Java2CSP: A tool for compiling Java programs into constraints for automated fault localization

Vlad Andrei Dumitru[1] and **Franz Wotawa**[2]
[1] Graz University of Technology
e-mail: dumitru@tugraz.at
[2]Institute for Software Technology, Graz University of Technology
e-mail: wotawa@ist.tugraz.at

Abstract

Fault detection and fault localization in programs is considered a hard task where especially in case of fault localization heavy manual work is still needed. In this paper, we contribute to research activities of automated program debugging and present a tool that allows compiling a subset of Java into an equivalent constraint representation. The constraint representation captures the current behavior of the program and can be directly used for fault localization relying on consistency-based diagnosis. A web interface of the tool is available for the public. Besides discussing the underlying foundations, we also present the tool's architecture and internals, and how to carry out a debugging session. Furthermore, we released the underlying program as open source for further supporting research in model-based debugging.

1 Introduction

Faults in software have always lead to trouble. Blogs[1] mentioned research indicating that about $2.8 trillion in the US alone because of software not fulfilling quality criteria. Software bugs also have harmed people, e.g., a software bug in Therac-25 caused patients to receive a massive overdoses of radiation during medical treatment[2]. Hence, detecting and removing bugs from software is of utmost importance. It is also worth noting that bugs not only cause safety issues arising or increased costs, but also directly compromise the security of systems allowing attackers to access private information or cause other damages.

In this paper, we focus in particular on locating bugs in software. There has been a lot of approaches focusing on automated (or semi-automated) debugging. Most recently Wong and colleagues [1] summarized different approaches that have been published. Interestingly about 20% of research papers have utilized model-based reasoning for this purpose. We follow this line of debugging research in this paper and discuss an implementation of a debugging tool that is available for the public. The main purpose of the tool is on supporting research in this particular domain and to provide a running system where others can carry out experiments and provide extensions.

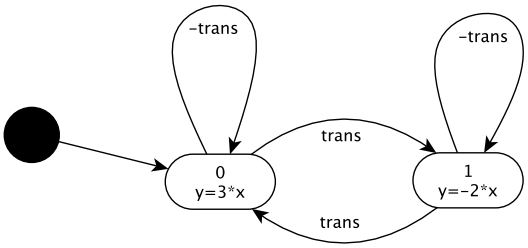


Figure 1: A hybrid automaton representing the specification of our running example.

Specifically, we are going to discuss the tool Java2CSP allowing to convert Java programs into the SMT-LIB [3] constraint representation that can be used by constraint solvers like Z3[4] [2]. The underlying ideas behind have alread been published. Wotawa and colleagues [3] provided the basic foundations behind such a conversion taking ideas from [4] and [5]. However, the idea behind making use of constraints in the context of program development is even older, e.g., see [6] where the authors make use of constraint solving for test case generation. The Java2CSP tool provides a constraint representation where the basic concepts of model-based diagnosis [7; 8] are represented. In particular, the representations makes us of the ab predicate allowing to state the health of a component and in case of programs of a statement. It is worth noting, that the constraint representation is not restricted to debugging, but can also be used for generating test cases when setting the health states of components to work as expected.

In the following, we briefly introduce the underlying debugging approach making use of a small example program. In Figure 1 we outline the underlying specification of the example where we have a system comprising two states `0` and`1`. In both states we have a function mapping an input, i.e., `x`, to an output `y`, which are different. A variable `trans` is used to move from one state to another.

In Figure 2 we depict the faulty implementation where in Line 13 we have a bug. Such bugs may occur because

[1]See `https://www.softwaretestingnews.co.uk/the-real-cost-of-software-bugs/`

[2]See `https://en.wikipedia.org/wiki/List_of_software_bugs`

[3]See `http://smtlib.cs.uiowa.edu`

[4]See `https://github.com/Z3Prover/z3`

```
1.  public class Test {
2.     public void foo (int x, int state, int trans) {
3.       int y;
4.       int new_state;
5.       if (state == 0) {
6.         y = 3*x;
7.         if (trans == 1) {
8.           new_state = 1;
9.         }
10.      } else {
11.        y = -2*x;
12.        if (trans == 1) {
13.          new_state = 1; // Bug!  Should be:  new_state = 0
14.        }
15.      }
16.    }
17. }
```

Figure 2: A small example program where method `foo` implements a simple deterministic finite automaton.

of copy actions. In the particular example, a programmer may copy the code from lines 7–9 but forgot to make the appropriate changes. Such a bug can be revealed using the following inputs and expected outputs:

$$(\texttt{x}, \texttt{state}, \texttt{trans}, \texttt{y}, \texttt{new\_state},) = (1, 1, 1, -2, 0)$$

When calling the function `foo` in the implemented class `Test` using the given inputs, we obviously receive a correct value for `y` but an incorrect one for `new_state` and we are interested in finding the root cause. In model-based diagnosis and also model-based debugging, we make use of the ab predicate to state the health state of a component, in this case a statement explicitly. In case of debugging each statement is converted into a constraint. E.g. the statement `11. y = -2*x;` is mapped into an equation looking similar to $ab_{11} \vee (y = -2 * x)$[5]. In this equation, we either assume that the statement `11` is not working as expected or the constraint corresponding to the statement has to be applied.

Diagnosis is performed via setting the ab's accordingly such that the constraints representing statements are not in contradiction with the information provided by the given test case. In the case of `foo` and the given test case, only Line 12 and Line 13 can explain the faulty behavior.

We organize the paper as follows. In order to be self-contained we first briefly discuss the underlying basic definitions of our tool. Afterwards, we discuss the Java2CSP architecture, functionality, the user interface, the capabilities and limitations. Finally, we summarize the paper.

2 Basic foundations

In the following we outline the basic definitions and concepts behind converting programs into a semantically equivalent constraint representation. For more details, we refer the interested reader to Wotawa et al. [3]. The conversion is done in 3 steps. In the first step we eliminate all loops and recursive function calls. For loops this can be done via converting them into nested if-then-else statements. For example, a loop `while` C `{` B `}`, is represented as

`if` C `{`
  B
  `if` C `{`
    B
    if C {
      ...

The number of nested if-then-else statements has to be chosen assuring that the maximum number of iterations can be handled. For the case of debugging or test case generation, this can be achieved. In the former case we know the passing and failing tests and, therefore, also the number of iterations. We may set the nesting depth to a larger value. Moreover, we may also add an else-part after the last if-part, where we raise an error. This allows us to explicitly state the limitations of this conversion step. For recursive calls we apply a similar method.

In the second step, we convert the loop-free program into its static single assignment form (SSA). The SSA guarantees that every variable is defined only once. Moreover, the conversion only requires to be executed for assignment and if-then-else statements. The SSA conversion makes use of indices applied to variables. For every variable we introduce an index starting with 0. Every time a variable is (re-) defined the index is increased during conversion. For example, consider the following program fragment:

```
1.    x = y + 1;
2.    x = x * 2;
```

The SSA form of this program is:

```
1.    x1 = y0 + 1;
2.    x2 = x1 * 2;
```

For if-then-else statements the SSA form can also be easily obtained. We only need to convert the then- and the else-branch separately using different indices. Afterwards for every variable that is defined we introduce a function Φ that sets the value depending on the condition C, i.e.:

$$x_i = \Phi(C, x_t, x_e)$$

where t is the last index of variable `x` in the then-branch, and e the last index of the same variable from the else-branch. Φ obviously is defined as follows:

$$\Phi(C, x, y) = \begin{cases} x & \text{if } C \text{ is true} \\ y & \text{otherwise} \end{cases}$$

[5] In the real conversion we have to take care of different statements re-defining variable `new_state`. We are going to discuss this in Section 2 in more detail.

In the final step of the conversion, we only have to map the SSA into a set of constraints. This is simple, because in the SSA we only make use of assignment statements. For example, `x1 = y0 + 1` can be directly converted into an equation $x_1 = y_0 + 1$. For more complex expressions, there are additional steps required, which are similar to conversions into three-address code well-known in compiler construction.

It is worth noting that we do not only convert a statement into a constraints but we also add information required for diagnosis. Hence, we introduce a variable ab_j for every statement j and map the statement S_j into a constraint $ab_j \vee S'_j$, where S'_j is the equation representing statement S_j. Applying the described conversion on method `foo` depicted in Figure 2, we obtain the set of constraints given in Figure 3 that can be directly feed into Z3.

In order to debugging a program converted into constraints, we also have to have a failing test case and its constraint representation. For method `foo` we know that

$$(\texttt{x}, \texttt{state}, \texttt{trans}, \texttt{y}, \texttt{new\_state},) = (1, 1, 1, -2, 0)$$

is such a test case. This test case can be easily represented using the SMT-LIB presentation:

```
(assert (= x_1 1))
(assert (= state_1 1))
(assert (= trans_1 1))
(assert (= y_3 -2))
(assert (= new_state_5 0))
```

Following the basic definitions of Reiter **??** we can directly use Z3 to compute diagnoses.

Definition 1. *Given a diagnosis system $(SD, COMP)$ and a set of observations OBS. A set Δ is a diagnosis if and only if $SD \cup OBS \cup \{ab(c) | c \in \Delta\} \cup \{\neg ab(c) | c \in COMP \setminus \Delta\}$ is satisfiable. A diagnosis is said to be minimal if there is no subset that itself is a diagnosis.*

In the case of debugging using the conversion described before SD is the set of constraints (e.g., the SMT-LIB representation given in Figure 3), OBS is the set of `assert`'s, and $COMP$ is the set $\{ab_0, \ldots, ab_6\}$ representing the statements of method `foo`. Note that the last statement of the SMT-LIB presentation of `foo` is used to search for the smallest number diagnoses first. It is also worth noting that Z3 only allows to compute one diagnosis at a time. What can be done is to compute one diagnosis, add a constraint that this diagnosis is not allow and use Z3 again for computing the next diagnosis. This process can be continued until no further diagnoses can be computed.

Using the SMT-LIB representation of `foo` and the definition of diagnosis, we finally obtain the following three minimal diagnoses using Z3 as constraint solver:

- Statement 13 `new_state = 1`
- Statement 12 `if (trans == 1) ...`
- Statements 6, 7, and 8 together

In the next section, we describe the conversion tool Java2CSP, and the interaction with the tool in more detail.

3 The Java2CSP tool

The Java2CSP tool comes as a package of three individual parts — the *service*, performing the transformation of source code into a constraint satisfaction problem, *z3aas*, which solves the CSP, and *front*, a web interface for putting the two aforementioned service together into a user-friendly environment.

The reason behind splitting the tool into three parts is to allow future developments of tools which take advantage of the API offered by the service directly. Furthermore, as solving the CSP is the most computationally-demanding task, it makes sense to isolate this functionality in order to be able to offload it to a different machine.

3.1 Intermediate Language

Java2CSP is made to be extendable to other programming languages of a similar nature to Java (i.e., imperative, procedural). Any language which can be *lowered* into the internal representation language (IL) can make use of the existing infrastructure.

The IL is composed of two languages: Firstly, a language of *expressions* describes pure computations, whose evaluation involves consuming the state[6] and producing values. Secondly, a language of *statements* describes state-altering computations, whose evaluation involves consuming the state and producing a new one.

The expression language is composed of the following types of operations:

- *Unary operations* — logical negation (`bool`→`bool`) and arithmetic negation (`int`→`int`, `float`→`float`).
- *Binary operations*
  - Arithmetic — addition, subtraction, multiplication, division and modulo defined as `int`×`int`→`int` and `float`×`float`→`float`;
  - Equality — defined as `any`×`any`→`bool`;
  - Logical — conjunction, disjunction defined as `bool`×`bool`→`bool`;
  - Ordering — $\geq, \geq, \leq, \leq$ defined as `int`×`int`→`bool` and `float`×`float`→`bool`.
- *References* — expressions which evaluate to the value of a binding from the state (i.e., the value of a variable).
- *Array references* — expressions which extract an item from an array by its index, by first retrieving the array from the state, and then retrieving an item within the array. The array must be a reference, and the index expression must evaluate to an `int` value.
- *Literals* — expressions which evaluate to the values contained within (e.g. `int`, `float`, `bool`).

The statement language is composed of the following types of operations:

- *Declaration statements* set the type of a variable in the current environment. Re-definitions are not allowed, and the declaration of a variable must precede assignment statements which set its value.
- *Assignment statements* update the value of a variable in the environment. The assigned expression must resolve to the same type that was used to declare the variable.
- *Block statements* chain together multiple statements to be executed in order.
- *Conditional statements* (if and if/else) yield results depending on a predicate condition.

[6] State is represented by a mapping from symbols (identifiers) to values and types.

```
(declare-const y_1 Int)
(declare-const ab_0 Bool)
(declare-const y_0 Int)
(declare-const new_state_2 Int)
.....
(assert (or ab_0 (= temp_0_0 (= state_1 0))))
(assert (or ab_1 (= y_1 (* 3 x_1))))
(assert (or ab_2 (= temp_1_0 (and temp_0_0 (= trans_1 1)))))
(assert (or ab_3 (= new_state_1 1)))
(assert (= new_state_2 (ite temp_1_0 new_state_1 new_state_0)))
(assert (or ab_4 (= y_2 (* (- 0 2) x_1))))
(assert (or ab_5 (= temp_2_0 (and (not temp_0_0) (= trans_1 1)))))
(assert (or ab_6 (= new_state_3 1)))
(assert (= new_state_4 (ite temp_2_0 new_state_3 new_state_2)))
(assert (= y_3 (ite temp_0_0 y_1 y_2)))
(assert (= new_state_5 (ite temp_0_0 new_state_2 new_state_4)))
(assert (= objective (+ (ite ab_0 1 0) (ite ab_5 1 0) (ite ab_1 1 0) (ite ab_6 1 0)
    (ite ab_2 1 0) (ite ab_3 1 0) (ite ab_4 1 0))))
```

Figure 3: The SMT-LIB presentation of method `foo` from Figure 2 (ignoring some `declare-const` items.

- *Loop statements* are equivalent to the `while` statement in languages such as Java and C — the body is executed as long as the predicate condition evaluates to true.
- *Return statements* end the execution of the program, yielding a value.

3.2 Java lowering

Input code is submitted in the form of a Java class declaration, containing one or more (static) methods. The class declaration unit is transformed into a list of the methods defined within. Method declarations are lowered into methods regardless of their access modifiers. All methods declared inside a class will also appear in the model's flat sequence of methods.

Mapping from Java to the internal format is unfortunately not a bijective mapping, due to the different way in which JavaParser [7] and Java2CSP understand the meaning of *expression* and *statement*. For example, JavaParser sees `int a = 1;` as an expression statement containing an assignment expression. The Java2CSP IL sees this as an assignment statement. Java expression statements (assignment expressions and variable declaration expressions) are converted into assignment and declaration statements respectively. Furthermore, compound assignment statements (eg. `a += b;`) are lowered into simple assignment statements (eg. `a = a + b;`).

Example A few correspondences between Java (left) and the IL (right).

```
1  int a;                              ["declare", "a", "int"]
2  a = 0;                  ["assign", ["ref", "a"], ["int", 0]]
3  int a = 0;            ["block", ["declare", "a", "int"],
4                          ["assign", ["ref","a"], ["int", 0]]]
```

Expression support Literal value expressions (`int`, `float`, and `bool`), name expressions, array access expressions, a subset of unary and binary operations (the ones with direct correspondents in the IL).

[7] The parser used in the Java pipeline frontend. See `https://javaparser.org/`

Statement support Expression statements (see above), block statements, and some control flow statements (`if`, `while` and `return`).

3.3 Workflow

In the development of the tool, the following workflow was assumed: given a method for which a failing test case exists (and therefore from the test case, input parameters and expected values can be extracted), a *diagnostic* is produced in the form of a set of statements which are considered to be *faulty*. Since the diagnostic may indicate that otherwise correct statements are the issue, a mechanism allows users to specify that some statements are definitely correct.

The web-based interface is modelled after this workflow:

1. *Submit Code* — code of the source program (in the form of a class definition containing one or more methods) is submitted and lowered into the IL;
2. *Setup Test Scenario* — the method under test is selected and applied on the given input arguments;
3. *Additional Constraints* — expectations related to the values of variables at different points in execution are submitted;
4. *CSP Formulation* — the CSP (offered in the SMT-LIB format) can be inspected. Furthermore, a job can be submitted with a query containing the formulation, in order to produce a diagnostic. At this step, *false abnormals*[8] can be set, after which a new diagnostic can be requested.

3.4 Service overview

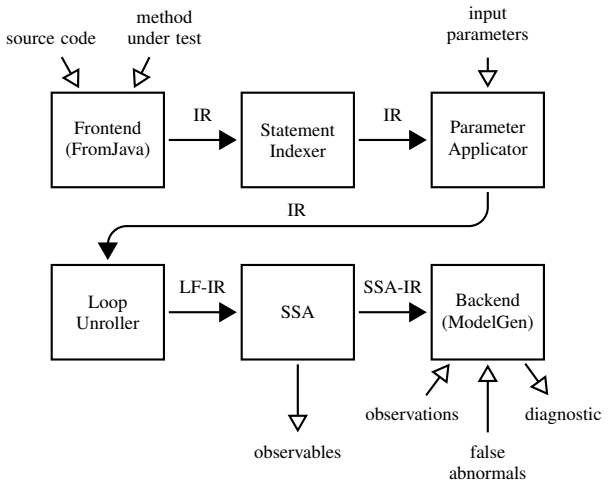
The *service* acts as an HTTP server, offering a stateless interface through which queries related to the individual transformation passes can be submitted. From an external perspective, the service can be seen as a sequence of three transformation passes:

1. The *frontend*, responsible for converting input source code into the IL. This functionality can be requested

[8] Markers which tell the solver that certain statements are to be assumed not faulty.

from the service by issuing a `POST` request with the program source code as payload to the route `/from/java`. Even though at the time of this writing the only supported input language is Java, the route structure allows future extensions.

2. The *transformation pipeline*, converting from the source program's internal representation into a set of constraints and an optimization objective.
3. The *backend*, converting the transformation pipeline's output into a form which can be understood by external tools for executing the CSP such that a diagnosis can be produced. At the service level, there are two routes handling this process — `/to/smtlib`, producing SMT-LIB compatible source code, and `/to/z3aas`, producing a query for the *z3aas* service.



Furthermore, the *transformation pipeline* is made up of five individual passes:

Statement Indexer walks the AST and assigns a unique identifier to every statement, in order to keep track of the origins of statements during subsequent transformation passes, which might otherwise rewrite the original statements.

Parameter Applicator prepends method parameter declarations and initializations to the body of a method, producing a sequence of statements which follow the execution of the method with the given parameters applied.

Loop Unroller transforms loop statements into nested conditional statements up to a certain level (which is by default 5, but can be overridden by decorating the loop with a comment indicating the number of unroll steps to use).

SSA performs static single assignment transformation, producing a sequence of (assignment) statements such that every variable is assigned to only once.

Model Generator produces a constraint satisfaction problem in a generic, internal form, given the execution trace resulted from the SSA transformation, a set of observations and a set of false abnormals.

3.5 Service interface

The service exposes the individual passes[9] through HTTP routes. Successful responses return a status code of `200`, while malformed or erroneous requests return `400`. Lastly, status code `500` is reserved for bugs inside the transformation pipeline.

What follows is an example interaction with the tool at the HTTP request level.

The first step involves submitting the input source code to the frontend in order to get a representation of the code in the intermediate language. This is done by issuing a `POST` request to `/from/java`. Given the example code in Figure 2, a partial response is reproduced below:

```
{ "name": "foo",
  "params": {
    "x": "int", "state": "int", "trans": "int" },
  "body": ["block", [
    ["declare", "y", "int"],
    ["declare", "new_state", "int"],
    ["if-else",
      ["eq?",
        ["ref", "state", {"range":"6:9-6:13"}],
        ["int", 0, {"range":"6:18-6:18"}],
        {"range":"6:9-6:18"}],
      ...
```

All expressions and statements which originate from the input source code contain metadata related to their position, again for the purpose of easily tracking down effects of transformations.

The next step in the pipeline is parameter application, which is done by performing a `POST` request to `/pass/apply` with a payload containing the method to be used and the parameters to be applied.

Transformation into a loop-free representation is handled by the `/pass/unroll` route, and SSA transformation is handled by `/pass/ssa`.

At this point, the SSA internal language is represented slightly differently, as variables have an *index* property:

```
[ ["assign", ["ref", "x", 1],     ["int", 1]],
  ["assign", ["ref", "state", 1], ["int", 1]],
  ["assign", ["ref", "trans", 1], ["int", 1]],
  ["assign", ["ref", "temp_0", 0],
    ["eq?",
      ["ref", "state", 1],
      ["int", 0]],
    0],
    ...
```

A note to be made about the representation is that the first three assignment statements do not have an index property, whereas the last statement in the listing has the index `0`. This is because the first three assignments come from initializing the parameters of the function, and not from code which was input by the user.

The CSP formulation in the SMT-LIB language can be requested through the `/to/smtlib` route. For immediate execution, the `/to/z3aas` route produces a query for the *z3aas* service, which can be directly sent to the server to submit a job to solve the constraint satisfaction problem.

In Figure 3 we outline the use of the Java2CSP tool when debugging the `foo` method from Figure 2. In the first step we insert the source code of `foo` into the tool. After pressing the *Next* button, we have to specify the input values in Step 2. Note that in the tool the values have to be specified using the following form `(` *type value* `)`, where *type* can be `int`, `float` or `bool`, and *value* the

[9] with the exception of the statement indexer, which is fused together with the frontend, emitting indexed statements

Figure 4: An example run through Java2CSP.

value. After pressing *Apply*, we arrive in Step 3, where we can add the expected output or any intermediate value. We have to do this for the variable comprising an index. The largest index stands for the last value the variable takes in the program. After pressing `Next` we arrive at Step 4, where we are able to start debugging via pressing the *Run* button. In each step only one fault candidate is delivered. However, via marking a certain statement as being not faulty and pressing *Run* again, we are able to proceed with debugging until no further candidates can be computed.

3.6 Distribution

Java2CSP is open-source, released under the terms and conditions of the MIT License. The three components are developed in lock-step, so the `master` branch of all three should make a working configuration.

The service is hosted at `https://git.sr.ht/~dumitru/modelinho-service`. Building from sources requires a Java Development Kit (JDK) and sbt[10]. A self-contained Java archive (JAR) is produced.

The frontend is hosted at `https://git.sr.ht/~dumitru/modelinho-front`. Building from sources requires Node.JS[11] and NPM[12]. The produced static artifacts need to be hosted by an HTTP server.

z3aas is hosted at `https://git.sr.ht/~dumitru/z3aas`. A Python installation is required, as well as pip[13] in order to install the required dependencies. z3aas also requires a Redis[14] node in order to be able to save the results of jobs.

A Docker-based distribution packing all of the service in one container is also available at `https://git.sr.ht/~dumitru/modelinho-container`. Starting the container will build the project from sources, from the latest version in the repositories.

4 Conclusions

In this paper, we discussed the foundations and implementation details of the *Java2CSP* tool, which maps Java programs into a constraint representation that can be directly utilized for locating bugs. The tool is publicly available and intended to serve as a proof-of-concept for the applicability of model-based reasoning for fault localization. Besides the tool itself its source code is also open source enabling others to contribute and further improve the tool.

Acknowledgments

The research presented in the paper has been funded in part by the Austrian Research Promotion Agency (FFG) under grant 865248 (Securing Web Technologies with Combinatorial Interaction Testing - SecWIT).

References

[1] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Trans. Software Eng.*, 42(8):707–740, 2016.

[2] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[3] Franz Wotawa, Mihai Nica, and Iulia Moraru. Automated debugging based on a constraint model of the program and a test case. *J. Log. Algebraic Methods Program.*, 81(4):390–407, 2012.

[4] H. Collavizza and M. Rueher. Exploring different constraint-based modelings for program verification. In *In* Principles and Practice of Constraint Programming (CP 2007), pages 49–63, Providence, RI, USA, September 2007.

[5] F. Wotawa and M. Nica. On the compilation of programs into their equivalent constraint representation. *Informatika*, 32:359–371, 2008.

[6] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 1998.

[7] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[8] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[10] The Scala Simple Build Tool. See `https://www.scala-sbt.org/`

[11] See `https://nodejs.org/en/`

[12] The Node Package Manager. See `https://www.npmjs.com/`

[13] The Python Package Installer. See `https://pypi.org/project/pip/`

[14] In-memory data structure store. See `https://redis.io/`