



Constraints over grammar elements can make test generation easier than ever.

BY DOMINIC STEINHÖFEL AND ANDREAS ZELLER

Language-Based Software Testing

SOFTWARE TESTING CONTINUES to be the number one technique to satisfy safety, security, and privacy requirements. Yet, manual testing is laborious, calling for automated software testing. While automatically *running* software tests is current practice, one still must *write* all these tests. Generating software tests promises to relieve humans of the testing task, leaving the work to “testing bots”—automatic processes tirelessly testing and assessing software around the clock (See Figure 1).

How can we build such testing bots? To create tests for some software, a testing bot first must be able to interact with it—that is, produce inputs, examine

outputs, relate these to the inputs, and thus explore and check its functionality. This task faces two long-standing challenges.

First, there is the *input generation problem*: How can a bot create inputs that reliably cover functionality? Random system input generators (“fuzzers”²⁰) easily detect issues and vulnerabilities in input processing of arbitrary programs. However, creating valid inputs and interactions that reliably reach code beyond input processing is still a challenge for which test generators require human assistance—either through input specifications^{1,5,6,10,12,23} or a comprehensive population of diverse input samples that cover behavior.^{3,17,19,27}

One might assume that large language models (LLMs) might alleviate this problem. Couldn’t a bot learn how to interact with software from examples or documentation? Or what makes a valid input? The problem is that to find bugs, one needs valid but also uncommon inputs—namely, those that trigger less common (and less tested) behavior. LLMs may help find common (and thus valid) inputs—but by construction, they will not find uncommon inputs.

The second challenge is the long-standing *test oracle problem*: How can a bot check the outputs of a system? All test generators and fuzzers assume some oracle that checks for correctness—by default, a generic, implicit oracle detecting illegal or unresponsive states. If, however, a bot is to run specific checks, it needs a test oracle that retrieves and evaluates the relevant information from the out-

» key insights

- For effective software testing, we must generate valid inputs and check outputs for correctness.
- We can address both generating and checking by expressing and solving constraints over grammar elements.
- We can learn grammars and constraints from existing inputs and programs.

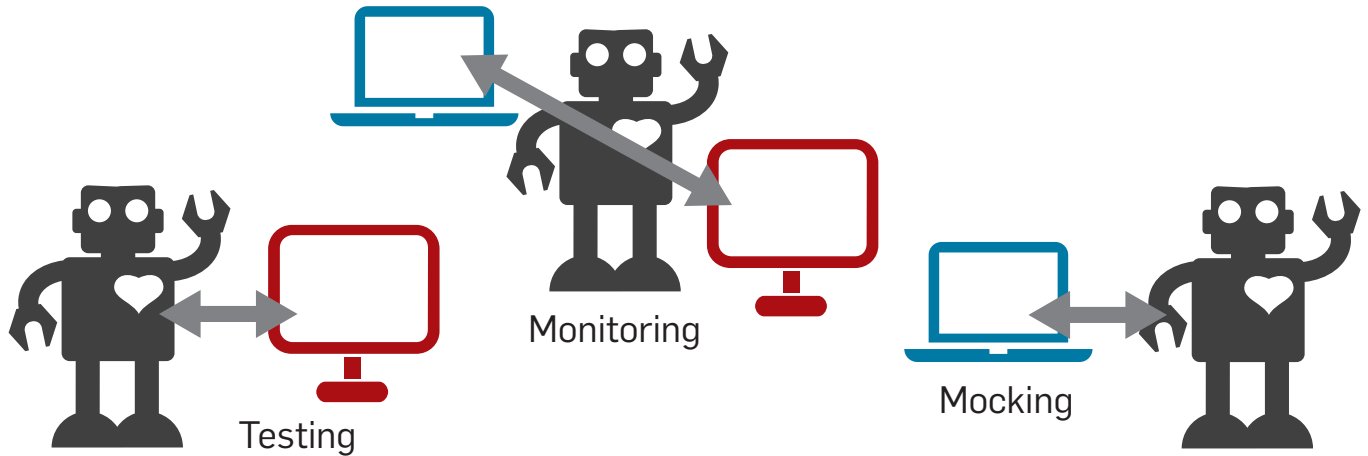


Figure 1. With language specs, software bots can test, monitor, and debug software.

put. Creating oracles manually^{4,6,16,21,24} is nontrivial and time-consuming: “Compared to many aspects of test automation, the problem of automating the test oracle has received significantly less attention and remains comparatively less well-solved.”²

The oracle problem not only affects the result of a computation. If the bot is to interact with the software and produce new inputs in reaction to output properties, we have the oracle problem at every step of the interaction. Now consider the scale and complexity of today’s software systems and their interactions, and you will see that software bots have a long way to go. For this reason, so much time goes into manual testing—and still, software catastrophes like Heartbleed⁷ or log4shell, the “largest vulnerability ever,”¹⁴ keep on haunting us.

To generate tests, we must produce valid inputs and check outputs for correctness.

Specifying Interactions

Why is generating inputs and checking outputs so hard? The Heartbeat extension²⁶ of a TLS server allows clients to check whether a server is still active by sending a 0x1 byte followed by a payload, which the server is expected to include verbatim in its response.

The syntax of such requests is easy to specify. The grammar in Figure 2 identifies the individual elements in the request; we can use it to parse given requests (and thus access and

check its constituents), but also to produce requests (and thus test the server). Using a grammar as a producer ensures *syntactic validity*. This makes test generation far more efficient than generating and mutating random bytes, as valid inputs reliably exercise functionality beyond input processing. (We can still mutate valid inputs to test input processing and handling of invalid inputs.)

To really test a system, we also must check its output. In our Heartbeat example, the server responds with a 0x2 byte, followed by the payload from the request and some padding. To specify the response, we could again use a grammar.

However, we also want to express the response is the result of the request. For this, we can chain request and response to a single I/O gram-

mar¹⁵ characterizing the interaction (Figure 3).

With such an I/O grammar, we can parse an entire client/server interaction to check whether a 0x1 client request gets a proper 0x2 server response. However, we can also produce inputs for the server, expanding `<heartbeat_request>`, and then parse and check the server response. Alternatively, we can mock a server by parsing its input and then expand `<heartbeat_response>` to produce its output. By interleaving multiple input and output sources in a single representation, we obtain a declarative specification of interactions that embeds all the expressiveness of finite-state protocol specifications yet is detailed enough to produce valid inputs and check concrete outputs alike.

Figure 2. An input grammar for the TLS Heartbeat extension.

```
<heartbeat_request> ::= 0x1 <payload_length> <payload> <padding>
<payload_length> ::= <uint16>
<payload> ::= ε | <byte> <payload>
<padding> ::= ε | <byte> <padding>
```

Figure 3. TLS Heartbeat I/O grammar. After a client sends a `<heartbeat_request>`, the server responds with `<heartbeat_response>`.

```
<exchange> ::= <heartbeat_request> <heartbeat_response>
<heartbeat_request> ::= 0x1 <payload_length> <payload> <padding>
<heartbeat_response> ::= 0x2 <payload_length> <payload> <padding>
<payload_length> ::= <uint16>
<payload> ::= ε | <byte> <payload>
<padding> ::= ε | <byte> <padding>
```

Syntax alone, as expressed in our grammar in Figure 3, is not sufficient for testing—neither for generating inputs nor for checking outputs. In our client/server exchange, the `<payload_length>` field holds the length of the `<payload>` that follows as a 16-bit integer. We must satisfy this relation; otherwise, our generated inputs will be invalid. Such properties that cannot be expressed in a context-free grammar are called semantic properties, as they go beyond syntax. How can one specify them?

To allow for a precise specification, we can switch to a different formalism. We could enrich grammars with general-purpose code to produce inputs. Such code, though, would be closely tied to an implementation language and require separate code for producing and parsing strings, which must be kept in sync. Unrestricted grammars can, in principle, specify any computable input property, but we see them as “Turing tar-pits,” in which “everything is possible, but nothing of interest is easy”²²—in our Heartbeat example, one would have to specify integer encodings to start with. In summary, while specifying syntax is well-understood, specifying the semantic properties of inputs and outputs is not—and this makes both generating inputs and checking outputs so difficult.

Specifying syntax of inputs and outputs is well understood. But how do we specify their semantics?

Specifying Syntax and Semantics

In recent work,²⁵ we have presented a means to specify such semantic properties and have shown how they can be used to produce valid inputs as well as to check outputs for correctness. Our Input Specification Language (ISLa) approach represents semantic properties as constraints on top of context-free grammars expressing the relationship between elements in the exchange. Such constraints are like function pre-/postconditions, except that grammar nonterminals take the role of function variables. They can thus make use of arbitrary formalisms, expressing, for example, arithmetic, strings, sets, or temporal logic. To express the relationship be-

tween the `<payload_length>` and `<payload>` fields, we write the ISLa specification

```
uint16(<payload_length>) =
  str.len(<payload>) (1)
```

The function `uint16()` evaluates two bytes as a 16-bit unsigned integer; `str.len()` determines the length of a string. We can also constrain the length further by stating

```
str.len(<payload>) < 16357 (2)
```

which happens to be the maximum length of a `<payload>` element.

Through functions like `uint16()` and `str.len()`, ISLa is effectively unrestricted in expressing input properties while keeping a single declarative specification for input and output properties. Yet, by separating grammars (syntax) and constraints (semantics), we can use tailored approaches to satisfy syntactical and semantic properties—both for producing inputs and checking outputs.

Combining grammars with constraints allows syntax and semantics of inputs and outputs.

Producing Valid Inputs

What can we do with an ISLa specification? To start with, we can use it to produce inputs. ISLa uses a constraint solver to solve the above constraints, instantiating the symbol `<payload_length>` to, say, `0x0005` and `<payload>` elements to, say, `"Hello."` It then places the solutions into the grammar to produce valid inputs. For the grammar in Figure 2, this would then be `<heartbeat_request>` elements such as

```
0x1 0x0005 Hello ...
0x1 0x0004 CACM ...
0x1 0x0000 ...
```

and many more, all syntactically and semantically valid. To express (and solve) constraints, ISLa incorporates all SMT-LIB theories, including integer arithmetics, strings, and regular expressions. On top, ISLa allows quantifying over syntactical elements (“forall,” “exists”) and addressing structural properties (“before,” “on the same level,” ...).

Here are a few examples of ISLa constraints used for various purposes.

In XML and like languages, any opening tag (`<body>`) must be followed by a closing tag (`</body>`) with the same identifier. This ISLa constraint for an XML grammar ensures the opening and closing tags indeed have the same identifier:

```
<xml - tree> . <xml - open - tag>
  . <id> = <xml - tree> .
  <xml - close - tag> . <id> (3)
```

The dot (.) refers to direct children in the derivation tree: `<xml-open-tag>.<id>` is the identifier of the opening tag.

If we would like at least one payload in our exchange to have a length of zero, we can use a quantifier:

```
exists <payload_length> pl in
  <exchange> : uint16(pl) = 0 (4)
```

We do not know whether a length of zero is allowed, but it would be fun to find out. Likewise, one may want to test with very large payloads to check if these trigger a buffer overflow.

And while we already test for vulnerabilities, let us go further and try script injections. This constraint ensures each string ends in a SQL injection:

```
str.suffixof(<string>, "");
DROP TABLE data; -") (5)
```

One could also write grammars generating common injections and test against all of these. Such targeted yet automated testing would allow developers to test their systems thoroughly as never before.

All these constraints can also be combined. For instance, one can have a set of constraints required for validity, add constraints for reaching specific functionality, and, on top, constraints for testing specific scenarios.

If one needs additional functions (say, for a specific domain), ISLa supports adding new predicates, which must come with specific solvers and checkers. For instance, we could create a signature (document, hash) predicate checking whether hash is the appropriate signature for document. With this, ISLa could first generate a document and then cryptographically sign it. Creating a document satisfying a given signature would still require ISLa to enumerate all possible documents until it finds a match-

ing one—which may take billions of years. So, while you can express lots of constraints, there is no guarantee that ISLa can also solve all these constraints. At least not in your lifetime.

Currently, ISLa is well-suited for a limited set of constraints that the constraint solver can handle well. Yet, suppose you want to use ISLa for producing valid C code without undefined behavior. In that case, you will run into limitations regarding the expressiveness of the current ISLa language and issues regarding the efficiency of modern constraint solvers that ISLa uses under the hood. In principle, this is feasible; it “just” requires more engineering work.

Solving constraints automatically produces valid inputs.

Checking Outputs

With ISLa addressing test generation, let us turn to the oracle problem: How can we check whether an output is correct? It turns out that ISLa also is an excellent tool for this. Grammars can express the elements an output is made of; ISLa constraints then capture the properties these elements must satisfy.

Let us return to our Heartbeat exchange in Figure 3, which specifies both request and response. To express that the payload in the response is identical to the payload in the request, we can easily write

```
<heartbeat_request>
.<payload>=<heartbeat_
response>.<payload> (6)
```

and hence, when the `<heartbeat_response>` comes in, compare its payload against the `<heartbeat_request>` payload sent earlier. We see that the nonterminals in the grammar effectively take the roles of variables in pre- and postconditions.

To parse outputs, ISLa can use the grammar—with a bit of help from the constraints, as it needs to know that `<payload_length>` indicates the length of the following `<payload>` field. Also, the I/O grammar needs to differentiate which part of the exchange comes from which of the involved parties. However, ISLa can also check an entire interaction between a client and a server; and even mock a

server by, for a given request, producing a correct response that satisfies the above constraint.

ISLa on its own does not solve the oracle problem completely. It would still be a vast effort to formally specify the correct behavior of an entire SSL/TLS server, with many domain-specific functions and predicates that go beyond the stock arithmetics and functions ISLa has on offer. But at least ISLa provides some means to decompose messages into individual elements and to specify their properties. These capabilities constitute an essential step toward fully formalizing input/output formats, incorporating all abstraction levels, from documents received and messages exchanged to the individual bits and bytes.

Constraints can also be used to check outputs for correctness.

Learning Input Languages

To cash in the benefits discussed, one only needs one thing—complete formal specifications of the input and output languages. Will developers be able to provide these? Possibly, if they can use a specification language that is sufficiently expressive and useful. However, we can undoubtedly ease their task by inferring approximate specifications from existing data and code.

In recent work,^{11,13} we have shown how to automatically extract input grammars from existing programs. Our MIMID prototype takes a program and a set of diverse sample inputs (which we can even generate from scratch using parser-directed fuzzing¹⁸) and tracks where and how individual bytes in the input are processed. Bytes that follow the same path form tokens; subroutines induce hierarchies. After refining grammars through active learning, MIMID thus produces a context-free grammar that accurately represents the input language of a program. As MIMID grammars reuse code identifiers and reflect code structure, they are concise, structured, and well-readable. Other sources of grammars include semi-formal specifications such as RFCs; XML schemas also are formal descriptions of input structure.

With a grammar (given or mined), we can decompose given inputs and outputs into their elements. Suppose we observe some Heartbeat interactions between a client and a server:

```
<client_request> =
  0x1 0x0005 Hello ...
<server_response> =
  0x2 0x0005 Hello ...
```

The grammar tells us that `0x0005` is the `<payload_length>`, and that `Hello` is the `<payload>` itself.

Having individual elements and their values allows us to infer constraints over these very elements. Doing so is like inferring abstractions that match given sets of observed values, a problem known as program synthesis. We have experimented with a traditional synthesis technique called *dynamic invariants*,⁹ which instantiates a catalog of patterns with all given variables and retains those patterns that hold for all values of these variables.

In our example, a pattern such as

```
str.len($1)=uint16($2) (7)
```

would be instantiated with

```
$1=<heartbeat_request>,
  <payload_length>,
  <payload>,... (8)
```

and `$2` likewise. Now, an instantiation like

```
str.len(<heartbeat_request>)=
  uint16(<payload_length>) (9)
```

would not match any of the observed interactions and thus would be discarded. However, the instantiation

```
str.len(<payload>)=
  uint16(<payload_length>)(10)
```

would apply to all observed interactions and could thus be extracted as a constraint for the grammar in Figure 3. We implemented the sketched procedure in our ISLearn prototype.²⁵

Learning constraints this way works surprisingly well. Our AVICENNA prototype⁸ further refines ISLearn to generate better input abstractions faster. Its main application is debugging: We aim to find an explanation for some software behavior, for example, a crash. AVICENNA uses techniques from explainable AI to restrict the search space, uses both

benign and crashing inputs to eliminate coincidental properties, and gradually refines candidate explanations by considering example inputs generated for them by ISLa. In our case study—based on actual bugs in real programs—AVICENNA produced concise diagnoses matching the precision of human experts. Until now, the only widely adapted automated debugging technique is the simplification of failure-inducing inputs.²⁸ We believe tools like AVICENNA that automatically explain failure circumstances will soon contribute to reducing the burden of software maintenance by joining input minimizers in automated testing and debugging toolchains. And if we have tools like AVICENNA target acceptance—that is, we determine the conditions under which an input is accepted by the program under test—we obtain the exact constraints that define valid inputs. Hence, given a set of seed inputs and a program that processes them, we can infer both the input grammar and the input constraints—and thus obtain a language for testing, checking, and monitoring inputs.

*Grammars and constraints
can be learned from inputs
and programs.*

Outlook

The future of test generation will be language oriented. Without knowledge about the input language of a program, we cannot efficiently reach deep program functionality, overcoming coverage plateaus that today's mutation-based fuzzers struggle with. Furthermore, without knowing a program's output language and the desired relation between inputs and outputs, we cannot precisely detect logic errors.

Formal input/output language models can also be used to mock functionality, generate inputs for an outcome of your choice, repair inputs, perform semantics-preserving mutations, and monitor services. In our vision, such language models will be generated by a collaboration of human experts, translators from schemas in other formats, grammar and constraint miners, and AI-based techniques inferring language descrip-

tions from semi-formal documents such as RFCs. At that point, the bots begin their work. They continuously test systems with powerful input generators and strong oracles, report suspicious system inputs or outputs, isolate failure circumstances, and suggest fixes. All these little helpers will free valuable resources for companies and developers, enabling them to focus on implementing new products and features.

ISLa is open source: Install it by running `pip install isla-solver` (requires Python). If you want to toy with it, we recommend following our interactive tutorial for inspiration.^a

Acknowledgment

This work is funded by the European Union (ERC Advanced Grant 101093186 – Semantics of Software Systems (S3)). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. **C**


a <https://www.fuzzingbook.org/beta/html/FuzzingWithConstraints.html>

References

- Aschermann, C. et al. NAUTILUS: Fishing for deep bugs with grammars. In *Proceedings of 2019 Network and Distributed System Security Symp.*; <https://bit.ly/3uRisTx>
- Barr, E.T. et al. The oracle problem in software testing: A survey. *IEEE Trans Softw Eng.* 41, 5 (2014), 507–525; 10.1109/TSE.2014.2372785
- Böhme, M., Pham, V.T., and Roychoudhury, A. Coverage-based greybox fuzzing as Markov Chain. In *Proceedings of the ACM SIGSAC Conf. Computer and Communications Security* (Vienna, Austria, 2016). ACM, New York, NY, USA, 1032–1043; 10.1145/2976749.2978428
- Booch, G. *The Unified Modeling Language User Guide*. Addison Wesley, 2005.
- Briand, L.C. and Labiche, Y. A UML-based approach to system testing. *Intern. Conf. on the Unified Modeling Language, Modeling Languages, Concepts, and Tools*. Springer, 2001, 194–208; 10.5555/647245.719446
- Claessen, K. and Hughes, J. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the 5th ACM SIGPLAN Intern. Conf. Functional Programming*. ACM, New York, NY, USA, 2000, 268–279; 10.1145/351240.351266
- Durumeric, Z. et al. The matter of Heartbleed. In *Proceedings of the 2014 Conf. Internet Measurement*. (Vancouver, BC, Canada), ACM, New York, NY, USA, 475–488; 10.1145/2663716.2663755
- Eberlein, M. et al. Semantic debugging. In *Proceedings of the 31st ACM Joint European Softw. Eng. Conf. and Symp. Foundations of Softw. Eng. (San Francisco, CA, USA, 2023)*, K. Blincoe and P. Tonella, eds. ACM, New York, NY; preprint <https://publications.cispa.saarland/3988/>
- Ernst, M.D. et al. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.* 27, 2 (Feb. 2001), 99–123; 10.1109/32.908957
- Godefroid, P., Kiezun, A., and Levin, M.Y. Grammar-based whitebox fuzzing. In *Proceedings of ACM SIGPLAN Conf. on Programming Language Design and Implementation* (Tucson, AZ, USA, 2008). ACM, 206–215; 10.1145/1375581.1375607
- Gopinath, R., Mathis, B., and Zeller, A. Mining input grammars from dynamic control flow. In *Proceedings of the 2020 Joint Meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Symp. Foundations of Softw. Eng.*; <https://publications.cispa.saarland/3101/>
- Holler, C., Herzig, K., and Zeller, A. Fuzzing with code fragments. In *Proceedings of the 2012 USENIX Security Symp.* USENIX Assoc., Bellevue, WA, 38–38; <https://bit.ly/41oj8Mv>
- Hörschele, M. and Zeller, A. Mining input grammars from dynamic taints. In *Proceedings of the IEEE/ACM Intern. Conf. Automated Softw. Eng.* (Singapore, 2016), 720–725; 10.1145/2970276.2970321
- Hunter, T. and de Vynck, G. "most serious security breach ever" is unfolding right now. *WSJ* (Dec. 20, 2021); <https://www.washingtonpost.com/technology/2021/12/20/log4j-hack-vulnerability-java/>
- Jones, B., Harman, M., and Danicic, S. *Automated Construction of Input and Output Grammars*. Technical Report. University of North London, 1999; <https://bit.ly/3TqEcQv>
- Kent, S. Model driven engineering. In *Proceedings of the 2002 Intern. Conf. Integrated Formal Methods*. Springer, 286–298; 10.5555/647983.743552
- LibFuzzer; <https://lvm.org/docs/LibFuzzer.html>
- Mathis, B. et al. Parser-directed fuzzing. In *Proceedings of the ACM SIGPLAN Conf. Programming Language Design and Implementation* (Phoenix, AZ, USA, 2019). ACM, 548–560; 10.1145/3314221.3314651
- McMinn, P. Search-based software testing: Past, present and future. In *Proceedings of the IEEE 2011 4th Intern. Conf. Softw. Testing, Verification and Validation Workshops*. IEEE, 153–163.
- Miller, B.P., Fredriksen, L., and So, B. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44; 10.1145/96267.96279
- Mussa, M., Ouchani, S., Al Sammane, W., and Hamou-Lhadj, A. A survey of model-driven testing techniques. In *Proceedings of the 2009 9th Intern. Conf. Quality Softw.*, 167–172; 10.1109/QSIC.2009.30
- Perlis, A.J. Epigrams on programming. *ACM SIGPLAN Notices* 17, 9 (1982), 7–13; 10.1145/947955.1083808
- Pham, V.-T. et al. Smart greybox fuzzing. *IEEE Trans. Softw. Eng.* 47, 9 (2019), 1980–1997; 10.1109/TSE.2019.2941681
- Rosenblum, D.S. A practical approach to programming with assertions. *IEEE Trans. Softw. Eng.* 21, 1 (1995), 19–31; 10.1109/32.341844
- Steinhöfel, D. and Zeller, A. Input invariants. In *Proceedings of the 30th ACM Joint European Softw. Engineering Conf. and Symp. the Foundations of Softw. Eng.* (Singapore, 2022). ACM, New York, NY, USA, 2022, 583–594; 10.1145/3540250.3549139
- Williams, M., Tüxen, M., and Robin Seggelmann, R. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. *RFC 6520 IETF*, 2012; <https://datatracker.ietf.org/doc/rfc6520/>
- Zatowski, M. *American fuzzy lop*; <https://lcamtuf.coredump.cx/afl/>
- Zeller, A. and Hildebrandt, R. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.* 28, 2 (Feb. 2002), 183–200; 10.1109/32.988498

Dominic Steinhöfel is a researcher of computer science at CISPA Helmholtz Center for Information Security, in Saarbrücken, Germany.

Andreas Zeller is on faculty at CISPA Helmholtz Center for Information Security and a professor of software engineering at Saarland University, Germany.

 This work is licensed under a <http://creativecommons.org/licenses/by/4.0/>