} & \texttt{\{not:Arr\}} & \{(\texttt{KW})^+\} & \{\texttt{not:}\{(\texttt{KW})^+\}\} \\ \texttt{Obj} & 0 & 1 & 1 & 1 & 0 & 1 \\ \texttt{Arr} & 1 & 0 & 1 & 1 & 0 & 1 \\ \texttt{\{not:Obj\}} & 1 & 1 & 0 & 1 & 0 & 1 \\ \texttt{\{not:Arr\}} & 1 & 1 & 1 & 0 & 0 & 1 \\ \{(\texttt{KW})^+\} & 1 & 1 & 1 & 1 & 0 & 1 \\ \{\texttt{not:}\{(\texttt{KW})^+\}\} & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

All the values on the diagonal are equal to 0. This is because no schema structures can be successfully combined with its own negation. For instance, the schema structures `Obj`

cannot be successfully merged with its negation, which is `otherTypes` $\vee$ `co(Obj)`. The conjunction of `Obj` with each of the other types results in the schema $false$, and the only remaining conjunction is `Obj` and `co(Obj)`, which cannot be merged.

Similarly, consider the schema structures `{not:{(objKW|arrKW)`$^+$`}}`. The canonicalized version of this pattern, when conjuncted with the negations of all other patterns, cannot be successfully combined. This is because it represents the complement of objects and arrays, and these complements cannot be merged with any negation of the other schema structures.

In contrast, an example of a successful conjunction with a value of 1 is the canonicalized version of `{not:Obj}` combined with the negation of `{not:{(objKW|arrKW)`$^+$`}}`. Generation will be successful for this conjunction because `otherTypes` in the first schema structures includes all types except `object`, while the negation in the second schema structures includes all types except `object` and `array`. Thus, their combination covers all necessary types for successful generation.

Additionally, since all schemas are negated except one, we must also analyze the interactions between the negated versions of the schema structures themselves. This involves evaluating how the conjunctions of the negated schema structures interact with each other. By examining these interactions, we can gain a more comprehensive understanding of how combining negations impacts the overall generation process. The results of this analysis are presented in the matrix $\mathtt{M}^-$, which complements the results from $\mathtt{M}^+$ by offering insights into the effects of negating and combining different schema structures.

$\mathtt{M}^- =$

	Obj	Arr	{not:Obj}	{not:Arr}	{(KW)$^+$}	{not:{(KW)$^+$}}
Obj	1	1	0	1	0	1
Arr	1	1	1	0	0	1
{not:Obj}	0	1	1	1	0	1
{not:Arr}	1	0	1	1	0	1
{(KW)$^+$}	0	0	0	0	0	0
{not:{(KW)$^+$}}	1	1	1	1	0	1

Now that we have identified the interactions between all the schema structures, including the canonicalized versions with their negations and the interactions among the negated schema structures themselves, we will visualize these relationships by constructing graphs. We first use the matrix $\mathtt{M}^+$ to build graphs, where the nodes represent schema structures and an edge between two nodes corresponds to a value of 1 in the matrix, indicating the possibility of merging those patterns. Afterward, we will use the matrix $\mathtt{M}^-$ to complete the graphs by adding edges between the negated schema structures.

Figure 7.7 represents the graphs that illustrate these interactions. In these graphs, we use abbreviations to represent the different schema structures: `O` stands for `{type:object, (objKW)`$^+$`}`, `A` for `{type:array, (arrKW)`$^+$`}`, `KW` for `{(objKW|arrKW)`$^+$`}`, `NO` for the negation `{not:{type:object, (objKW)`$^+$`}}`, `NA` for `{not:{type:array, (arrKW)`$^+$`}}`,
and `NKW` for `{not:{(objKW|arrKW)`$^+$`}}`. In this context, the superscript $^-$ is used to represent the negation form of a schema, while its absence represents its canonical form.

Now that we have introduced all the necessary material, we return to the function *oneOfCond*, which evaluates whether the schemas within the `oneOf` operator could potentially cause generation failures. This assessment is performed by analyzing the graphs presented in Figure 7.7. Specifically, the function checks whether, within the graphs containing the canonical forms of the schema structures corresponding to the schemas within `oneOf` (excluding graphs that contain the canonical forms of the schema structures not present in `oneOf`), there exists a subgraph where all the schema structures form a fully connected component. For a fully connected subgraph to be valid, it must include at least one schema in its canonical form, with the other schemas in their negated forms. If such a fully connected subgraph exists, merging these schemas is feasible, and the generation will proceed. If no fully connected subgraph meeting these criteria can be found, the function returns $\mathcal{T}$, indicating that generation will fail.

Example 20 *Consider the following schema, the schema structures featured within the* `oneOf` *operator are* $\{(objKW|arrKW)^+\}$ *(KW) for the first and last schemas,* $\{$`type:object,`$(objKW)^+\}$ *(O) for the second schema, and* $\{$`not:{type:object,`$(objKW)^+\}\}$ *(NO) for the third schema.*

Here, the function oneOfCond only checks the graphs containing the canonical forms KW, O, and NO, which correspond to the graphs of Figures 7.7e, 7.7a and 7.7c. Since the three nodes representing these schema structures are not connected in any of these graphs, oneOfCond returns $\mathcal{T}$, indicating that generation will fail.

It's important to note that the simultaneous appearance of these schema structures will always lead to generation failures, even in the presence of other schema structures.

```
{ "oneOf": [
    { "required": ["a"] },
    { "type": "object", "required": ["a","b"] },
    { "not": { "type": "object", "required": ["c"] } },
    { "required": ["b","d"] }
  ]
}
```

Example 21 *Consider the following schema that only contains the schema structures KW. The function oneOfCond checks the graph shown in Figure 7.7e. Since the nodes labeled KW and* KW^- *are not connected, the function also returns $\mathcal{T}$.*

```
{ "oneOf" : [
    { "properties": { "a": true, "b": true }, "minProperties": 2 },
    { "required": ["a", "b"] }
  ]
}
```

Example 22 *Consider the following schema, which contains the schema structures O, NKW and NO. Since the graph of figure 7.7a contains a complete subgraph with the three nodes labeled O,* NKW^- *and* NO^-*, the function oneOfCond returns $\mathcal{F}$, indicating that the generation will be successful.*

```
{ "oneOf" : [
    { "type": "object", "properties": { "a": true, "b": true },
      "minProperties" : 2 },
    { "not": { "required" : ["a", "b"] } },
    { "not": { "type": "object", "required" : ["c"] } }
  ]
}
```

Results. Table 7.7 presents the results from applying this new optimization to the GitHub and containment-draft4 collections, which are the only ones containing schemas that encountered generation failures and interruption errors.

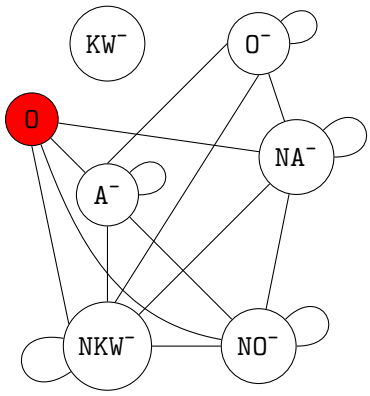
We recall that both the naive hybrid approach and the previous version of the hybrid approach optimized with the earlier strategy had similar execution times, as the earlier strategy had a nearly insignificant impact. However, as shown in the table, this new strategy demonstrates some improvements. The average execution times for both the GitHub and containment-draft4 collections have decreased compared to those recorded previously. In the containment-draft4 collection, the improvement is minimal, which is attributed to the fast processing of schemas in this collection by both the optimistic and pessimistic methods. In contrast, the GitHub collection shows a more significant gain. Specifically, the function *isFail2* successfully detected the failure outcomes for the 3 out of 4 schemas that previously encountered timeout errors. Additionally, the strategy contributed to reduced execution times for schemas that were processed correctly but had previously resulted in generation failures.

Moreover, the new optimization strategy enhances our ability to detect schemas that experienced timeout errors when processed with the optimistic approach. Out of the 22 schemas that previously encountered timeout errors, this verification process identified 19 as failures. The remaining 3 schemas include 1 that experienced a timeout during the reference expansion phase and could not be detected, and 2 that exhibited complex schema structures not covered by the *isFail2* function.

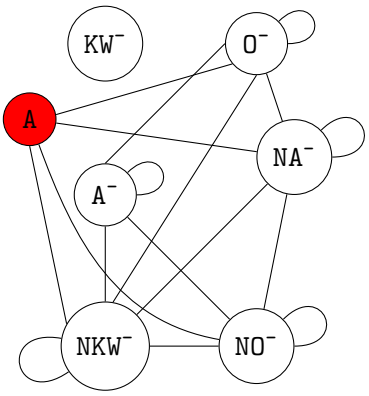
collection	Success	Interrupt.	GenFail	Logical Error	Time (ms) med./avg.
Git	99.66%	0.28%	0.06%	0%	1/670
CC4	100%	0%	0%	0%	1/6.21

Table 7.7: optimized hybrid approach: correctness results and execution times

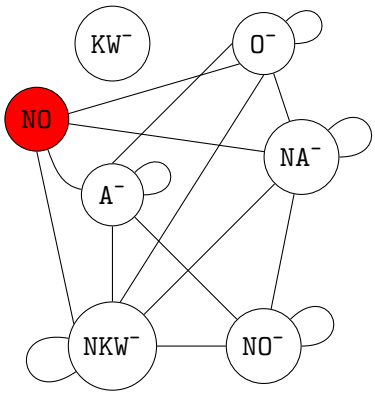
The pre-canonicalization switch optimization introduces notable improvements over the previous strategy by addressing some of the key limitations of the post-canonicalization fallback. Unlike its predecessor, this new approach tackles both generation failures and interruption errors, particularly timeout issues that arise during the canonicalization phase.



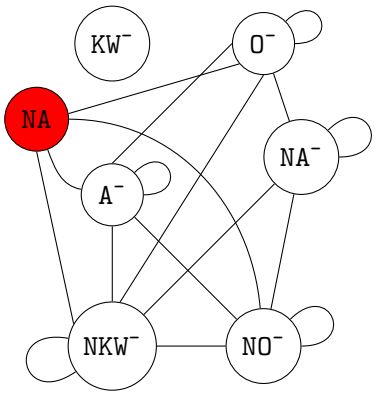
(a) `{type:object,(objKW)+}`



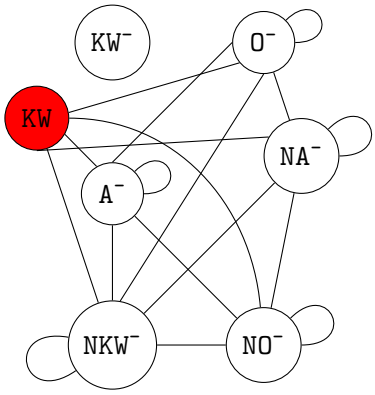
(b) `{type:array,(arrKW)+}`



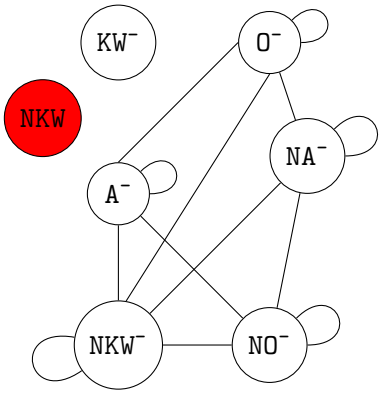
(c) `{not:{type:object,(objKW)+}}`



(d) `{not:{type:array,(arrKW)+}}`



(e) `{(objKW|arrKW)+}}`



(f) `{not:{(objKW|arrKW)+}}`

Figure 7.7: Schema structures interactions

By detecting failure-prone schemas before canonicalization begins, the function *isFail2* has successfully reduced execution times and improved overall efficiency in the hybrid approach.

While some schemas remain unaddressed by this strategy, particularly those that encounter timeouts during reference expansion and complex patterns not detected by the static analysis, the overall impact of the pre-canonicalization switch is promising. It offers a more robust solution for optimizing JSON Schema data generation, especially in scenarios where the generation process is resource-intensive and prone to failure.

To conclude, this strategy is a valuable addition to help optimise the hybrid approach, as it shows significant improvements in performance and error handling. Future refinements, targeting even more complex schema structures could further extend its effectiveness.

7.4 Conclusion

The development and evaluation of the hybrid approach for JSON Schema data generation presented in this chapter offer significant advancements over existing strategies. By combining the pessimistic (pessimistic) approach with the faster, more flexible optimistic approach, the hybrid methodology has demonstrated a balanced solution capable of addressing a wide variety of schemas, while maintaining both correctness and efficiency.

Through the optimizations proposed, we have seen incremental improvements in both performance and error handling. The naive hybrid approach laid the groundwork, showing that a straightforward combination of the two methods could yield comparable success rates and execution times to the optimistic approach, particularly for simpler schemas. However, limitations in handling more complex or failure-prone schemas highlighted the need for further refinement.

The introduction of the post-canonicalization fallback optimization addressed specific issues related to generation failures but struggled to resolve interruption errors, particularly those occurring during the canonicalization phase. While this strategy reduced some computational overhead, its impact was limited, especially for schema collections where failure detection occurs immediately.

The pre-canonicalization switch optimization, on the other hand, provided a more substantial improvement by proactively addressing both generation failures and interruption errors. By implementing failure detection earlier in the process, we successfully reduced execution times and improved the efficiency of the hybrid approach, especially in scenarios where resource-intensive operations previously led to timeouts. Although some schemas remain beyond the reach of this strategy, it represents a more robust and adaptable solution for complex schema structures.

Overall, the hybrid approach, particularly with the integration of the pre-canonicalization switch, shows significant promise for optimizing JSON Schema data generation.

Chapter 8

Conclusion

8.1 Summary and Key Findings

This thesis has explored the problem of data generation for JSON Schema, a widely adopted vocabulary used to describe the structure and semantics of JSON data, which is crucial for web APIs and is supported by a large ecosystem of tools. Data generation serves not only as a means to address challenging theoretical problems but also as a practical tool for verifying program consistency and validating functionalities.

In Chapter 3, we introduced the syntax of both JSON and JSON Schema, providing a detailed overview of the latter's features and constraints while focusing on a subset of the Draft 2019 version [69]. To streamline the analysis and avoid the additional complexity introduced by certain operators, we intentionally excluded functionalities such as dynamic referencing.

We then presented in Chapter 4 an existing method for witness generation in JSON Schema, which guarantees both correctness and completeness by generating valid instances for satisfiable schemas while flagging unsatisfiable ones. This method, referred to as the *pessimistic* approach, includes a preprocessing phase that simplifies the schema by eliminating complex constructs like negations and thoroughly prepares object and array types for more efficient instance generation. While this method is capable of producing a single valid instance, adapting it for multiple instances generation is possible but does not confirm that these instances will be distinct, which is one of the method's limitations. Additionally, it does not support the `uniqueItems` constraint in arrays, which enforces the uniqueness of values within arrays.

In Chapter 5, we introduced our novel *optimistic* approach, which emerged from an extensive experimental analysis of both the pessimistic method and other existing tools, such as JSON Generator (DG) [28] and the Containment checker (CC) tool [51, 46], marking a key contribution of this thesis. This analysis was conducted on a diverse set of schema collections, as outlined in Table 4.2, to evaluate the performance of the pessimistic approach against the other data generation methods. The experiments highlighted that while the pessimistic approach guarantees both correctness and completeness, its compu-

tational overhead limits its practicality in certain scenarios. Conversely, JSON Generator (DG), which directly generates instances without a preprocessing phase, belongs to the class of optimistic solutions that prioritize efficiency over completeness. Although this optimistic approach yielded faster results, it struggled with more complex schemas, particularly those involving intricate constraints like negation and conjunction. These findings motivated the design of our own optimistic approach, drawing on the advantages and shortcomings identified during the comparative analysis. The Containment checker (CC) tool [51, 46], although specifically designed for schema inclusion rather than data generation, also influenced how our method handles schema complexities.

Through this thorough evaluation of existing approaches, we conceptualized our optimistic approach, focusing on balancing the correctness of the pessimistic solution with the speed of existing optimistic methods while addressing key challenges such as array uniqueness (`uniqueItems`) and generating multiple distinct instances. This new optimistic method is designed as an alternative to the existing approaches, aiming to maximize efficiency at the expense of completeness. Unlike JSON Generator (DG), this approach relies on a preprocessing phase that is less complex and complete than that of the pessimistic method, adapted from the methodologies described in [46], along with elements drawn from the preprocessing of the pessimistic approach. Notably, we deliberately avoided the complete elimination of negation constructs, which contributes to the incompleteness of this approach.

The optimistic approach advances the state of the art in data generation for JSON Schema by addressing challenges such as generating arrays under the `uniqueItems` constraint. It effectively provides a solution that is not based on randomness and does not involve high computational overhead, facilitating the creation of multiple distinct instances. The comprehensive experimental analysis highlighted in Chapter 6 has confirmed its efficiency across various aspects. In comparison with both the pessimistic method and other optimistic tools like JSON Generator (DG)[28], JSON-Schema-faker (JSF)[1], and JSON-everything (JE)[36], our method demonstrated balanced performance, particularly when considering efficiency and correctness time as combined metrics.

Despite its strengths, the optimistic approach has its limitations. These drawbacks have been discussed throughout the thesis and are summarized as follows:

- One of the limitations is that we considered only a subset of the Draft 2019 version of JSON Schema, omitting a few operators. This omission arises because prior versions of JSON Schema, such as Draft-06 [70], are the most widely used. Consequently, we primarily focused on the operators included in those earlier versions and incorporated only a few from the latest version, specifically excluding features like dynamic referencing.

- Incompleteness: This is the major limitation of our approach. Given its design for efficiency, we opted for a preprocessing phase that performs basic transformations, during which negation is not completely eliminated, particularly when dealing with `object` and `array` types. This decision arises from the fact that complete negation elimination has already been explored in JSON Schema [72], and addressing it in our

context does not provide additional value. The incompleteness is indicated by the method's failure to generate instances, returning GenFail as a signal to users that generation cannot proceed. Despite this limitation, our approach performs well across various real-world schema collections; however, this issue is particularly noticeable in the HW schema collection, where it achieves a low success rate along with a high GenFail rate, as highlighted in Table 6.1.

- String generation: another significant limitation arises from the approach's inconsistency in generating correct string values, as it occasionally produces incorrect results for certain cases. This issue stems from the lack of a mechanism for translating ECMA-262 [39], the language of the regular expressions used by JSON Schema, into the language supported by Brics [60], the library employed for generating string values. The absence of a complete translation mechanism is a deliberate choice, as it does not add significant value. The complete translation is already defined in the context of the pessimistic approach [11, 12], which implements this translation mechansim. Thus, this limitation is more an issue of implementation within the optimistic tool rather than an inherent limitation of the approach.

- Limited support for `array` keywords: as stated in Remark 4, we only consider single `contains` schemas in our generation process, ignoring the cases involving conjunction. Given the existential nature of this operator, it is not sufficient to simply process and merge the schemas; each schema of `contains` requires a specific number of items to satisfy its constraints (indicated by `minContains`, if present). Consequently, developing a complete solution to address this complexity would be more intricate and could potentially decrease the efficiency of our approach; thus, we have decided not to consider it for now. Additionally, the `maxContains` constraint is ignored because its involvement necessitates negating the `contains` schema. Given our partial elimination of negation, the result generally ends up in generation failures when encountering `object` and `array` types.

- Despite the unsatisfiability verification presented in Section 5.3.1, the approach leaves some cases unaddressed that involve complex verifications that cannot be performed statically. The approach does not conduct any verification during the generation process, resulting in scenarios where, for example, the method indicates a generation failure while the processed schema is actually unsatisfiable.

- While the proposed approach for generating JSON arrays in the presence of `uniqueItems` in Section 5.3.5 showed promising results as highlighted by the experimental analysis of Chapter 6, there are a few scenarios that may lead to generation failure. These scenarios occur when the order of visiting the schemas does not allow for generating enough values, and the algorithm that checks for the existence of a perfect matching (or a maximum cardinality matching) has not found one. Additionally, the generation of multiple arrays respecting the uniqueness constraint is not supported, which is essential for handling nested arrays with unique items.

- Other limitations include those that are not inherent to the approach but rather stem from the choices we made. The approach is deterministic and always generates the same set of values for a given schema. One limitation is that it does not allow users to control the instances generated, nor does it enable covering different types of instances by exploring various branches of the schema during each generation. While this issue is interesting, it represents a broader challenge on its own.

- Our generation technique only allows for synthetic generation, meaning that the generated values do not always reflect the context. For example, if a schema requires a property called `"name"`, with its corresponding schema defined as `"type": "string"`, our generator will produce valid string values without considering the semantics. As a result, the technique lacks support for generating real data that is more aligned with the intended context.

Finally, to overcome some of the limitations of the optimistic approach and design a more robust solution, we introduced a hybrid approach in Chapter 7 that combines the strengths of both the pessimistic and optimistic methods. This preliminary design showcased promising results, demonstrating its capability to address a wider variety of schemas while maintaining both correctness and efficiency. The initial implementation of the hybrid approach revealed that a straightforward integration of the two strategies could yield competitive success rates and execution times, especially for simpler schemas. However, it also highlighted the need for further refinement when dealing with more complex schema structures. The introduction of various optimizations, such as the post-canonicalization fallback and the pre-canonicalization switch, aimed to enhance performance and error handling. Overall, the hybrid approach represents a significant advancement in JSON Schema data generation, offering a balanced solution that effectively leverages the benefits of both methodologies.

8.2 Perspecitives for Data Generation for JSON Schema

From the various insights gained throughout our work on JSON Schema, we intend to define some directives for the novel optimistic approach, as well as for the hybrid approach. These directives will serve as a foundation for future enhancements and refinements, addressing the limitations identified in the research while maximizing the effectiveness of the generation process. By synthesizing these findings, further investigations can be guided to advance the state of the art in JSON Schema data generation and ensure that the approaches remain aligned with the evolving needs of users and the broader ecosystem of web APIs.

The novel optimistic approach introduced has shown promising results despite the limitations outlined in the previous section, demonstrating competitive performance by balancing efficiency and correctness. However, to further enhance this approach as a foundational element of the hybrid method, several areas of improvement deserve exploration.

Language Coverage. A key improvement involves expanding the optimistic approach to consider the entire language defined by the latest version of JSON Schema. While this is a relatively minor concern given the widespread adoption of prior versions, aligning with the latest specifications will enhance the approach's robustness and applicability, allowing us to build a solution that is easy to adapt to novel versions of the schema language.

Improvement of Negation Elimination. Although complete elimination of negation is a complex task, certain scenarios allow for feasible simplifications without exhaustive processing.

Example 23 *For instance, consider the following object schema S that solely contains the* `minProperties` *and* `maxProperties` *constraints:*

```
{ "type": "object", "minProperties": min, "maxProperties": max }
```

Eliminating negation in a schema composed of the negation of S, `{ "not": S }`*, is straightforward and does not require complex processing. Handling such cases efficiently will enhance both the completeness and performance of the optimistic approach. The resulting schema after negation elimination is as follows:*

```
{ "anyOf": [
    { "type": "null" }, { "type": "boolean" }, { "type": "integer" },
    { "type": "number" }, { "type": "string" }, { "type": "array" },
    { "type": "object", "minProperties": max+1 },
    { "type": "object", "maxProperties": min-1 }
 ]
}
```

Heuristic Evaluation for UniqueItems. To improve the generation of arrays under the `uniqueItems` constraint, a possible approach would consists of adopting heuristics that evaluate the domain sizes of candidate schemas, ensuring an optimal order of traversal to prevent generation failures. By carefully determining this order, we can increase the chances of producing valid instances. Building on this, the problem can be extended from generating a single array to generating N distinct arrays conforming to the `uniqueItems` constraint. This extension will involve constructing a bipartite graph and identifying N perfect matchings, significantly boosting the robustness and scalability of the approach.

Schema Coverage for Diverse Instances. Improving the generation process by refining the exploration of branches within the schema is essential for generating diverse instance types. This coverage problem is more complex than merely producing instances that differ in field values. A comprehensive analysis of schema exploration strategies will be necessary to ensure a rich variety of generated instances.

Integration of Large Language Models (LLMs). Machine learning, and particularly Large Language Models (LLMs), presents an exciting opportunity to enhance the generation of real-world data. LLMs have shown promising results in understanding context and generating specific data types. By integrating calls to LLMs, we can significantly improve the realism of generated data, aligning it more closely with the intended application context.

User-guided Generation. Giving users more control over the generation process could greatly improve the flexibility and usefulness of the approach. By enabling custom heuristics or user-defined priorities, the generation strategy can be adapted to meet specific needs across different domains, such as automated testing, data synthesis, or API simulation. This customization would make the tool more versatile and better suited to a wide range of use cases.

By addressing these areas of improvement and the other limitations mentioned in the previous section, the optimistic approach can be made a more robust and efficient solution for data generation in the evolving landscape of JSON Schema. As a key component of the hybrid approach, it can also enhance its overall effectiveness. One of the core advantages of the hybrid approach is its ability to leverage the strengths of both optimistic and pessimistic strategies. However, defining heuristics that govern the switching mechanism will be critical for maximizing performance. By developing more sophisticated decision-making processes that evaluate factors such as schema complexity and specific constraints (e.g., negation or `uniqueItems`), a better determination of when to prioritize speed (optimistic) or completeness (pessimistic) can be achieved. Such enhancements would allow the hybrid approach to automatically adapt to a wider range of scenarios, ensuring it remains both efficient and reliable.

Bibliography

[1] Json-schema-faker. Available at https://github.com/json-schema-faker/json-schema-faker.

[2] Json schema test suite, 2020. Available at https://github.com/json-schema-org/JSON-Schema-Test-Suite.

[3] Json schema validator, 2022. Available at https://github.com/networknt/json-schema-validator.

[4] Snowplow Analytics. Iglu central, 2022. Available at https://github.com/snowplow/iglu-central, commit hash 726168e. Retrieved 19 September 2022.

[5] Lyes Attouche. Example generation for json schema. In *37èmes Journées Bases de Données Avancées, BDA*, 2021.

[6] Lyes Attouche, Mohamed Amine Baazizi, and Dario Colazzo. Overview and perspectives for optimistic JSON schema witness generation. In *39èmes Journées Bases de Données Avancées, BDA*, 2023.

[7] Lyes Attouche, Mohamed-Amine Baazizi, and Dario Colazzo. Optimistic data generation for JSON schema. *Trans. Large Scale Data Knowl. Centered Syst.*, pages 119–152, 2024.

[8] Lyes Attouche, Mohamed Amine Baazizi, Dario Colazzo, Yunchen Ding, Michael Fruth, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. Reproduction package: A Test Suite for JSON Schema Containment, 2021. Available on Zenodo at https://zenodo.org/record/5336931#.YshD0XZBxD8 and maintained on GitHub at https://github.com/sdbs-uni-p/json-schema-containment-testsuite.

[9] Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Yunchen Ding, Michael Fruth, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. A test suite for JSON schema containment. In Roman Lukyanenko, Binny M. Samuel, and Arnon Sturm, editors, *Proceedings of the ER Demos and Posters 2021 co-located with 40th International Conference on Conceptual Modeling (ER 2021), St. John's, NL, Canada, October 18-21, 2021*, volume 2958 of *CEUR Workshop Proceedings*, pages 19–24. CEUR-WS.org, 2021.

[10] Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Francesco Falleni, Giorgio Ghelli, Cristiano Landi, Carlo Sartiani, and Stefanie Scherzinger. A tool for JSON schema witness generation. In Yannis Velegrakis, Demetris Zeinalipour-Yazti, Panos K. Chrysanthis, and Francesco Guerra, editors, *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*, pages 694–697. OpenProceedings.org, 2021.

[11] Lyes Attouche, Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. Witness generation for JSON schema. *Proc. VLDB Endow.*, 15(13):4002–4014, 2022.

[12] Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. Witness generation for json schema, 2022.

[13] Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. Validation of modern JSON schema: Formalization and complexity. *Proc. ACM Program. Lang.*, 8(POPL):1451–1481, 2024.

[14] Mohamed-Amine Baazizi, Houssem Ben Lahmar, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Schema inference for massive JSON datasets. In Volker Markl, Salvatore Orlando, Bernhard Mitschang, Periklis Andritsos, Kai-Uwe Sattler, and Sebastian Breß, editors, *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, pages 222–233. OpenProceedings.org, 2017.

[15] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Counting types for massive JSON datasets. In Tiark Rompf and Alexander Alexandrov, editors, *Proceedings of The 16th International Symposium on Database Programming Languages, DBPL 2017, Munich, Germany, September 1, 2017*, pages 9:1–9:12. ACM, 2017.

[16] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Parametric schema inference for massive JSON datasets. *VLDB J.*, 28(4):497–521, 2019.

[17] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Schemas and types for JSON data. In Melanie Herschel, Helena Galhardas, Berthold Reinwald, Irini Fundulaki, Carsten Binnig, and Zoi Kaoudi, editors, *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, pages 437–439. OpenProceedings.org, 2019.

[18] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Schemas and types for JSON data: From theory to practice. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference*

2019, Amsterdam, The Netherlands, June 30 - July 5, 2019, pages 2060–2063. ACM, 2019.

[19] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. A type system for interactive JSON schema inference (extended abstract). In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 101:1–101:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[20] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. An empirical study on the "usage of not" in real-world JSON schema documents. In Aditya K. Ghose, Jennifer Horkoff, Vítor E. Silva Souza, Jeffrey Parsons, and Joerg Evermann, editors, *Conceptual Modeling - 40th International Conference, ER 2021, Virtual Event, October 18-21, 2021, Proceedings*, volume 13011 of *Lecture Notes in Computer Science*, pages 102–112. Springer, 2021.

[21] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. An empirical study on the "usage of not" in real-world JSON schema documents (long version). *CoRR*, abs/2107.08677, 2021.

[22] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. Not elimination and witness generation for JSON schema. *CoRR*, abs/2104.14828, 2021.

[23] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. Negation-closure for JSON schema. *CoRR*, abs/2202.13434, 2022.

[24] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. The usage of negation in real-world JSON schema documents. In Giuseppe Amato, Valentina Bartalesi, Devis Bianchini, Claudio Gennaro, and Riccardo Torlone, editors, *Proceedings of the 30th Italian Symposium on Advanced Database Systems, SEBD 2022, Tirrenia (PI), Italy, June 19-22, 2022*, volume 3194 of *CEUR Workshop Proceedings*, pages 101–108. CEUR-WS.org, 2022.

[25] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. Negation-closure for JSON schema. *Theor. Comput. Sci.*, 955:113823, 2023.

[26] Ignacio Ballesteros, Luis Eduardo Bueso de Barrio, Lars-Ake Fredlund, and Julio Marino. Tool demonstration: Testing json web services using jsongen.

[27] Guillaume Baudart, Martin Hirzel, Kiran Kate, Parikshit Ram, and Avraham Shinnar. LALE: Consistent Automated Machine Learning. In *Proc. KDD Workshop on Automation in Machine Learning (AutoML@KDD)*, volume abs/2007.01977, Jul 2020.

[28] Jim Blackler. Json generator, 2022. Available at https://github.com/jimblackler/jsongenerator. Retrieved 19 September 2022.

[29] Jim Blackler. jsonschemafriend, json schema-based data validator, 2024. Available at https://github.com/jimblackler/jsonschemafriend.

[30] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoc. JSON: data model, query languages and schema specification. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *PODS*, pages 123–135. ACM, 2017.

[31] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, December 2017.

[32] Hugo André Coelho Cardoso and José Carlos Ramalho. Synthetic data generation from JSON schemas. In João Cordeiro, Maria João Pereira, Nuno F. Rodrigues, and Sebastião Pais, editors, *11th Symposium on Languages, Applications and Technologies, SLATE 2022, July 14-15, 2022, Universidade da Beira Interior, Covilhã, Portugal*, volume 104 of *OASIcs*, pages 5:1–5:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[33] Alberto Hernández Chillón, Severino Feliciano Morales, Diego Sevilla, and Jesús García Molina. Exploring the visualization of schemas for aggregate-oriented nosql databases. In Cristina Cabanillas, Sergio España, and Siamak Farshidi, editors, *Proceedings of the ER Forum 2017 and the ER 2017 Demo Track co-located with the 36th International Conference on Conceptual Modelling (ER 2017), Valencia, Spain, - November 6-9, 2017*, volume 1979 of *CEUR Workshop Proceedings*, pages 72–85. CEUR-WS.org, 2017.

[34] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 268–279. ACM, 2000.

[35] Douglas Crockford and Chip Morningstar. Standard ecma-404 the json data interchange syntax, 12 2017.

[36] Greg Dennis. Json-everything, 2022. Available at https://github.com/gregsdennis/json-everything. Retrieved 19 September 2022.

[37] Filipa Alves dos Santos, Hugo André Coelho Cardoso, João da Cunha e Costa, Válter Ferreira Picas Carvalho, and José Carlos Ramalho. Datagen: JSON/XML dataset generator. In Ricardo Queirós, Mário Pinto, Alberto Simões, Filipe Portela, and Maria João Pereira, editors, *10th Symposium on Languages, Applications and Technologies, SLATE 2021, July 1-2, 2021, Vila do Conde/Póvoa de Varzim, Portugal*, volume 94 of *OASIcs*, pages 6:1–6:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[38] Clara Benac Earle, Lars-Åke Fredlund, Ángel Herranz-Nieva, and Julio Mariño. Jsongen: a quickcheck based library for testing JSON web services. In Laura M. Castro and Hans Svensson, editors, *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang, Gothenburg, Sweden, September 5, 2014*, pages 33–41. ACM, 2014.

[39] ECMA Ecma. 262: Ecmascript language specification. *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,*, 1999.

[40] Lars-Åke Fredlund, Clara Benac Earle, Ángel Herranz-Nieva, and Julio Mariño-Carballo. Property-based testing of JSON based web services. In *2014 IEEE International Conference on Web Services, ICWS, 2014, Anchorage, AK, USA, June 27 - July 2, 2014*, pages 704–707. IEEE Computer Society, 2014.

[41] Angelo Augusto Frozza, Ronaldo dos Santos Mello, and Felipe de Souza da Costa. An approach for schema extraction of JSON and extended JSON document collections. In *2018 IEEE International Conference on Information Reuse and Integration, IRI 2018, Salt Lake City, UT, USA, July 6-9, 2018*, pages 356–363. IEEE, 2018.

[42] Michael Fruth, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. Challenges in checking JSON schema containment over evolving real-world schemas. In Georg Grossmann and Sudha Ram, editors, *Advances in Conceptual Modeling - ER 2020 Workshops CMAI, CMLS, CMOMM4FAIR, CoMoNoS, EmpER, Vienna, Austria, November 3-6, 2020, Proceedings*, volume 12584 of *Lecture Notes in Computer Science*, pages 220–230. Springer, 2020.

[43] Francis Galiegue and Kris Zyp. Json schema: interactive and non interactive validation - draft-fge-json-schema-validation-00. Technical report, Internet Engineering Task Force, feb 2013.

[44] Andrew Habib. *Learning to Find Bugs in Programs and their Documentation.* PhD thesis, Technical University of Darmstadt, Germany, 2021.

[45] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. Type safety with json subschema, 2019.

[46] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. Finding data compatibility bugs with JSON subschema checking. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, pages 620–632, 2021.

[47] Peter Hall. On representatives of subsets. *Journal of The London Mathematical Society-second Series*, pages 26–30, 1935.

[48] Zac Hatfield-Dodds. Hypothesis-jsonschema. Available at https://github.com/python-jsonschema/hypothesis-jsonschema and https://pypi.org/project/hypothesis-jsonschema/.

[49] Zac Hatfield-Dodds and Dmitry Dygalo. Deriving semantics-aware fuzzers from web API schemas. In *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022*, pages 345–346. ACM/IEEE, 2022.

[50] John E. Hopcroft and Richard M. Karp. A nˆ5/2 algorithm for maximum matchings in bipartite graphs. In *12th Annual Symposium on Switching and Automata Theory, East Lansing, Michigan, USA, October 13-15, 1971*, pages 122–125. IEEE Computer Society, 1971.

[51] IBM. jsonsubschema, 2022. Available at https://github.com/IBM/jsonsubschema. Retrieved January 2022.

[52] Javier Luis Cánovas Izquierdo and Jordi Cabot. Discovering implicit schemas in JSON data. In Florian Daniel, Peter Dolog, and Qing Li, editors, *Web Engineering - 13th International Conference, ICWE 2013, Aalborg, Denmark, July 8-12, 2013. Proceedings*, volume 7977 of *Lecture Notes in Computer Science*, pages 68–83. Springer, 2013.

[53] Jonah Kagan. Schematic-ipsum. Available at https://github.com/jonahkagan/schematic-ipsum.

[54] Meike Klettke, Uta Störl, and Stefanie Scherzinger. Schema extraction and structural outlier detection for json-based nosql data stores. In Thomas Seidl, Norbert Ritter, Harald Schöning, Kai-Uwe Sattler, Theo Härder, Steffen Friedrich, and Wolfram Wingerath, editors, *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, volume P-241 of *LNI*, pages 425–444. GI, 2015.

[55] Christoph Koehnen. Witness generation for json schema patterns. In *BTW 2023*, pages 1113–1119. Bonn, 2023.

[56] Michel Kromer and futagoza. Peg.js. Available at https://github.com/pegjs.

[57] Kubernetes. Kubernetes json schemas, 2022. Available at https://github.com/instrumenta/kubernetes-json-schema, commit hash b3cf311.

[58] Ivan Veinhardt Latták and Pavel Koupil. A comparative analysis of JSON schema inference algorithms. In Hermann Kaindl, Mike Mannion, and Leszek A. Maciaszek, editors, *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2022, Online Streaming, April 25-26, 2022*, pages 379–386. SCITEPRESS, 2022.

[59] David Maciver and Zac Hatfield-Dodds. Hypothesis: A new approach to property-based testing. *J. Open Source Softw.*, 4(43):1891, 2019.

[60] Anders Møller. dk.brics.automaton – Finite-State Automata and Regular Expressions for Java, 2021. Available at https://www.brics.dk/automaton/. Retrieved 19 September 2022.

[61] Andrei Neculau. Json-schema-random. Available at https://github.com/andreineculau/json-schema-random.

[62] OI4. Industry 4.0 json schema collection. Available at https://github.com/OI4/oi4-oec-service/tree/development/packages/oi4-oec-json-schemas/src/schemas.

[63] Björn Otto and Tobias Kleinert. Fences: Systematic sample generation for JSON schemas using boolean algebra and flow graphs. In Francesca Lonetti, Antonio Guerriero, Mehrdad Saadatmand, Christof J. Budnik, and Jenny Li, editors, *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024), Lisbon, Portugal, April 15-16, 2024*, pages 66–75. ACM, 2024.

[64] Felipe Pezoa, Juan L. Reutter, Fernando Suárez, Martín Ugarte, and Domagoj Vrgoc. Foundations of JSON schema. In *Proc. WWW*, pages 263–273, 2016.

[65] The Washington Post. ans-schema, 2022. Available at https://github.com/washingtonpost/ans-schema, commit hash abdd6c211. Retrieved 19 September 2022.

[66] Diego Sevilla Ruiz, Severino Feliciano Morales, and Jesús García Molina. Inferring versioned schemas from nosql databases and its applications. In Paul Johannesson, Mong-Li Lee, Stephen W. Liddle, Andreas L. Opdahl, and Oscar Pastor López, editors, *Conceptual Modeling - 34th International Conference, ER 2015, Stockholm, Sweden, October 19-22, 2015, Proceedings*, volume 9381 of *Lecture Notes in Computer Science*, pages 467–480. Springer, 2015.

[67] Stefanie Scherzinger, Meike Klettke, and Uta Störl. Managing schema evolution in nosql data stores. In Todd J. Green and Alan Schmitt, editors, *Proceedings of the 14th International Symposium on Database Programming Languages (DBPL 2013), August 30, 2013, Riva del Garda, Trento, Italy*, 2013.

[68] Arash Vahidi. Jdd, 2020. Retrieved 19 September 2022.

[69] A. Wright, H. Andrews, and B. Hutton. JSON Schema validation: A vocabulary for structural validation of json - draft-handrews-json-schema-validation-02. Technical report, Internet Engineering Task Force, sep 2019.

[70] A. Wright, G. Luff, and H. Andrews. Json schema validation: A vocabulary for structural validation of json - draft-wright-json-schema-validation-01. Technical report, Internet Engineering Task Force, apr 2017.

[71] Austin Wright and Henry Andrews. JSON Schema: A Media Type for Describing JSON Documents. Internet-Draft draft-handrews-json-schema-01, Internet Engineering Task Force, March 2018. Work in Progress.

[72] Austin Wright, Henry Andrews, Ben Hutton, and Greg Dennis. Json schema: A media type for describing json documents. Technical report, 2020.

[73] Kris Zyp and Gary Court. A JSON Media Type for Describing the Structure and Meaning of JSON Documents. Internet-Draft draft-zyp-json-schema-03, Internet Engineering Task Force. Work in Progress.

RÉSUMÉ

JSON (JavaScript Object Notation) est un format d'échange de données largement utilisé, notamment dans le cadre des APIs web. Sa simplicité et sa légèreté en font un choix idéal pour l'échange de données entre clients et serveurs. JSON Schema constitue un vocabulaire puissant pour définir la structure, les contraintes et les règles de validation des données JSON, garantissant ainsi l'intégrité et la cohérence des informations tout en respectant les formats et la sémantique spécifiés. Cette synergie entre JSON et JSON Schema renforce la fiabilité de la gestion des données, permettant aux développeurs de créer des applications robustes et faciles à maintenir, tout en mettant en œuvre efficacement une validation solide des données.

La génération de données pour JSON Schema revêt une importance majeure, car elle répond à des défis critiques associés aux langages de schéma, tels que l'inclusion, la satisfiabilité et l'équivalence. Par exemple, l'inclusion est essentielle pour vérifier que les données existantes sont conformes aux schémas mis à jour, préservant ainsi l'intégrité des données lors de l'évolution des systèmes. Au-delà de l'inclusion, la génération de données JSON est cruciale pour tester les APIs web, permettant aux développeurs de simuler divers scénarios et de valider le comportement de leurs applications dans différentes conditions. En produisant des données conformes aux schémas spécifiés, les développeurs peuvent évaluer efficacement la robustesse et la précision de leurs systèmes.

Dans cette thèse, nous explorons divers aspects liés à JSON Schema, en mettant particulièrement l'accent sur les défis de la génération de données. Nous proposons une approche nouvelle qui équilibre efficacité et correction, facilitant la génération de données valides qui respectent les spécifications définies par les schémas. Cette approche vise à améliorer tant la performance que la fiabilité du processus de génération de données.

De plus, nous introduisons une stratégie hybride qui tire parti de l'efficacité de notre technique de génération de données tout en intégrant des éléments d'une méthode établie reconnue pour sa complétude et sa correction dans le processus de génération. En combinant ces deux méthodologies, nous visons à créer une solution de génération plus robuste.

Enfin, nous proposons des orientations pour des recherches futures dans le domaine de la génération de données pour JSON Schema. Nous définissons des directives destinées à améliorer notre technique de génération ainsi que les outils existants de manière générale, tout en explorant des défis supplémentaires liés au langage. En abordant ces problématiques, nous aspirons à faire progresser l'état de l'art dans la génération de données pour JSON Schema.

MOTS CLÉS

JSON, JSON Schema, Génération de données, Données synthétiques, Validation, Satisfiabilité, Inclusion

ABSTRACT

JSON (JavaScript Object Notation) is a widely used data interchange format, especially in the context of web APIs. Its simplicity and lightweight design make it an excellent choice for exchanging data between clients and servers. JSON Schema acts as a powerful vocabulary for defining the structure, constraints, and validation rules for JSON data, ensuring the integrity and consistency of information while adhering to specified formats and semantics. This synergy between JSON and JSON Schema enhances the reliability of data management, empowering developers to build robust and maintainable applications that effectively implement strong data validation.

Data generation for JSON Schema is of great interest, as it addresses critical challenges associated with schema languages, including inclusion, satisfiability, and equivalence. For instance, ensuring inclusion is crucial for confirming that existing data complies with updated schemas, thus safeguarding the integrity of data through system evolutions. Beyond inclusion, generating JSON data is essential for testing web APIs, allowing developers to simulate various scenarios and validate their applications' behavior under diverse conditions. By producing data that conforms to specified schemas, developers can effectively assess the robustness and correctness of their systems.

In this thesis, we explore the multifaceted issues related to JSON Schema, with a particular focus on the challenges of generating compliant data. We propose a novel data generation approach that balances effectiveness and correctness, facilitating the production of valid data adhering to specified schemas. This approach aims to improve both the efficiency and reliability of the data generation process.

Furthermore, we introduce a hybrid strategy that leverages the strengths of our rapid data generation technique while incorporating elements from established methods known for their completeness and correctness. By combining these methodologies, we intend to create a more robust solution that effectively addresses the demands of generating valid JSON data while maintaining high performance.

Finally, we outline directions for future research in the field of data generation for JSON Schema. We define guidelines aimed at steering subsequent investigations and enhancements, exploring additional challenges related to the language. By addressing these issues, we seek to advance the state of the art in data generation for JSON Schema.

KEYWORDS

JSON, JSON Schema, Data Generation, Synthetic Data, Validation, Satisfiability, Inclusion