SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

An Environment for Continuous Integration and Software Testing for sys-sage

Jim Eckerlein

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

An Environment for Continuous Integration and Software Testing for sys-sage

Author:	Jim Eckerlein
Supervisor:	Prof. Dr. Martin Schulz
Advisor:	Stepan Vanecek
Submission Date:	17.04.2023

Acknowledgments

I would like to thank Prof. Dr. Martin Schulz and Stepan Vanecek for offering me the topic of this work. Further, special thanks go to Stepan Vanecek for always supporting me throughout the months and answering all of my questions.

Abstract

This thesis designs and implements a continuous software engineering strategy for sys-sage, a library for capturing and manipulating hardware topologies of computer systems. While doing so, it explores different approaches and techniques to check a correct implementation such as unit testing, mocking, dynamic analysis, coverage, and sanitizing while highlighting their respective strengths and benefits. Furthermore, this thesis compares several available Continuous Integration (CI) solutions and implements a working approach to CI for sys-sage.

Keywords: Test Driven Development, Unit tests, xUnit, Continuous Integration

Contents

1 Introduction

The sys-sage[1] library allows users to capture their hardware components such as NUMA memory nodes, storage, processors, chips, and caches. Furthermore, it models a containment relation and paths of data traversal between components. This allows programs to optimize procedures for a number of factors such as time or energy consumption. At the time of writing, sys-sage employs several ways of sourcing information:

- *mtg4*[2] for capturing the configuration of Nvidia GPUs
- *hwloc*[3] to retrieve an abstraction of the hierarchical topology of modern architectures
- *Intel CAT*[4] to gather cache properties of Intel CPUs
- Files such as */proc/cpuinfo* to gather CPU related information on Linux machines

Data from all sources is aggregated in a tree-graph data structure called node topology which models containment. Connections between components of uni- or bidirectional data transportation is also modeled by a digraph whose edges are called data paths and whose vertices are nodes from the node topology.

To increase the library's reliability we will augment the project with a unit testing library and implement a Continuous Integration (CI) solution. The following chapters are structured like this: In chapter 2 we will discuss several approaches to validation of source code and how we integrated a testing suite into sys-sage. In chapter 3 we will explain the purpose of CI, available solutions and how we integrated CI into sys-sage. Finally, in the chapter 4 we summarize all conclusions.

[1]https://github.com/caps-tum/sys-sage
[2]https://github.com/caps-tum/mt4g
[3]https://www.open-mpi.org/projects/hwloc/
[4]https://github.com/intel/intel-cmt-cat

2 Software Testing

Software testing describes the process of validating and verifying various aspects of software systems. As described in [22], there are many approaches to software testing. One notable method of software testing is *unit testing*.

2.1 Test Driven Development

Test Driven Development (TDD) is a specific approach to author unit tests in that the programmer writes test before implementing the feature to be tested. As explained in [3, 2], TDD can be part of the more general software development methodology known as extreme programming (XP). In general this process typically involves the following steps:

1. Write a test case for a new feature.
2. Run the test case, which initially fails because the feature has not been implemented yet.
3. Write the minimum code necessary to pass the test case.
4. Re-run the tests to verify a correct implementation.
5. Refactor the code to improve its design and maintainability.
6. Repeat the process for the next feature.

The steps are visualized in 2.1 with emphasis on the cyclicity of the TDD process. The advantages of using TDD is that software is thoroughly tested and that code quality is maintained throughout the development process. Furthermore, cost of software development is reduced as defects are caught early in the development process, when they are easier and less expensive to fix.

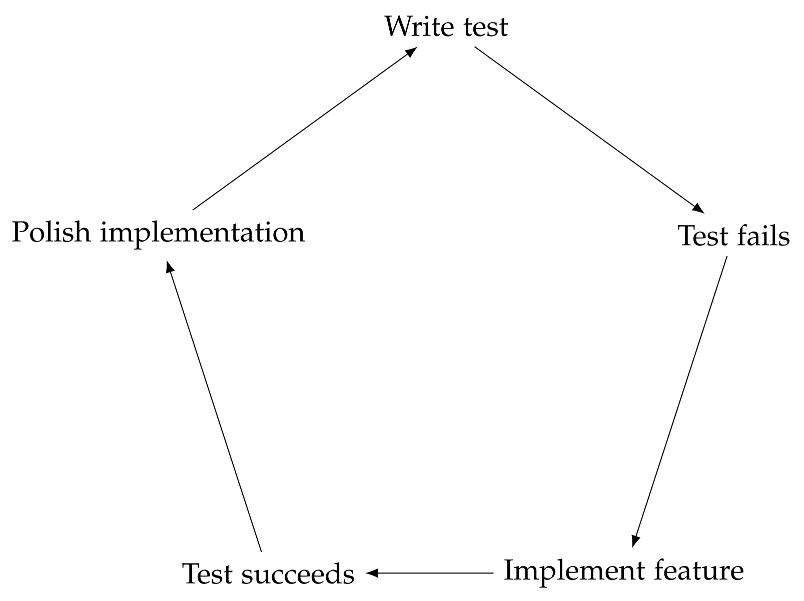

Figure 2.1: Steps of TDD

2.2 Unit testing

Unit testing is a software testing method in which individual units or components of a software system are tested in isolation from the rest of the system. A unit is the smallest testable part of a software system, such as a function or method. Unit testing is performed during the development phase of software development and is a key part of the TDD methodology. The main goal of unit testing is to ensure that each unit of the software system performs as expected and meets the requirements set out in the design specification. This is achieved by writing test cases that exercise the code in each unit and verifying that the output is correct for a given input. Usually, testing frameworks are used which implement a logic to construct assertions, structure the tests, and execute the tests, such as JUnit for Java or NUnit for .NET. Unit testing has several benefits, including:

- Unit testing allows developers to catch defects as early as possible.
- Unit testing speeds up development by reducing the time required for manual testing.
- Unit testing promotes the development of modular, reusable, and maintainable code.

- Unit testing improves collaboration between developers by providing a shared understanding of the software system.

Overall, unit testing is an essential part of modern software development practices and is an effective way to ensure that software systems are reliable, maintainable, and meet the requirements set out in the design specification.

2.2.1 xUnit

xUnit is a family of unit testing frameworks that originated from the original Smalltalk-based SUnit framework. xUnit frameworks follow a similar structure and share many common features, making it easy to learn and use a new framework from within the family.

The basic structure of an xUnit test framework involves defining test cases as methods in a test class, with each method testing a specific aspect of the code being tested. Each test method typically contains one or more assertions, which check whether a specific condition is true or false. The xUnit framework then provides a test runner that discovers and executes the test cases and reports the results.

Some examples of xUnit frameworks include JUnit for Java, NUnit for .NET, and CppUnit for C++. xUnit frameworks have become a widely used standard for unit testing across many programming languages and platforms, providing a consistent and standardized way to write and execute unit tests.

There is a plethora of unit test frameworks available, each with its distinguishing features and trade-offs, and the choice of framework depends on the needs of the specific project. While some frameworks support a rich set of features and thus are well-suited for large-scale projects which have non-trivial testing requirements, other frameworks focus more on simplicity, easy maintainability of the tests, and developer friendliness. Furthermore, other frameworks simplify the process of integrating the test suites into a predefined build system or allow the tests to be run in a resource-constrained environment.

We will compare a selection of xUnit implementations for the C++ language:

1. Boost.Test[1]
    - Part of the Boost libraries, open source
    - Provides a rich set of assertions and test fixtures
    - Supports test suites and hierarchical test organization
    - Supports both unit and acceptance testing

[1] `https://github.com/boostorg/test`

    - Can generate XML output for integration with other tools
    - Has integration with the Boost build system

2. Catch2[2]
    - Open source, header-only library
    - Provides a simple syntax for writing tests and assertions
    - Supports BDD-style testing with natural language constructs
    - Supports tag-based test filtering
    - Has a built-in test runner and supports integration with other test runners
3. CppUnit[3]
    - Open source, part of the xUnit family of frameworks
    - Provides a rich set of assertions and test fixtures
    - Supports test suites and hierarchical test organization
    - Can generate XML output for integration with other tools
    - Has integration with the CMake build system
4. doctest[4]
    - Open source, header-only library
    - Provides a simple syntax for writing tests and assertions
    - Supports subcases, which allow for more granular test results
    - Can detect memory leaks and report them as part of test results
    - Supports integration with other test runners
    - Lightweight and fast
5. GoogleTest[5], also known as gtest
    - Open source, developed by Google
    - Provides a rich set of assertions and test fixtures
    - Supports parameterized tests

[2] `https://github.com/catchorg/Catch2`
[3] `https://freedesktop.org/wiki/Software/cppunit/`
[4] `https://github.com/doctest/doctest`
[5] `https://github.com/google/googletest`

- Supports sharding, a form of work distribution, i.e. the set of tests to run is partitioned across several machines
- Can generate XML and JSON output for integration with other tools
- Can be used with a variety of build systems, including CMake and Make

6. µt[6]
   - Open source, header-only library
   - Leverages C++20 features:
     - `std::source_location` to replace the `__FILE__` and `__LINE__` macros
     - Non-type template parameters to allow for constant matchers
     - Template parameter list for generic lambdas
   - Provides a simple syntax for writing tests and assertions
   - Supports test suites and hierarchical test organization
   - Supports test fixtures, but with a lightweight implementation
   - Has a small footprint and is designed for use in resource-constrained environments
   - Can generate XML output for integration with other tools
   - It is deeply customizable, since custom test runners can be implemented.
   - Supports Behavior-driven development (BDD).
   - Supports Gherkin specifications.
   - In contrast to all other unit test frameworks, µt is implemented without any macros, if full C++20 language support is available.

The sys-sage project is rather small, and the focus was on keeping it that way, ruling out Boost.Test and Google Test. As sys-sage compilation time should not significantly increase due to the additional testing library, choosing a library with a small compile time footprint was essential, ruling out Catch2. According to Viktor Kirilov, the creator of doctest, Catch2 usually compiles 25x slower than doctest [7]. We want to be able to write several disjoint test suites which would subsume tests of different aspects of the library. Also, to ease maintenance, all test cases should be automatically registered. Since the API of the library is quite simple to use, we had no use of advanced testing techniques. As sys-sage already uses C++20, we were not restricted

[6]`https://github.com/boost-ext/ut`
[7]https://sched.co/BgsI

to macro-based test implementations. As such, we ended up choosing µt as out testing library because it is easy to integrate since it is a header-only library, has very fast compilation times, supports all the required testing constructs and due to its C++20 focused implementation future-proofed the project as a whole.

2.2.2 Writing unit tests with µt

To use the functionality from µt, first we have to include `boost/ut.hpp`. In sys-sage, this header is exported from the µt CMake interface library. Given a summation function defined like this:

```cpp
constexpr auto sum(auto... values) { return (values + ...); }
```

We can now place write expectations inside the main function:

```cpp
expect(sum(0) == 0_i);
expect(sum(1, 2) == 3_i);
expect(sum(1, 2) > 0_i and 41_i == sum(40, 2));
```

Integer literals are suffixed with the `_i` literal defined by µt. This allows the overloaded equal operator to be used. Without using the suffix, the built-in comparison operator would be used instead. This would still work, as `expect` can take a boolean value, but using the overloaded operators allows for more precise test failure messages.

Compare the following outputs:

1. Literally just using integers yields a plain boolean valued response:
   - **`expect(sum(2) == 0)`**
   - `FAILED in: ...:9 - test condition: [false]`
2. Using the overloaded literal suffixes yields a more descriptive error message:
   - **`expect(sum(2) == 0_i)`**
   - `FAILED in: ...:9 - test condition: [2 == 0]`
3. Alternatively, we can use matchers to enforce the same behavior:
   - **`expect(that % sum(2) == 0)`**
   - `FAILED in: ...:9 - test condition: [2 == 0]`

Assertions can be subsumed in tests using an overloaded string literal for convenience:

```
"sum"_test = []
{
  expect(sum(0) == 0_i);
  expect(sum(1, 2) == 3_i);
  expect(sum(1, 2) > 0_i and 41_i == sum(40, 2));
};
```

These tests can be arbitrarily nested inside other test groups. The square bracket pair indicates that the test block is actually a lambda, thus its execution is deferred and invoked by a test runner.

Initially, these expectations can reside inside the main function. But to decouple subsystems to be tested, we can define test suites. If there is a bijective mapping between test source files and subsystems, the test suite variable does not have to be named meaningfully. Instead, the name is encoded in the test suite's value template parameter. Declaring the global variable to be **static** allows us to name all the test suites _.

```
static suite<"arithmetic"> _ = []
{
  "sum"_test = []
  {
    expect(sum(0) == 0_i);
    expect(sum(1, 2) == 3_i);
    expect(sum(1, 2) > 0_i and 41_i == sum(40, 2));
  };
}
```

Tests inside suites are automatically invoked. For successful linkage, a main function still has to be defined, but if all test cases reside within suites, the body of the main function is empty.

2.3 Automatic XML validation

One particularly interesting function is **exportToXml**, serializing a topology from memory to the disk using an XML representation. This is quite a difficult functionality to test because each test should only validate a single feature. Since libxml2 is a C API, it is also quite verbose and error-prone to use. In order to easier scale the XML export tests, we outsource much of validation work to a formal XML Schema Definition (XSD). Instead of procedural validating a semantically valid hierarchy of XML nodes in each test, we author another XML file which specifies a correctly exported document in a declarative way. Fortunately, libxml2 already implements XML schemas, thus we do not need to introduce another library dependency.

XML schemas are used to define the structure, content, and data types of an XML document. An XML schema provides a set of rules that a document must follow in order to be considered valid according to that schema.

An XML schema is typically defined in a separate XML file, which can be referenced by other XML documents. The schema file contains a set of elements, attributes, and data types, along with rules for how they can be used in a document. For example, the schema might specify that an element must have a certain name, can occur a certain number of times, or must contain certain child elements or attributes.

XML schemas use namespaces to identify the elements and data types defined in the schema. This allows different schemas to be used together in the same document without conflicts. A schema can also define complex data types, such as sequences or choices of elements, and can include constraints, such as minimum and maximum values, regular expressions, or references to other elements or attributes.

When an XML document is validated against an XML schema, the document is checked to make sure that it follows all the rules specified in the schema. This includes checking the structure of the document, validating the data types of its elements and attributes, and ensuring that it meets any other constraints specified in the schema. If the document does not conform to the schema, it is considered invalid.

XML schemas are widely used in a variety of applications, including web services, data exchange formats, and configuration files. By providing a standard way to define the structure and content of XML documents, XML schemas make it easier for different systems to exchange data and work together seamlessly.

For the sys-sage unit tests, we require a single XML schema describing a valid export. The head of such a schema file states the XML version and the version of the XML schema.

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

To state that a certain node named `x` is required in a certain place, we use the `<xs:element name="x">` tag. Inside such an element, we can use `<xs:complexType>` to describe the structure of said element. In our case, the outermost element is `<sys-sage>` which can contain both the `<components>` and `<data-paths>` elements.

```xml
<!-- The <sys-sage> root tag -->
<xs:element name="sys-sage">
  <xs:complexType>
```

```
    <xs:all>

      <!-- The component trees -->
      <xs:element name="components">
        <!-- ... -->
      </xs:element>

      <!-- The data paths -->
      <xs:element name="data-paths">
        <!-- ... -->
      </xs:element>

    </xs:all>
  </xs:complexType>
</xs:element>
```

2.3.1 Components

The components form a recursive data structure. The schemas for all possible sub-elements such as cores or threads are grouped within a sequence of choices.

```
<!-- Each component can recursively contain more components -->
<xs:group name="components">
  <xs:sequence>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="None" type="none" />
      <xs:element name="HW_thread" type="hw_thread" />
      <xs:element name="Core" type="core" />
      <xs:element name="Cache" type="cache" />
      <xs:element name="Subdivision" type="subdivision" />
      <xs:element name="NUMA" type="numa" />
      <xs:element name="Chip" type="chip" />
      <xs:element name="Memory" type="memory" />
      <xs:element name="Storage" type="storage" />
      <xs:element name="Node" type="node" />
      <xs:element name="Topology" type="topology" />
      <xs:element name="Attribute" type="attribute" />
    </xs:choice>
  </xs:sequence>
</xs:group>
```

All component elements inherit from a common base type called **component** as XSD allows for type inheritance. This base type contains the common fields **id**, **name**, **addr**, and **count**. The following code listing defines the component base type and its fields which are typed accordingly:

```
<!-- Component base class -->
<xs:complexType name="component">
  <xs:group ref="components"></xs:group>
  <xs:attribute name="id" type="xs:integer" />
  <xs:attribute name="name" type="xs:string" />
  <xs:attribute name="addr" type="addr" />
  <xs:attribute name="count" type="xs:integer" />
</xs:complexType>
```

The **group** tag refers to the group defined above and means that each component itself can contain one or more components itself.

The **addr** type of the equally called **addr** field is a custom type. This memory address is used as a unique identifier for all components and are required by data paths and is encoded has a hexadecimal string of digits with a **0x** prefix:

```
<!-- A memory address, e.g. 0xff0011234 -->
<xs:simpleType name="addr">
  <xs:restriction base="xs:string">
    <xs:pattern value="0x[0-9a-fA-F]+"></xs:pattern>
  </xs:restriction>
</xs:simpleType>
```

To derive from a type, we use the **<xs:extension>** tag. The following code listing defines the memory type, its inheritance, and its fields which are typed accordingly:

```
<!-- Memory component -->
<xs:complexType name="memory">
  <xs:complexContent>
    <xs:extension base="component">
      <xs:attribute name="size" type="xs:integer" />
      <xs:attribute name="is_volatile" type="xs:integer" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

The other components are defined similarly.

2.3.2 Data paths

Data paths form a second graph with components as vertices and data paths as edges. Since data paths do not form a recursive data structure, their XSD is much more terse. Indeed, their type is defined inline, not at another place in the XML file and are grouped within a repeated choice. The **source** and **target** attributes refer to components using the unique memory addresses and are thus typed as **addr**.

```
<!-- The data paths -->
<xs:element name="data-paths">
  <xs:complexType>
    <xs:choice>
      <xs:element name="datapath" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Attribute" type="attribute" minOccurs="0"
              maxOccurs="unbounded" />
          </xs:sequence>
          <xs:attribute name="source" type="addr" />
          <xs:attribute name="target" type="addr" />
          <xs:attribute name="oriented" type="xs:integer" />
          <xs:attribute name="dp_type" type="xs:integer" />
          <xs:attribute name="bw" type="xs:double" />
          <xs:attribute name="latency" type="xs:double" />
        </xs:complexType>
      </xs:element>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

2.3.3 Attributes

Sys-sage allows the library user to embed custom data in components and data paths. Some hardware specific extensions of sys-sage itself use this feature. An example would be the CPU info module which can retrieve the CPU frequencies. These custom attributes are stored in a hash map, mapping strings which encode the attribute name to their actual values. Attributes are exported using user-supplied callbacks which are called during the export. Since the layout is not knowable by sys-sage, their XSD explicitly tells the validator to skip their contents:

```
<!-- Attributes for components and data paths -->
```

```
<xs:complexType name="attribute">
  <xs:sequence>
    <xs:any minOccurs="0" maxOccurs="unbounded" processContents="skip" />
  </xs:sequence>
  <xs:attribute name="name" />
  <xs:attribute name="value" />
</xs:complexType>
```

2.4 Mocking

Mocking as explained in [20] is a technique used in software testing to simulate the behavior of a complex system or component that a software unit depends on, but which is difficult or impossible to include in the testing environment. In other words, it allows developers to create test environments that mimic the behavior of external dependencies, without actually relying on the real system or component.

Mocking is typically used in unit testing to isolate and test individual components of a software system. When a unit under test depends on another component or system, such as a database, web service, or external API, mocking can be used to simulate the behavior of the external dependency and control the input and output of the system or component. This allows the developer to test the behavior of the unit under different scenarios, without affecting the real external dependency.

Mocking is often implemented using a mocking framework or library, such as Mockito for Java or Moq for .NET. These frameworks allow developers to create mock objects that simulate the behavior of external dependencies. The mock object provides a stand-in for the real dependency, allowing the developer to control its behavior and responses to specific inputs.

Mocking has several benefits, including:

- Isolation: Mocking allows to isolate the component being tested from its dependencies. This isolation ensures that only the specific behavior of the component under test are being tested, rather than the combined behavior of the component and its dependencies.

- Control: Mocking provides a way to control the behavior of the mocked objects during testing, allowing to simulate various scenarios and test edge cases that might be difficult to reproduce in a real-world environment.

- Speed: Mocking can help speed up testing, as it allows to test the component in isolation, without the need for a full environment or external resources. This can help reduce the time and effort required for testing and make the development process more efficient.

- Reliability: Mocking helps to improve the reliability of tests, as it ensures that the tests are consistent and repeatable, regardless of the external environment or conditions.
- Maintainability: Mocking can also help to improve the maintainability of software systems, as it allows to easily update and refactor code without affecting the behavior of the mocked objects or the tests that depend on them.

In the specific case of sys-sage, we investigated whether we can mock hardware and platform specific features in order to run the tests on any hardware. The first problem we encountered was the fact that freestanding functions, i.e. not methods, cannot be mocked at all. In addition to this requirement, all mocked methods must be virtualized, since mock objects overwrite methods by inheriting from the real base class. Some mocking frameworks such as gmock[8] also allow using non-virtual mocking objects, but in that case the whole library has to be rewritten to template functions, which either accept the real implementation or the mocked one. Furthermore, it has to be made clear that the mocked code itself is not tested. Instead, the interaction between the mocked code and the rest of the code base is tested. But since the platform specific functions just interact with third-party APIs and store the result in a variable, there is no interaction worth being tested. The results are also not exported in the XML. We deem the trade-off to virtualize methods to make them mockable as not worth being paying.

2.5 Code coverage

Code coverage refers to the practice of injecting code in order to track which code paths a program executes at runtime. GNU Compiler Collection (GCC) has options in order to automatically inject such instructions. Assume with have a C++ program like this:

```cpp
int foo(int x)
{
  if (x < 2)
  {
    return -1;
  }
  else if (x < 10)
```

[8]https://google.github.io/googletest/gmock_cook_book.html#MockingNonVirtualMethods

```
  {
    return 2;
  }
  else
  {
    return 5;
  }
}

int main()
{
  int acc = 0;
  for (int i = 0; i < 5; ++i)
  {
    acc += foo(i);
  }
  return acc;
}
```

To inspect with code paths the program takes at runtime, we use GCC[9] to generate code to reconstruct coverage information. In order to do so, compile the file with an additional option:

```
g++ main.cpp -o main --coverage
```

When inspecting the file system after compilation, we notice a new file called *main.gcno*. It contains additional compile-time information such as a graph representation of the program and a mapping of that graph to source line numbers. When executing the binary file *main*, a new file will be written called *main.gcda* containing runtime information about how the graph represented by the program has been executed. The information in both the *.gcno* and *.gcda* files are encoded in a compact binary format, thus not readable with a text editor. In order to access the information in a human-readable way, we need to use another tool: **`gcov main.cpp`** generates a *main.cpp.gcov* file which is the verbatim C++ code with annotations:

```
        -:    0:Source:main.cpp
        -:    0:Graph:main.gcno
        -:    0:Data:main.gcda
        -:    0:Runs:1
        -:    0:Programs:1
```

[9] https://gcc.gnu.org/

```
        5:    1:int foo(int x)
        -:    2:{
        5:    3:  if (x < 2)
        -:    4:  {
        2:    5:    return -1;
        -:    6:  }
        3:    7:  else if (x < 10)
        -:    8:  {
        3:    9:    return 2;
        -:   10:  }
        -:   11:  else
        -:   12:  {
    #####:   13:    return 5;
        -:   14:  }
        5:   15:}
        -:   16:
        1:   17:int main()
        -:   18:{
        1:   19:  int acc = 0;
        6:   20:  for (int i = 0; i < 5; ++i)
        -:   21:  {
        5:   22:    acc += foo(i);
        5:   23:  }
        1:   24:  return acc;
        -:   25:}
```

The first five lines form the preamble containing meta information about the coverage itself, such as the binary files this dump is constructed from and how many times the program has been executed. Consecutive program executions are non-destructive and instead accumulate the graph traversals. A leading integer number before a colon on a line states how many times this line has been executed across all program executions. A leading hash symbol states that this line has never been executed. Lines with a leading dash do not contain executable code. From the above listing, we can infer that **`foo`** has been called five times, twice returning negative one, thrice returning two, and never returning five.

Code coverage is quite helpful when authoring tests because it pinpoints code paths which have not been tested due to a special argument permutation required or even entire functions.

For sys-sage, we employed code coverage to check how many library functions have been tested, and how many edge cases were triggered by the unit tests. To enable code coverage, the configuration option **`TEST_COVERAGE`** has to be set.

2.6 Dynamic code analysis

Dynamic code analysis is a software testing technique that involves analyzing the behavior of a program as it runs on an actual processor or within a simulated environment. Tools derive errors, memory leaks, buffer overflows, race conditions, performance issues, security vulnerabilities, and other problems from the runtime behavior of the program. Examples for dynamic code analysis tools for C++ include Valgrind[10] and chap[11].

In contrast, static code analysis investigates the program without actually running it. Static code analysis tools can operate at different levels such as source code, assembly or byte code level. They identify problems such as syntax errors, type errors, code duplication, or potential vulnerabilities. Since the program does not have to be actually run, static analysis works regardless the hardware requirements of the program and thus is usually tightly integrated in an IDE. Examples for static code analysis tools for C++ include cpplint[12] and clang_analyzer[13].

2.7 Input Generation

Automated test input generation is the technique to automatically generate input data for tests in order to conform to a specific condition, such as failing an assertion and thereby proving the unsoundness of the component under test. With advances of computer power in the 2000s, new computationally intensive approaches have been developed, such as symbolic execution, search based testing, and fuzzing. In the following sections, we will shed light on a few of these techniques.

2.7.1 Symbolic Execution

Symbolic execution has been proposed by [18]. It is a static analysis technique that executes a program with symbolic inputs rather than with concrete values. While running a program, at each instant the program is said to be in a symbolic state. This state depends solely on the program inputs and a set of constraints on the inputs which all have to be fulfilled. The set of constraints is called the path condition. While running the program, the symbolic state and the path condition are built. Initially, the path condition starts out as a tautology as the program is able to start with any input values. During each step modifying a variable, the symbol is also updated inside the symbolic

[10]https://valgrind.org/
[11]https://github.com/vmware/chap
[12]https://google.github.io/styleguide/
[13]https://clang-analyzer.llvm.org/

state. When reaching a divergent control flow, the analysis tracks each separate branch while adding a constraint to each new path condition that makes the program follow a specific branch. To compute what inputs, if any, will execute a particular statement or result in branching in a specific way, the path condition at that point is given to an satisfiability modulo theories (SMT) solver.

A way of generating a minimal set of inputs that yield a certain symbolic state from the path condition is explained in [21, 1, 11], and SAT4J[14] is available as a Java implementation.

[4] is an implementation of a symbolic execution tool which automatically generates test with focus on high coverage (usually above 90%). Another implementation of symbolic execution can found in [5]. [8] is an improvement in the sense that it only symbolically executes arbitrary portions of a full system and allows to seamlessly transition back and forth between symbolic and concrete execution. Further reading about the topic of symbolic execution is available in [6].

The effectiveness of symbolic execution is bottle necked by the theories of constraint solving. Sometimes sets of constraints may arise that the constraint solver is unable to solve. A major improvement in that regard is dynamic symbolic execution (DSE) as described in [9, 15, 19]. In contrast to regular symbolic execution, it runs a concrete execution in addition to the symbolic execution. Implementations for DSEs are available in [13, 24, 26].

2.7.2 Search-based Testing

Search-based software testing (SBST) is another approach that uses metaheuristic search algorithms to explore a space of input candidates to cover all branches in a program. The problem of finding such a candidate is interpreted as an optimization problem which is guided by heuristics such as genetic algorithms, simulated annealing, hill climbing techniques, scatter search, particle swarm optimization, and tabu search. Found solutions are ranked by fitness functions.

SBST is able to generate large number of tests, increasing coverage thereby. The technique is also good at finding edge cases and boundary conditions that might be difficult to identify manually. Main challenges include the need to define appropriate fitness functions that capture the desired testing criteria, the computational complexity of the search process, and the risk of overfitting the test cases to specific implementations of the software. Despite these challenges, SBT seems to be an effective approach to automated testing in a variety of applications.

An application of SBST in the broader context of software engineering is given in [16].

[14] `http://www.sat4j.org/`

2.7.3 Random Testing

While totally random input data for tests is a valid approach, it generates a possibly unmanageable number of test cases. Thus, techniques have been developed to select a subset of random data to decrease the number of tests such as in [10, 23]. One such technique is adaptive random testing as in [7] which selects new test input that is the most distant from the previously executed test inputs, according to a certain criterion, while other candidates are discarded. Studies like [17] show that adaptive random testing yields in far less test cases, although at the cost of additional overhead generating them.

2.7.4 Fuzzing

Like random testing, fuzzing is a technique for generating random input for tests in order to find vulnerabilities or errors. Unlike random testing though, the input data is not sampled from a random input space, but instead retrieved by mutating or transforming input data that is known to be valid in a way that makes the originally valid input data malformed, out of range, or otherwise erroneous. Fuzzing is particularly useful for testing systems that accept untrusted input, such as web applications, network protocols, and file formats, since it can help to identify security vulnerabilities or other unexpected behavior.

The problem with black-box fuzzers is that without knowledge about the semantic structure of the program to be tested, the odds of triggering a vulnerability is usually low. As an example, the probability of triggering the error case in this function is 2^{-32}.

```
int foo(int x) {
  int y = 2 * x;
  if (y == 42) {
    std::abort();
  }
  return y + 2;
}
```

White-box fuzzers such as [14] alleviate this problem by gaining insight to the program semantics using approaches such as DSE.

In the specific case of sys-sage, after consideration, fuzzing has been deemed to be not feasible. The library puts a lot of burden on the user. Effectively no function tests for null pointers or invalid values in general. Also, semantically invalid inputs to functions such as a cyclic tree structure are not detected by functions and lead to non-terminating executions. Increasing robustness would introduce additional complexity and overhead to the library which has been deemed undesirable.

2.8 Sanitizers

Sanitizers are tools that inject code or instructions into the output binary which are used to detect runtime errors that can lead to security vulnerabilities, crashes, or other unexpected behavior in a software.

There are several types of sanitizers, each targeting different types of bugs. Some of the most common types include:

1. *Address Sanitizer (ASan)*: Detects memory-related errors such as buffer overflows, use-after-free, and memory leaks.
2. *Memory Sanitizer (MSan)*: Detects uninitialized memory reads.
3. *Thread Sanitizer (TSan)*: Detects data race conditions, which occur when multiple threads access the same data simultaneously without proper synchronization.
4. *Undefined Behavior Sanitizer (UBSan)*: Detects sources of undefined behavior such as integer overflows or null pointer dereferences.

By detecting issues that are impossible to detect ahead of execution, sanitizers help to improve the quality and security of software systems. Since they provide automated feedback to the developer, sanitizers also reduce time and effort required for manual testing and debugging. A lot of IDEs also support sanitizers natively or through third-party plugins, which can be enabled by the developer during the build process or as part of the testing pipeline. sys-sage accepts the following configuration options:

- **`TEST_ASAN`** enables the address sanitizer
- **`TEST_TSAN`** enables the thread sanitizer
- **`TEST_UBSAN`** enables the undefined behaviour sanitizer

Notice though that **`TEST_ASAN`** and **`TEST_TSAN`** mutually exclusive, i.e. only one can be enabled per build.

3 Continuous Integration

CI as explained in [25] is a part of the continuous software engineering methodology consisting of CI, Continuous Deployment (CD), and Continuous Delivery (CDE). The goals of these three paradigms can be summarized as such:

- **CI** refers to the practice of committing and merging authored source code from all developers working on the same project on a regular basis, such as multiple times a day. Each time work is merged, the production-readiness is verified by automated tests. CI aims to reduce release cycles and increase developer productivity [12].
- **CDE** aims to improve customer feedback by always delivering a production-ready build after each successful CI run [27].
- **CD** is an advancement of CDE as it automatically deploys the built application to the production environment. CDE on the one hand can be seen as a pulled-based mode where a deployment has to be decided, planned, and at least partially involves manual steps. CD on the other hand can be interpreted as a push-based ll approach where deployment occurs automatically on a regular basis without any manual steps. [25]

3.1 Comparison of CI solutions

Some popular CI tools include Jenkins[1], Travis CI[2], CircleCI[3], and GitHub Actions[4], which provide a wide range of features such as automated builds, testing, and deployment, as well as integrations with popular code repositories and development tools. Many modern software development teams use CI as a core part of their development workflow, as it helps them to deliver high-quality software more efficiently and reliably. As sys-sage is hosted on a GitHub repository, we decided to implement the CI solution

[1]`https://www.jenkins.io`
[2]`https://www.travis-ci.com`
[3]`https://circleci.com`
[4]`https://github.com/features/actions`

	Jenkins	Travis CI	CircleCI	GitHub Actions
Available since	2011	2011	2011	2019
Open source	✓	✓		
Free hosted tier			✓	✓
Minutes per month in free hosted tier	N/A	N/A	6000	2000
Free self-hosting	✓	✓		✓
CD	✓	✓	✓	✓
Hardware				
x86	✓	✓	✓	✓
Arm	✓	✓	✓	
PowerPC	✓	✓		
IBM Z	✓	✓		
OS				
Linux	✓	✓	✓	✓
FreeBSD	✓	✓		
macOS	✓	✓	✓	✓
Windows	✓	✓	✓	✓
C++ support	✓	✓	✓	✓

Table 3.1: Comparison of CI solutions.

using GitHub Actions to keep the configuration and maintenance overhead of sys-sage at a minimum.

3.2 GitHub Actions

According to [28], approximately 30% of open-source GitHub repositories use GitHub Actions. Furthermore, the survey indicates that implementing the CI solution has key benefits:

1. More rejections of pull requests (PRs)
2. More communication in accepted PRs
3. Less communication in rejected PRs
4. Fewer commits in accepted PRs
5. More commits in rejected PRs

6. More time to accept a PR

3.3 Workflow

GitHub Actions allows defining workflows which are executed upon a specified triggering event, such as a push, a new PR, or merging a PR into the main branch. Workflows are defined in declarative YAML files inside the **`.github/workflows/`** directory. Each workflow file defines its triggering event using the **`on`** property and can define multiple actions using the **`jobs`** property. Every action has to specify on which platform it is supposed to run using the **`runs-on`** property. In the case of sys-sage, we use the **`ubuntu-latest`** platform referring to the latest available release of the Ubuntu distribution. Jobs can form vertices inside a possibly disconnected digraph using the **`needs`** property. If the specified action happens to fail, the action the property is declared on will not be invoked. This allows to construct entire pipelines composed of multiple action which can fail mid-execution. We made use of this feature to run the sanitizers only if the unit tests succeeded.

The commands executed by the action are defined using the **`steps`** array-property. Each step can be either a predefine action indicated by **`uses`** or a shell command set by **`run`**. The unit testing action for sys-sage has 4 steps:

1. Checkout the repository using **`actions/checkout@v3`**
2. Configure the project by running CMake
3. Build the test suites
4. Run the tests

Steps can communicate failure through a non-zero exit code from the process.

The other actions are in principle the same as the unit test but enable certain configuration options to enable a specific sanitizer. This introduces considerable code repetition inside the workflow file. GitHub provides reusable actions which can be used as subroutines with input and outputs variables. The common workflow that checks out, configures, builds, and runs the tests is realized as a reusable workflow with a single parameter for additional configuration flags. The unit testing action and all sanitizer actions call this common action, the sanitizer action specifying the required flag. The overhaul CI workflow can be visualized as the digraph in 3.1.

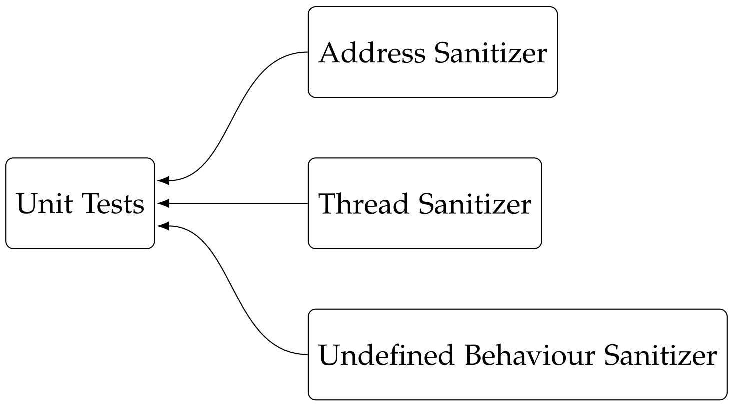

Figure 3.1: CI workflow
Arrow direction indicates dependency.

3.3.1 Documentation Coverage

To generate a documentation coverage report, we use Coverxygen [5] which is an open-source command-line tool. We first instruct Doxygen, which is the documentation generation tool that sys-sage uses, to produce an XML representation of the documentation. This output is then consumed by Coverxygen which assembles a single JSON report. To print all important undocumented symbols and their source location, we created a Python script. To verify adequate documentation coverage on an automatic basis, we added a job to the CI pipeline.

[5]https://github.com/psycofdj/coverxygen

4 Summary

We have successfully implemented a CI solution for sys-sage. For the test framework we chose µt. In total there are 7 test suites corresponding to the modules: **`topology`**, **`data-path`**, **`export`**, **`cpuinfo`**, **`hwloc`**, **`gpu-topo`**, and **`caps-numa-benchmark`**. There are 43 test cases containing 226 assertions. To run the tests, configure the project while enabling the configuration option in the sys-sage root directory. Building the test target then yields a binary which executes all test cases:

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Debug -DTEST=ON
cmake --build build --target test
./build/test/test
```

The tests can easily be extended by new suites and test cases as new modules and functions are implemented into sys-sage. To aid the validation of the XML exporter, we authored a XSD file which can be used to formally verify exported XML files. To find more bugs, we added configuration options to build instrumented binaries. GCC offers an address-, a thread-, and an undefined behavior sanitizer. To inspect that all functions and all of their branches were tested, we added a configuration option to build special binaries which report their runtime coverage.

As the CI solution, we chose GitHub Actions. Upon each commit pushed to the repository, a pipeline is started which first runs all test cases. If they succeeded, the tests are re-run with different sanitizers. In addition to that, the documentation coverage is reported.

Abbreviations

TDD Test Driven Development

BDD Behavior-driven development

XSD XML Schema Definition

GCC GNU Compiler Collection

CI Continuous Integration

CD Continuous Deployment

CDE Continuous Delivery

SMT satisfiability modulo theories

DSE dynamic symbolic execution

SBST search-based software testing

PR pull request

Bibliography

[1] J. Bailey and P. J. Stuckey. "Discovery of Minimal Unsatisfiable Subsets of Constraints Using Hitting Set Dualization." In: *Proceedings of the 7th International Conference on Practical Aspects of Declarative Languages*. PADL'05. Long Beach, CA: Springer-Verlag, 2005, pp. 174–186. ISBN: 3540243623. DOI: `10.1007/978-3-540-30557-6_14`.

[2] Beck. *Test Driven Development: By Example*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0321146530.

[3] K. Beck. *Extreme Programming Explained: Embrace Change*. USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0201616416.

[4] C. Cadar, D. Dunbar, and D. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 209–224.

[5] C. Cadar and D. Engler. "Execution Generated Test Cases: How to Make Systems Code Crash Itself." In: Aug. 2005, pp. 902–902. ISBN: 978-3-540-28195-5. DOI: `10.1007/11537328_2`.

[6] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser. "Symbolic execution for software testing in practice: preliminary assessment." In: *2011 33rd International Conference on Software Engineering (ICSE)* (2011), pp. 1066–1071.

[7] T. Chen, T. Tse, and Y. Yu. "Proportional Sampling Strategy: A Compendium and some Insights." In: *Journal of Systems and Software* 58 (Aug. 2001), pp. 65–. DOI: `10.1016/S0164-1212(01)00028-0`.

[8] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. "Selective Symbolic Execution." In: (Jan. 2009).

[9] L. A. Clarke and D. J. Richardson. "Applications of symbolic evaluation." In: *Journal of Systems and Software* 5.1 (1985), pp. 15–35. ISSN: 0164-1212. DOI: `https://doi.org/10.1016/0164-1212(85)90004-4`.

[10] C. Csallner and Y. Smaragdakis. "JCrasher: an automatic robustness tester for Java." In: *Software: Practice and Experience* 34.11 (2004), pp. 1025–1050. DOI: `https://doi.org/10.1002/spe.602`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.602`.

[11] B. Dutertre and L. M. de Moura. "The YICES SMT Solver." In: 2006.

[12] B. Fitzgerald and K.-J. Stol. "Continuous Software Engineering: A Roadmap and Agenda." In: *Journal of Systems and Software* 25 (July 2015). DOI: `10.1016/j.jss.2015.06.063`.

[13] P. Godefroid, N. Klarlund, and K. Sen. "DART: Directed Automated Random Testing." In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA: Association for Computing Machinery, 2005, pp. 213–223. ISBN: 1595930566. DOI: `10.1145/1065010.1065036`.

[14] P. Godefroid, M. Y. Levin, and D. Molnar. "SAGE: Whitebox Fuzzing for Security Testing: SAGE Has Had a Remarkable Impact at Microsoft." In: *Queue* 10.1 (Jan. 2012), pp. 20–27. ISSN: 1542-7730. DOI: `10.1145/2090147.2094081`.

[15] N. Gupta, A. Mathur, and M. Soffa. "Generating test data for branch coverage." In: *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*. 2000, pp. 219–227. DOI: `10.1109/ASE.2000.873666`.

[16] M. Harman, S. A. Mansouri, and Y. Zhang. "Search-Based Software Engineering: Trends, Techniques and Applications." In: *ACM Comput. Surv.* 45.1 (Dec. 2012). ISSN: 0360-0300. DOI: `10.1145/2379776.2379787`.

[17] H. Hemmati, A. Arcuri, and L. Briand. "Achieving scalable model-based testing through test case diversity." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22 (Feb. 2013). DOI: `10.1145/2430536.2430540`.

[18] J. C. King. "Symbolic Execution and Program Testing." In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: `10.1145/360248.360252`.

[19] B. Korel. "A dynamic approach of test data generation." In: *Proceedings. Conference on Software Maintenance 1990* (1990), pp. 311–317.

[20] T. Mackinnon, S. Freeman, and P. Craig. "Endo-testing: unit testing with mock objects." In: *Extreme programming examined* (2000), pp. 287–301.

[21] L. de Moura and N. Bjørner. "Z3: an efficient SMT solver." In: vol. 4963. Apr. 2008, pp. 337–340. ISBN: 978-3-540-78799-0. DOI: `10.1007/978-3-540-78800-3_24`.

[22] A. Orso and G. Rothermel. "Software testing: A research travelogue (2000-2014)." In: *Future of Software Engineering, FOSE 2014 - Proceedings* (May 2014). DOI: `10.1145/2593882.2593885`.

[23] C. Pacheco and M. D. Ernst. "Eclat: Automatic Generation and Classification of Test Inputs." In: *Proceedings of the 19th European Conference on Object-Oriented Programming*. ECOOP'05. Glasgow, UK: Springer-Verlag, 2005, pp. 504–527. ISBN: 354027992X. DOI: 10.1007/11531142_22.

[24] K. Sen, D. Marinov, and G. Agha. "CUTE: A concolic unit testing engine for C." In: vol. 30. Sept. 2005, pp. 263–272. DOI: 10.1145/1095430.1081750.

[25] M. Shahin, M. Ali Babar, and L. Zhu. "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices." In: *IEEE Access* PP (Mar. 2017). DOI: 10.1109/ACCESS.2017.2685629.

[26] N. Tillmann and J. Halleux. "Pex-white box test generation for .NET." In: Apr. 2008, pp. 134–153. ISBN: 978-3-540-79123-2. DOI: 10.1007/978-3-540-79124-9_10.

[27] I. Weber, S. Nepal, and L. Zhu. "Developing Dependable and Secure Cloud Applications." In: *IEEE Internet Computing* 20 (May 2016), pp. 74–79. DOI: 10.1109/MIC.2016.67.

[28] M. Wessel, J. Vargovich, M. A. Gerosa, and C. Treude. "GitHub Actions: The Impact on the Pull Request Process." In: (June 2022). DOI: 10.48550/arXiv.2206.14118.