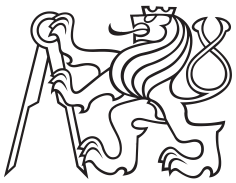
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Framework to improve testing of software heavily dependent on HW and OS configuration

Jan Novák

Supervisor: Ing. Karel Frajták, Ph.D.
Field of study: Open Informatics
Subfield: Software Engineering
May 2021

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Novák** Jméno: **Jan** Osobní číslo: **420343**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Framework pro zkvalitnění testování softwaru silně závislého na hardwarové konfiguraci a konfiguraci OS

Název diplomové práce anglicky:

Framework to improve software testing heavily dependent on the HW and OS configuration

Pokyny pro vypracování:

Vytvořte framework pro zkvalitnění testování softwaru silně závislého na hardwarové konfiguraci a konfiguraci OS. Vyvíjená aplikace je silně závislá na HW konfiguraci a konfiguraci OS, na kterém pak běží. Ovladače a další kritické komponenty OS jsou výrobci vydávány a aktualizovány téměř týdně; vývojářské týmy upravují kód téměř neustále. Počet regresních chyb je v tomto okamžiku vysoký a otestovat novou verzi SW na všech možných kombinacích HW a OS je nemožné z důvodu jejich vysokého počtu, zároveň není možné všechny požadované kombinace fyzicky realizovat.
Cílem práce je tedy navržení frameworku, který zoptimalizuje pomocí kombinatorického testování (CIT) testovací proces a sníží celkové nároky na HW. Úlohou CIT je najít množinu optimálních HW konfigurací, které zajistí co největší testovací pokrytí.
Vstupem frameworku budou požadavky na hardware, informace o stávajících a požadovaných komponentách, informace o závislostech mezi nimi a podmínkách, které musí být platné pro bezproblémový běh HW stroje.
Hlavním a zásadním výstupem bude optimalizační nástroj pro výběr vhodných testovacích konfigurací. Jeho vstupem budou konfigurace vyvíjeného SW a testovacího prostředí, podmínky a závislosti mezi nimi. Výstupem bude seznam optimálních HW+SW konfigurací k otestování. Dalšími výstupy budou dynamicky generovaný skript pro zvolený CI/CD server, který spustí testovací proces na zvolených konfiguracích, a dále seznam realizovatelných/nedostupných konfigurací.

Seznam doporučené literatury:

Bures, Miroslav & Ahmed, Bestoun. (2017). On The Effectiveness of Combinatorial Interaction Testing: A Case Study. 10.1109/QRS-C.2017.20.
Bestoun, S. Ahmed and Garcia Miraz, Angelo and Zamli, Kamal Z. and Yılmaz, Cemal and Bures, Miroslav and Szeles, Marek (2019) Code-aware combinatorial interaction testing. IET Software . ISSN 1751-8806 (Print) 1751-8814 (Online) Published Online First http://dx.doi.org/10.1049/iet-sen.2018.5315

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Karel Frajták, Ph.D., laboratoř inteligentního testování systémů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **18.09.2020** Termín odevzdání diplomové práce: **21.05.2021**

Platnost zadání diplomové práce: **19.02.2022**

Ing. Karel Frajták, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

Acknowledgements

I would like to thank my thesis supervisor for his support and guidance through this work. I highly value his insight he provided me with while tackling the theoretical and practical challenges needed to be overcome during the preparation and implementation of the SysCCIT framework. Secondly I would like to thank NIST organization for granting free access to ACTS tool, which is used as a basis of this work.

Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 21. května 2021

Abstract

Techniques of combinatorial interaction testing with constraints have been widely adopted by industries with the aim to lover the time and labor costs of testing of software while maintaining the desired level of quality.

This work proposes another exploitation of these techniques in the are of testing of interactions of software applications with complex dependencies on and interactions with the underlying HW they are being run on — such as operating systems, anti-malware applications, device drivers, virtualization tools and others.

Firstly an overview of current possibilities in the area of combinatorial interaction testing is given, Available systems for combinatorial generation of test inputs combinations (with constraints) are discussed together with algorithms used by combinatorial generators.

Then we discuss the options and requirements for creation of real-world usable optimization framework for generation of testing system combinations with respect to restricted resources for building of physical testing computer systems in HW labs when testing the complete interactions of the whole HW and SW system stack.

The core of this work is in introduction of a new SysCCIT framework implementing a modular system for selecting the best available HW configurations to be built from an inventory of HW components in testing laboratory, taking into account requirements and dependencies of tested application on given HW. Also its performance and usability attributes are measured and discussed in terms of the required time to produce an output for different hypothetical HW component inventory sets and sizes.

Finally, we discuss possible extensions to the implemented framework as well as options and areas for performance improvements.

Keywords: ACTS, CIT, software testing, combinatorial interaction testing, optimization

Supervisor: Ing. Karel Frajták, Ph.D.
FEE CTU, Karlovo náměstí 13, Praha 2

Abstrakt

Techniky kombinatorického testování s podmínkami jsou široce adoptovány průmyslem s cílem snížit cenu a pracnost systematického testování software při současném zajištění úrovně kvality.

Tato práce navrhuje další použití těchto technik v oblasti testování interakcí software silně závislého na konfiguraci systému, na kterém daný software běží — jako jsou operační systémy, anti-malware aplikace, ovladače zařízení, virtualizační nástroje a další.

Nejprve je uveden přehled současných možností použití technik kombinatorického testování, dostupných systémů pro kombinatorické generování testovacích kombinací s podmínkami, společně s dostupnými algoritmy pro kombinatorické generátory.

Následně jsou diskutovány možnosti a požadavky na vytvoření reálně použitelného optimalizačního systému pro generování testovacích kombinací s ohledem na omezené testovací prostředky pro sestavování fyzických testovacích systémů při testování celých konfigurací HW a s ním silně interagujícím SW.

Jádro práce tvoří představení implementace frameworku SysCCIT implementujícího modulární systém pro určení nejlepších dostupných HW konfigurací k sestavení z dostupného inventáře komponent testovací laboratoře beroucí v potaz požadavky a závislosti testované aplikace na daném HW. Také je změřena výkonnost představeného řešení z pohledu času potřebného ke spracování vstupu a vygenerování výsledků pro různě rozsáhlé hypotetické inventáře HW komponent.

Nakonec jsou diskutovány možnosti rozšíření vytvořeného frameworku a možnosti pro zlepšení jeho výkonnosti.

Klíčová slova: ACTS, CIT, testování software, kombinatorické testování interakcí, optimalizace

Překlad názvu: Framework pro zkvalitnění testování softwaru silně závislého na hardwarové konfiguraci a konfiguraci OS

Contents

1 Introduction **1**
1.1 Objectives and thesis organization 1
1.2 Motivation 1

2 About Combinatorial Interaction Testing **3**
2.1 The basis of CIT 3
2.2 The Concept of Covering Arrays . 3
2.3 Mixed CIT and the introduction of constraints 3
2.4 Test suite generation strategies, algorithms 4
2.5 Available CIT generators 5
2.5.1 PICT 5
2.5.2 ACTS 5
2.5.3 Avocado framework CCIT plugin 5
2.5.4 CAGEN 5

3 Problem analysis and solution requirements **7**
3.1 Problem statement 7
3.2 Context 7
3.3 Inputs 7
3.4 Selecting best systems to assemble 8
3.5 Outputs 9
3.6 Expected usage patterns, other attributes 10

4 SysCCIT framework **11**
4.1 Architecture 11
4.1.1 Overview, architectural goals 11
4.1.2 SysCCIT core 12
4.1.3 CCIT Generator backend . . . 12
4.1.4 Parameter providers 13
4.1.5 Rating of configurations 14
4.1.6 Results processing and output 15
4.2 Implementation details 17
4.2.1 Runtime Environment and Dependencies 17
4.2.2 The object model 17
4.2.3 Parameter and Constraint primitives 18
4.2.4 ACTS formats handling 18
4.2.5 Modeling HW components and their relationships 19
4.2.6 Usage of CCIT generator backend 20
4.3 Deploying and using SysCCIT . . 21
4.4 Performance and usability 21
4.4.1 Benchmarking methodology . 21
4.4.2 Results 24
4.5 Possible future expansion and improvements 29

5 Discussion and conclusion **31**

Bibliography **33**

A List of acronyms **37**

B SysCCIT framework complete component diagram **39**

C SysCCIT framework complete class hierarchy diagram **41**

Figures

4.1 SysCCIT framework architecture overview 11
4.2 SysCCIT framework classes and relationships overview 18
4.3 HW components inventory database schema 20
4.4 Time to complete whole SysCCIT.run() invocation for different hw inventory inputs 24
4.5 Time to load all input data for different hw inventory inputs 25
4.6 Time to generate base configurations for different hw inventory inputs 25
4.7 Time to generate complete configurations for different hw inventory inputs 27
4.8 Time to rate generated complete configurations for different hw inventory inputs 28
4.9 Time to select final systems to build from rated complete configurations for different hw inventory inputs 28
4.10 Time to process results for different hw inventory inputs 29

B.1 Complete SysCCIT framework implementation component diagram in bigger more readable edition . . . 40

C.1 Complete SysCCIT framework implementation class diagram in bigger more readable edition pt. 1/2 42
C.2 Complete SysCCIT framework implementation class diagram in bigger more readable edition pt. 2/2 43

Tables

4.1 HW inventory inputs for benchmarking 22
4.2 Base configuration numbers and final system numbers produced by HW inventory inputs for benchmarking 23

Chapter 1

Introduction

1.1 Objectives and thesis organization

First a general motivation behind this work is given.

Then follows an overview of the current CCIT techniques landscape.

Thirdly we discuss options and requirements for creation of an automated optimization framework for generation of test combinations using (constrained) interaction testing techniques and algorithms while taking into considerations the practical limits in real world test environments and resources available for testing efforts.

Then in the fourth part an architecture and implementation of the implemented framework is described, together with performance evaluation and possible improvements. The benefits such a tool can bring to the modern SDLC include better insight into test coverage, increased quality with given resources and speed-up of the testing process.

Lastly, achieved goals and results of this work are discussed as well as possible extensions and improvements.

1.2 Motivation

Nowadays the problem of software testing might seem largely as the issue of ensuring the correct functioning of the project's business logic. However, despite the successful adoption of virtualization and containerization technology in the last two decades (as discussed i.e. by [28]), the problem of software interaction with the underlying OS+HW stack has not received as much attention as pure software or pure hardware testing.

Many applications today rely more than ever on the interaction of software and hardware. Those interactions can facilitate benefits such as deeper computation optimizations (usage of specialized instructions available in only a subset of utilized platforms), power optimizations (to reduce operational costs and or improve battery life of mobile devices), better security and runtime isolation (through virtualization/containerization) and others.

Let's consider Windows operating system ecosystem. Not only is Microsoft dealing with the consequences of their well known pledge to keep backwards

compatibility — the Windows family of operating systems has been present pretty much everywhere from consumer devices across key business operations to world's most critical infrastructure — they are also dealing with the explosion of different device configurations they (together with vendors of those devices) need to support.

Currently, Microsoft publishes updates for their ecosystem in rapid fashion. Device drivers can be updated as soon as those become qualified under numerous Microsoft programs, core products can receive updates weekly and security updates no slower than monthly. [19]

The big players in the industry are switching to rapid release cycles (such as monthly browser updates by Google/Mozilla, but bi-weekly or even weekly releases are getting common). The industry members have created mechanisms and APIs to shield the developers for their ecosystems from the issues these trends bring to the SDLC (operating systems' task is to abstract away the work with HW and provide consistent usage patterns, and nowadays we as a society are pushing that paradigm further by moving to virtualized environments, effectively also abstracting away the HW details from the OS). Yet many software projects cannot take advantage of those. Anti-malware applications still need to do deep kernel hooks to provide their core shielding and filtering functionality (often through kernel drivers and usage of undocumented APIs due to the closed-source nature of Microsoft products). High performance computation libraries need to take into account availability of instruction set extensions or completely different architectures of CPUs. Similar challenges apply to the usage of graphics, networking, virtualization, storage and other subsystems and related device drivers provided by modern platforms. These themselves need to handle new challenges coming from security vulnerabilities being discovered. Inevitable bugs are present in the underlying hardware, that cannot be easily fixed/replaced (such as the Meldown and Spectre vulnerabilities of today's superscalar out-of-order executing CPUs).

The challenges mentioned above lead to the need of rapid (re)testing of new software on as many different hw+sw combinations as possible to assure targeted levels of software quality and compliance. The need to use real world hardware for these tests (i.e. virtualization drivers cannot be loaded and tested in a virtual machine) results in large time and other engineering costs associated with such testing effort. Also the combinatorial explosion of the many parameters of the systems interacting with each other makes it practically infeasible to perform anything close to exhaustive testing.

Chapter 2

About Combinatorial Interaction Testing

2.1 The basis of CIT

CIT — Combinatorial interaction testing, is a technique for testing complex systems with many input and configuration parameters. As presented by works of R. Kuhn et. al. ([16], [15]), it servers as a means to reduce the total amount of test cases needed for ensuring a certain test coverage of system under test, while reducing requirements for testing resources and therefore making the testing effort feasible and reducing costs [20].

Also known as ıt-way testing, the technique relies on the empirical findings that interaction between only a subset of input and/or configuration parameters of given SUT — system under test — still carries a high chance of exhibiting defects [13], [1]. These findings have also been experimentally verified in pseudo-synthetic experiments on smaller software systems, by systematically injecting defects into otherwise functional code and comparing the defect discovery efficacy when subjected to test suites generated by CIT approach with different t-strengths [6] (where ıt is the interaction strength, meaning number of simultaneously interacting parameters).

2.2 The Concept of Covering Arrays

Generally credited to AT&T's mathematician Sloane [25] and later refined in [17], [26] a covering array CA is a mathematical object used to construct and represent test cases. CA is defined as $CA(N; t; k; v)$, and it is constructed as an $N \times k$ array on v values such that every $N \times t$ sub-array comprises all ordered subsets from the v values of size t at least once.

2.3 Mixed CIT and the introduction of constraints

In practice, input parameters can have different number of values, so a so-called Mixed Covering Array MCA can be used. A $MCA(N; t; v_1, v_2, \ldots, v_k)$ is an $N \times k$ array on v values, where the rows cover each $N \times t$ sub-array and all t interactions of values from the t columns occur at least once.

Further, constraints are introduced to suppress combinations of inputs and/or configurations that are infeasible, make no sense or we are not interested in them (for example a combination calling for running a Windows-only application on UNIX OS). Then we talk about Constrained CIT with the usage of Constrained CA [9], [12].

2.4 Test suite generation strategies, algorithms

Several categories of algorithms are popular. In [14] the authors list the major ones as:

- algebraic techniques
- greedy algorithms
- heuristic searches
- constraint satisfaction problem solvers

The different strategies are not exclusive. They can be mixed to leverage trade-offs that need to be considered for the particular system under test, with regards to the size and nature of inputs and constraints.

For some special situations, algebraic techniques and constraint satisfaction solvers are known to provide optimal solutions in good time. Otherwise, greedy algorithms can be reasonably accurate.

However, with increasing input sizes and additions of seeding/constraints, heuristics are needed to find good enough solutions in reasonable time.

Research activities into better heuristic algorithms have produced the following t-way test suite generation strategies (in no particular order):

- General strategy [18]
- Simulated annealing-based strategies [7]
- Forbidden tuples-based searching [10]
- Genetic algorithms [24]
- Ant colony-based algorithms [24]
- Particle swarm optimization [2]
- Harmony search [4]
- Cuckoo search [1]
- Bat-inspired algorithm [3]
- Bees-inspired algorithm [21]
- (Adaptive) teaching learning-based optimization [22], [30]
- Fuzzy logic based meta-heuristics [30]

2.5 Available CIT generators

2.5.1 PICT

PICT is a primarily pair-wise (meaning interaction strength $t = 2$) test generation tool and algorithm developed by Jacek Czerwonka, Microsoft. As he describes in [8], the tool has been designed to bring the pair-wise testing strategies to the level of real-world usefulness required by real-world projects in the industry.

It supports mixed-strength generation and application of constraints, which is basically a must for successfully generating usable test scenarios for any bigger system.

PICT is publicly available both for Windows and Linux environments and comes free of charge.

2.5.2 ACTS

Advanced Combinatorial Testing System (or ACTS) is a NIST-funded combinatorial test generation tool [29] based on multiple variants of the so called IPOG algorithm [18]. It is implemented in Java and provides GUI, CLI and Java API, making it possible to integrate the tool to more complex and/or special-purpose projects.

The ACTS has been around for a while and has become a somewhat standard baseline when benchmarking speed, resource requirements and resulting test suite size of different combinatorial generation algorithms and tools. For this reason, we opted to use this tool as the basis of this work.

ACTS is also publicly available, but is distributed only upon request to the NIST organization, which made it available for us to use in this work.

2.5.3 Avocado framework CCIT plugin

In [23], [11] the authors implemented a CIT varianter with combination of greedy and meta-heuristic techniques as follows. First the whole search space is generated, then the forbidden tuples according to constraints are pruned.

Next, a Monte-Carlo style searching algorithm do derive new rules from existing constraints and then simplify the working solution. By doing so, the constraint satisfaction problem is elegantly side-stepped, in exchange for steep memory requirements increases with the number of inputs.

On machine with 16 GB RAM, it was usable only for number of parameters roughly under 140 for $4 - way$ testing and much less with increased combinatorial strength.

2.5.4 CAGEN

Very recent addition to combinatorial generation tools is CAGEN by M. Wagner et al. [27]. It support some of the same CIT generation algorithms

and constraint handling techniques as ACTS does. It also aims to keep the interoperability with ACTS and so supports ACTS style input and output formats.

CAGEN authors claim its performance is superior to the ACTS and among the carried out optimizations as a switch to compiled language — Rust. Also, the implemented algorithms are templatized according to the desired interaction strength t which allows them to instantiate different versions of the algorithms at compile time and allow compiler to make optimizations using the constant value of interaction strength, which results in significant performance uplift compared to the Java based ACTS.

CLI is provided for integration with other projects upon request. Besides that a Web application is provided as a GUI front end and also serves as local browser-based generator, where the computations are done client-side through the use of Rust code compiled to WebAssembly, which s a standard format of binary executable code supported by all major Web browsers.

Chapter 3

Problem analysis and solution requirements

3.1 Problem statement

The problem at hand can be worded as follows:

Given a set of hw components, application inputs/configuration options and dependencies, what kind of systems should be built from these so that best possible coverage is achieved when testing the application on such systems?

3.2 Context

Testing on real-world hardware systems can achieve various states in today's organizations. Some undergo a lot of effort and planning and have testing coverage well defined and testing environments well built. For such organizations, a lack of information about coverage of testing on hw is not really a threat, but a solution to the problem stated in previous section could still be used as a validation or micro-optimization tool. However, there are also other organizations, where the testing of complete systems is not a first class citizen in the full test and QA process of SDLC. For those organizations, many reasons could exist, but one of the main is usually lack of resources both in terms of available testing hw and lab space as well as in lack of engineering time.

This work is trying to solve that problem in one such company, where hw testing laboratory is being maintained mostly by enthusiast employees and the available hw components mainly consist of old decommissioned systems previously used by the company employees, plus small number of newer generation components that are bought usually because of incidents.

3.3 Inputs

To know what components are available for building test systems, an inventory is needed. Such inventories can be created and maintained by hand in case of very small numbers of components, but usually can be collected automatically, especially if some form of automated test running on hw systems is already

in place. Having an inventory accessible programmatically means we can design a solution that automatically pulls all required information about hw components without any need to specify it manually.

For this work we will design the mechanism for pulling data about hw components from simple local database, that stores available components by type and relationships between them. It is crucial though to allow for easy replacement of this input gathering method to facilitate reuse and extension in other environments.

The second input is a list of application's inputs and configuration parameters and their values and inter-dependencies, as well as dependencies on the underlying hw components. Again, these could be inferred from a formal model of given application, or they could be automatically generated from source code, documentation, or another artifact of SDLC. It is also common that such inputs already exist from previous efforts to test the application itself using CIT techniques.

As the focus for this work is mainly on the processing of hardware components to build suitable systems, for software model input we will only operate with an existing input format common for CIT tools, consisting of application parameters, values and constraints stored in ACTS-style file format. Again though we will keep in mind the need to provide easy way to extend our work to accept and/or automatically generate this input differently.

The input data as a whole needs to contain information how to build a system to run tests on. For that reason, the inputs are split to hw components information that defines the base system, and the sw model. Together, single permutation of possible parameter-value pairs forms one test configuration and so one complete system under test.

3.4 Selecting best systems to assemble

Because generating of all possible configurations to select from would in real world be impractical or impossible due to the sheer size of the search space, we will employ CCIT technique to generate only combinations covering all $t - way$ interactions for the base systems and $t' - way$ interactions for complete test systems.

The main procedure in the core of the solution consists of the following steps:

1. from inventory of hw components construct set of parameters, values and contraints modeling the base hw system (basically a working computer) - aka "hw params" set

2. load and parse the application model and also construct a set of corresponding parameters, their values and constraints

3. using CCIT generator backend, generate all base configurations with t-way coverage from the hw param set

4. for each of the base configuration generated, construct constraints such only that base configuration would be generated, and merge the model with the application model, then again using the CCIT generator backend generate all complete test configurations with $t' - way$ coverage ($t' >= t$) applicable for the given base system and save it as a potential complete test configuration associated with it's base configuration

5. rate all base configurations by the associated potential complete test configurations using provided rating implementation and sort them best to worst, producing a list of rated systems

6. go through the ordered list of rated systems and for each one allocate components to the computer build as described by the base configuration. If a component from the base configuration is already used, given base configuration cannot be realized, so it is skipped. This produces a list of test systems, whose base configurations can all be assembled with respect to the given hw inventory.

7. submit the result to various results consumer components to be stored, reported, to infer uncovered parameter values and to generate CI/CD test invocation script.

To produce a rating of given complete test system configuration different strategies could be designed. In this work we will simply count all unique $t' - way$ combinations of parameters covered by it, as in general there is more value in having more combinations covered. However it could be also argued some particular combinations could be preferred or even required in each complete test system configuration, base on various reasons, such as being more prevalent in the user base, being something that is guaranteed by the company as always tested and certified or being identified as common area of problems. For that reason, a concept of rating service with defined interface is used, where different organizations could implement their own rating service taking into account the desired information when producing a rating of one particular complete test system.

3.5 Outputs

Once it is established which systems should be built, we relay that information to the output. In it's simplest form we at least need to know which components make up the particular base systems. It is also useful to know what are the complete hw+sw configurations that are to be run given particular base system.

On the other hand we would also like to know which particular parameter values could not be covered either because of lack of resources or because inter-dependencies and constraints on input parameter values do no allow us to cover them (for example a particular operating system is not supported by any of the selected base configurations, we are missing a motherboard

compatible with requested cpu architecture, particular inputs/configuration variation requires specific device present, that is already used in higher ranking base system, etc.).

It is also desired to produce a generated script that would invoke tests on the selected base systems and produced complete system configurations using available CI/CD server.

Using dependency injection, we can provide multiple different results consuming components that implement each of these needs, achieving modularity, concern separation, future extensibility and reuse.

3.6 Expected usage patterns, other attributes

Because it takes some time to build once selected testing computer, it is not expected the solution would have to be searched for too often, certainly not more than once in single iteration of the SDLC. For that reason it is also not expected single run of the solution routine would have too strict performance requirements under the order of minutes for some real-world size hw lab inventory.

However it is expected the requirements on access to the hw inventory, as well as on the source and format of application model can change significantly. Also different CCIT generator backend may be preferred for performance/features/licensing/pricing reasons. As discussed already, organization might have different ideas about what actually constitutes rating of particular test system configuration and might have different requirements for the further processing of solution results.

For all of these reasons, main focus during development will be to design and implement a solution framework with suitably modular architecture to enable future extensions and reuse. Less focus will be placed on performance, while keeping the solution usable in real world scenarios.

Chapter 4

SysCCIT framework

This chapter describes the architectural decisions and implementation of optimization framework called **SysCCIT** that uses CCIT generator backend ACTS for the actual generation of t-way covering test system configurations.

4.1 Architecture

4.1.1 Overview, architectural goals

As discussed in chapter 3, main focus when defining the architecture of SysCCIT is placed on modularity due to the expected need to provide custom modules to adapt the framework to target environment and input and output requirements. A good way to achieve great modularity is to cautiously separate different concerns to different parts of the system and use well-defined interfaces to facilitate interoperability and easy component integration.

The complete architecture overview is depicted in figure 4.1.

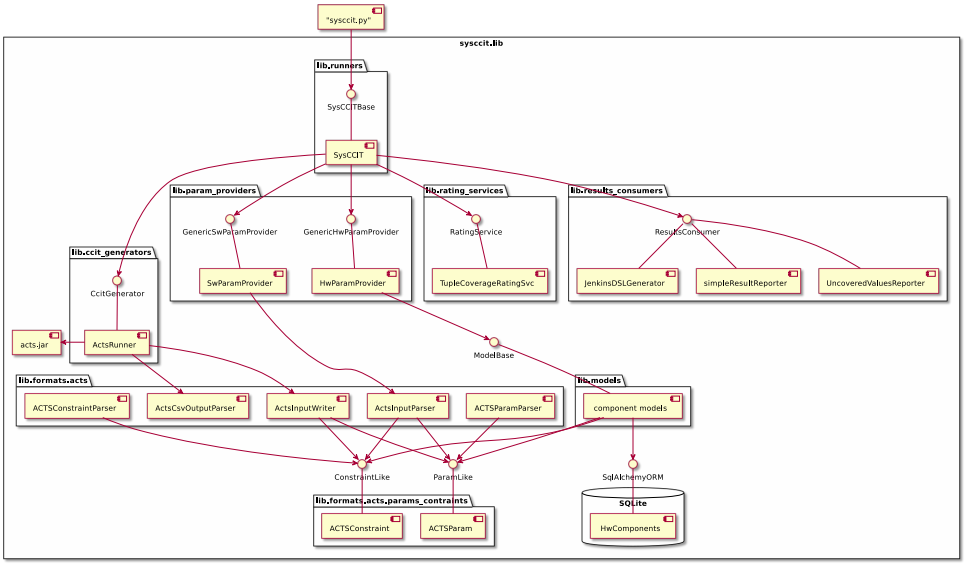


Figure 4.1: SysCCIT framework architecture overview

4.1.2 SysCCIT core

The core of the framework is the *SysCCIT* component implementing the *SysCCITBase* interface. The interface defines all required components needed to construct an implementing component to perform the core functionality of the framework, and forces the use of dependency injection during instance construction, so that any implementation of it, such us the *SysCCIT* component, has to work with defined subsystems and makes the dependencies explicit.

It is the job of the core of the framework implementing the *SysCCITBase* interface to drive the solving process. In this case the *SysCCIT* component implements the algorithm listed in chapter 3. If different algorithm is one day proposed, a different core component implementing the *SysCCITBase* interface would be needed, though only small changes (if any) would be required for the injected dependencies.

4.1.3 CCIT Generator backend

To perform the generation of t-way combinations of base system and complete test system configurations, a CCIT generator backend is needed. The general requirements for this backend are defined by the *CcitGenerator* interface.

In general, all CCIT generators on input expect the desired interaction strength value and some form of list of parameters, their possible values, and constraints between them, plus options specific to particular CCIT generator implementation. On output they provide some form of listing of generated combinations covering the input with desired interaction strength.

It is the job of particular implementation of the *CcitGenerator* interface to provide this functionality and expose it to the framework core.

In this work, we use the services of CCIT generator ACTS discussed in chapter 2 among available CCIT tools. The component *ActsRunner* implements the *CcitGenerator* interface by wrapping the ACTS implementation in *acts.jar*, which is a java based complete implementation of the ACTS framework with CLI interface.

The obvious benefit is we do not need to implement our own CCIT generator, which would be non-trivial amount of work outside of the scope of this thesis. Next benefit is the tool has become quite popular and it's input and output formats have became de-facto standards among CCIT generators, so if better tool is made available, it should not be too much of a work to replace the backend.

Among negatives of this chosen approach is the fact we do not have much control over how the generator operates and cannot really improve it's performance or other attributes. It also always only work on single input model, so for our use case, where we iteratively generate complete test suite configurations for different system base configurations, we need to invoke the acts.jar separately. This brings some performance losses due to repeated process (and java JVM) starts and exits.

On top of the redundant process restarts, the CLI works with input and output files only, which brings possible performance bottleneck in the form of I/O operations and unwanted repeated writes to the underlying storage medium.

4.1.4 Parameter providers

As the chosen CCIT technique relies on working with configuration/input parameters and constraints, it is necessary to have a way of converting inputs in the form of hw inventory and sw model into some common abstraction of the CCIT input parameters.

It is expected the different parameter providers will be crated as an adaptation form the world of databases and file records to the world of SysCCIT framework.

Parameter model

Common interfaces *ParamLike* and *ConstraintLike* are defined to facilitate interoperability and common language to be used by different so called *Parameter Providers* to feed the framework with data to operate on.

A given parameter provider implementation is responsible for loading the data into the SysCCIT framework instance and parsing and decoding the data plus internal relationships and dependencies into set of *ParamLike* and *ConstraintLike* objects that represent the parameter name, values and constraints used by *CcitGenerator*.

For the framework to be able to distinguish between parameters representing the scarce resources whose usage we want to optimize and the parameters coming from the sw model, combinations of which we can test as many of as we want without the need for human intervention (in general, OS deployment and application installation can be done fully automatically, but an engineer needs to go an swap components by hand), implementations of two distinct interfaces are required by the SysCCIT framework. *GenericHwParamProvider* is implemented by component sourcing data about the limited hw inventory and *GenericSwParamProvider* implementation is responsible for providing the representation of model of the sw application under test.

Working with HW inventory

In this work we are using an SQLite relational database to represent the hw component inventory. Each component type like central processing unit or motherboard is represented by a table of particular component instances. Some component types can have more attributes important for the complete model of the base system, like a processor micro-architecture, chipset type of network card and so on.

Dependencies and relationships between the components and attributes are stored via association tables through many-to-one and many-to-many relationships.

It is then the responsibility of *HwParamProvider* component implementing the *GenericHwParamProvider* interface to be able to connect to the database, read the data and decode the relationships into set of parameters, values and constraints in such a way, that CCIT generator produces for such input a set of configurations, that are valid and can be built (we do want results that tell us to use for example an Intel processor in it's compatible motherboard, to connect a Sata drive to a Sata controller and connector equipped motherboard and so on).

To be able to do so, the HwParamProvider works with explicit abstractions of computer components, that map to records in the database via SQLAlchemy ORM. Also, each of these abstractions implements a *ModelBase* interface so that a way to transform these to *ParamLike* and *ConstaintLike* implementing objects is unified. This together with the usage of ORM mechanism makes it very easy to add more component types to the database and make them available to the *HwParamProvider* component.

Working with SW model

The *GenericSwParamProvider* interface is implemented by *SwParamProvider* component. As was mentioned in the chapter about problem analysis and solution requirements, in this work we use an ACTS-style file format to record the sw application model and pass that as the sw model input to the SysCCIT framework.

The *SwParamProvider* component uses an Acts file format parser component to transform this input to internal representations implementing the *ParamLike* and *ConstraintLike* interfaces. This ensures we can easily merge the parameters and constraints for the hw configurations as well as the application model when that is needed during the runtime of the core routine of the framework.

4.1.5 Rating of configurations

To select best base hw configuration candidates, the core of the framework needs to be able to evaluate how good each generated base configuration is. To do so, we need to be able to measure that property in some way and assign a *Rating* based on that.

To make it possible to define custom ratings, an interface *Rating* is defined. The only attribute it enforces is that each pair of objects of the same type implementing it are comparable via less-then-or-equal magic method used by the python builtin sort() method.

It is then the job of another required component by the framework implementing *RatingService* interface to know how to actually produce these ratings given all generated *System* objects (base configurations and associated complete system configurations).

In this work we implemented a *TupleCoverageRatingSvc* that exposes the *RatingService* interface. It assigns *IntRating* to *System* objects making them

into *RatedSystem* objects, based on the total number of unique $t-tuples$ of parameters covered by the complete system configurations in each System.

It is also possible to specify which particular parameters should be ignored for this rating. That is useful when some parameters are needed to inform the model about what constitutes valid base configuration that would make up functioning system through specifying some constraints on parameters, but the value of such parameter has in reality nothing to do with improving system combinations coverage, and so we do not want it to influence the rating.

4.1.6 Results processing and output

From the specification and problem analysis in chapter 3 the main outputs we care about are:

- list of base configurations representing actual base systems to be built from available hw components inventory, together with associated complete system test combinations to be run on given base system
- report of which configuration options could not be covered
- generated script to invoke testing process in CI/CD pipeline

As also discussed in the analysis, it is quite likely for the output requirements to change to adapt to changes in environments where the tool is to be used and also to respond to new requirements on further information extracted from the solution process.

To help with that, the framework core accepts a list of different services all implementing the *ResultsConsumer* interface. It describes how to pass found results to further processing by each *ResultsConsumer* service to produce desired output artifacts and again can be easily extended by implementing new components bases on this interface. In this work three services were implemented satisfying the output requirements.

Recommended HW configurations

The *SimpleResultsReporter* component is responsible for reporting the list of base configurations representing actual base systems we should build using the components in our hw inventory, plus the associated complete system test combinations we shall run on each base system.

The inner workings are very simple. The service accepts list of RatedSystem objects, transforms the data into json format and saves it to specified file as the main output artifact.

Unused/Uncovered inputs

The *UncoveredValuesReporter* has the job of reporting which configuration options are not covered by the solution result in the form of list of RatedSystem objects. It does so by building an inventory of every parameter-name and

value pair that is available from parameter providers to the framework, and records all that are not present in the list of results from the original problem solution.

It must be noted this component does not report t-way combinations, just the parameter name and value pairs. The reason is to report all uncovered t-way combinations we would first need to generate al of the t-way combinations based on the parameters available, ignoring all the constraints. But that would give us a lot of nonsensical results and not much useful information. The way the component does it via reporting of the uncovered values gives us an important and only useful information about the coverage gaps in our testing possibilities. If for example we ask for testing on cpu with micro-architecture x, but no motherboard supporting such cpu exists in our inventory, we will get a listing of CpuUarch_x not covered. That is much more useful than getting a listing of all t-way combinations including this cpu micro-architecture for identifying what is the inventory gap and how to cover it.

CI/CD integration

Today's CI/CD systems usually define some form of domain specific language that can be used to configure build + test + release processes in unified and documented fashion. It is the responsibility of component *JenkinsDSLGenerator* to generate so called *declarative pipeline* script for Jenkins CI/CD server environment, that defines a Jenkins job to run tests using the generated test configurations suites for each selected base configuration.

Sadly we have not been able to access the target environment to provide tight integration for a particular organization. For that reason, as each organization has unique CI/CD environment details, we were not able to provide fully functional solution for the CI/CD script requirement. The *JenkinsDSLGenerator* component in its current form only generates a generalized version of such DSL script from a template, that would need to be extended with the organization's particular CI/CD environment details to actually run the tests.

Nevertheless, as it stands the resulting artifact creates a Jenkins matrix-style job that takes among inputs a json file containing association of base configuration id with the target complete test configurations to be run against it and invokes each combination of the job matrix. To achieve fully functional member of the CI/CD pipeline at the bare minimum a way to get actual tests and application under test installed would need to be added, as well as some management and claiming/unclaiming of particular hw base configuration for test through a management interface of the laboratory, where built base configurations are located.

4.2 Implementation details

4.2.1 Runtime Environment and Dependencies

The *SysCCIT framework* is written predominantly in Python 3, targeting release version 3.8 and higher due to extensive use of Python's dataclasses mechanism and type hinting.

To communicate with database ORM - object relational mapping is used, through the Python package *sqlalchemy*, which enabled us to implicitly support any kind of relational database engine and instance. Adding support to new HW component types in HW component inventory is made very straightforward - one only needs to subclass an sqlachelmy's Base class, then define which properties is the new component made up from, what relationships do these properties have to the rest of the components (so that we still can reason about what make a buildable system configuration) and then define the *ModelBase* interface methods, so that CCIT parameters and constraints can be constructed from instances of given component.

All python related requirements can be installed via Python's packaging system *Pip* using provided standard *requirements.txt* file.

As the CCIT generator backend is using tool called ACTS implemented in Java and distributed as Java's JAR runnable package, the SysCCIT framework through the ActsRunner backend depends on Java environment being available. The used acts.jar release is version 3.2 and it was used through OpenJRE flavor of Java version 11.

The development was done on Linux operating system Debian, release code name Buster. Thanks to the nature of Python and Java being interpreted languages with runtime releases available for all major platform, there should be nothing in the way of using it on other platforms like Windows, OpenBSD or cloud environments, but only Linux Debian, Ubuntu and Gentoo distributions have been tested so far.

4.2.2 The object model

The classes that actually implement the various framework functions adhere to the architectural model described in section 4.1. The overview is given in figure 4.2 and for more readable version refer to diagram in appendix C.

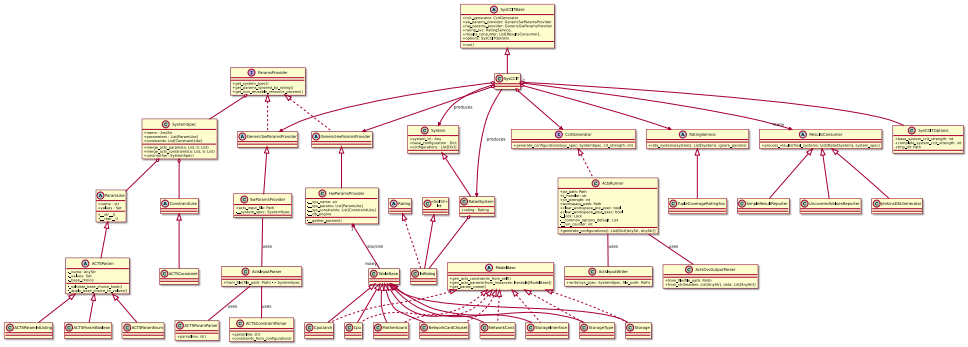


Figure 4.2: SysCCIT framework classes and relationships overview

4.2.3 Parameter and Constraint primitives

Aside from the implementations of various components, the framework relies heavily on the representation of parameters and constraints implementing the *ParamLike* and *ConstraintLike* interfaces.

To support interoperability between different CCIT generators we decided to make use of an existing schema provided by the ACTS (which has gained notoriety and became a sort of standard) and we implemented parameter types *ACTSParamEnum*, *ACTSParamIntListing* and *ACTSParamBoolean*. These types are responsible for representing the parameter name, values and other details as prescribed by the common input format used by ACTS and other generators to hold enumerated (read string) parameters, integers (or number generally actually) and boolean flags.

The same goes for constraint type *ACTSConstraint*. We only needed to work with a string representation of constraints so an *ACTSConstraint* object is simply constructed from string value. A formal specification of the constraint definition schema is provided by ACTS and it might be needed to fully implement it in case of more advanced rating implementations to provide further insight into quality of given parameter combination.

4.2.4 ACTS formats handling

The ACTSParam* and ACTSConstraint types map onto the acts-style input file format. Not only do we need to be able to serialize the objects to this format to provide it as an input to acts.jar generator in *ActsRunner* implementation, but we also need to be able to parse it as the SwParamProvider expects the application model in this exact format.

Listing 4.1: ACTS style input file formatg

```
1 --- comments and ignored lines are prefixed with tripple-
     dash
2 [System]
3 Name: <required system name>
4 [Parameter]
5 --- parameters and values listing section
6 OS (enum) : win_10, win_8_1, win_7, win_xp, ubuntu_20_10
```

```
7  Feature_A_Installed (boolean) : TRUE, FALSE
8  min_size_GB (int) : 8, 16, 64
9
10 [Constraint]
11 --- constraints listing section
12 --- boolean statements on single lines, a new-line
       character signifies a logical "AND" relation
13 --- values of enum-type parameters always must be double
       quoted
14 OS = "win_xp" => ( CpuUarch = "westmere" || CpuUarch = "
       nehalem" )
15 Feature_B_Installed = TRUE => min_size_GB >= 64
```

The *ActsInputParser* class implements class method from_file() to facilitate reading of this file format using *ActsParamParser*, which is a factory that decides which particular type of *ActsParam* to instantiate for each parameter line parsed, and an *ActsConstraintParser*. Writing of the ACTS style input is done by implementation in *ActsInputWriter* implementation.

Then we also need to able to read the results from acts.jar invocations. It outputs CSV files, so a customized wrapper around Python's CSV handling utilities is implemented in *ActsCsvOutputParser*.

4.2.5 Modeling HW components and their relationships

The implementation currently supports the following list of HW components:

- Processors
- Motherboards with single processor socket
- Network interface cards
- Storage devices

To be able to specify which components work together, we specify the dependencies using shared details.

The processors and motherboards define which processor micro-architecture they implement/support. A motherboard can support multiple processor micro-architectures.

Motherboards and Storage devices define, which storage interfaces they support/connect through.

Network interface cards are associated with particular networking chipsets and can be integrated on motherboards.

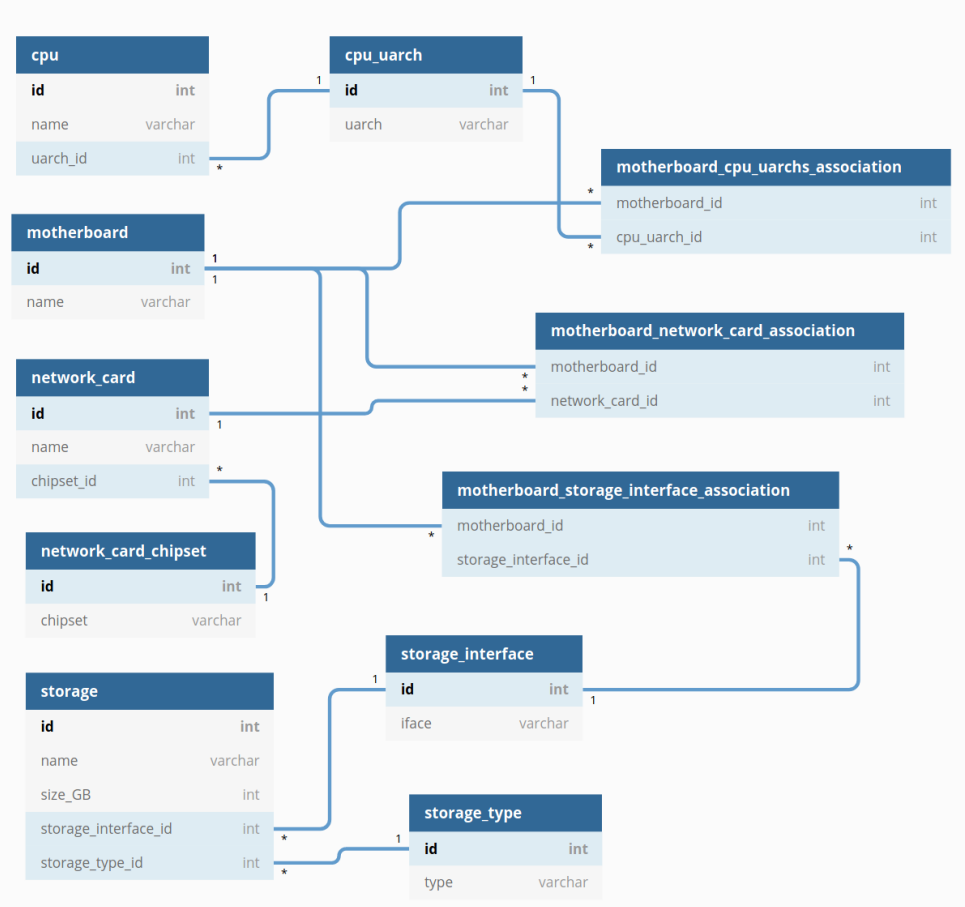


Figure 4.3: HW components inventory database schema

How these components and relationships are stored in the database can be seen from the database schema on figure 4.3. Each component type and component detail corresponds to one table. Each component/detail instance then corresponds to records in respected tables, so each row in cpu table represents particular cpu we have available in our inventory and vice versa for other components.

To model one-to-many and many-to-many relationships, where for example a particular cpu micro-architecture represented by a record in cpu_uarch table is implemented by multiple processors (multiple records in the cpu table) and supported by multiple motherboards, plus a motherboard can support multiple cpu micro-architectures, we use so called association tables, which basically map ids of records from one component type/detail to another through foreign keys.

4.2.6 Usage of CCIT generator backend

The *ActsRunner* implementation of the *CcitGenerator* interface is thread-safe and aware of available multiprocessing environment. As such, the framework tries to schedule as much acts.jar invocations as there are logical processors

available to take advantage of the perfect input-output partitioning of data.

The *ActsRunner* acts like a wrapper around acts.jar and transparently handles serialization of given parameters and constraints to form the input for acts.jar and parses the output csv file back to generated configurations represented as key-value dictionaries.

It is also possible to specify the interaction strength and constraint solving algorithm, though the SysCCIT framework implementation currently only uses default constraint solving algorithm called *forbiddentuples* and also the default CIT generation algorithm *IPOG* is used. Possibility to change the CIT generation algorithm is not currently implemented, but such a change would be trivial.

4.3 Deploying and using SysCCIT

We are not officially allowed to distribute the acts.jar tool, so the SysCCIT framework is not immediately usable out of the box. However the authors of the ACTS tool do distribute it free of charge upon email request.

As discussed in the subsection 4.2.1, Python 3 environment needs to be installed and bootstrapped using provided standard requirements.txt file, and Java Runtime Environments needs to be available to run acts.jar.

Once the user acquires the acts.jar backend, path to the file needs to be specified on the command line to *sysccit.py* script together with the inputs and outputs specifications. The script is a thin wrapper and CLI implementation, that instantiates all the required components, assembles the SysCCIT framework instance and drives the solving process.

4.4 Performance and usability

Because usually the biggest problem when working with combinations of many parameters with many values is usually the inevitable combinatorial explosion of the search space, we ran a simple benchmark to see whether the current implementation of SysCCIT framework's core algorithm using the acts.jar CCIT generator is actually usable in real world.

4.4.1 Benchmarking methodology

We measured the run time of the whole procedure implemented by SysCCIT.run() and also the particular run times of the main components the framework is composed of, against single application model as the sw input side, and five synthetic HW inventories with increasing amount of components and potential buildable base configurations as the hw input side.

This is the used application model in acts-style format:

Listing 4.2: SW model for benchmarking

```
1 [System]
2 Name: SampleSystem
```

```
3
4  [Parameter]
5  OS (enum) : win_10, win_8_1, win_7, win_xp, ubuntu_20_10
6  Feature_A_Installed (boolean) : TRUE, FALSE
7  Feature_B_Installed (boolean) : TRUE, FALSE
8  Feature_Win_Only_Installed (boolean) : TRUE, FALSE
9  Core_Win_Runtime_Installed (boolean): TRUE, FALSE
10 Core_Linux_Runtime_Installed (boolean): TRUE, FALSE
11 min_size_GB (int) : 8, 16, 64
12
13 [Constraint]
14 Feature_A_Installed = TRUE => ( OS = "win_10" || OS = "
      win_7" )
15 OS = "win_xp" => ( CpuUarch = "westmere" || CpuUarch = "
      nehalem" )
16 Feature_Win_Only_Installed => Core_Win_Runtime_Installed
      = TRUE
17 Core_Win_Runtime_Installed = TRUE =>
      Core_Linux_Runtime_Installed = FALSE
18 Core_Linux_Runtime_Installed = TRUE =>
      Core_Win_Runtime_Installed = FALSE
19 Core_Win_Runtime_Installed = TRUE => (OS = "win_10" || OS
       = "win_8_1" || OS = "win_7" || OS = "win_xp")
20 CpuUarch = "zen" || CpuUarch = "zenplus" || CpuUarch = "
      zen2" || CpuUarch = "zen3" => ( OS = "ubuntu_20_10" )
21 Storage_size_GB >= min_size_GB
22 Feature_B_Installed = TRUE => min_size_GB >= 64
```

The important properties of HW inventory inputs used are summarized in table 4.1 and table 4.2. Note the input names as they are used to identify these inputs later in tables and graphs of the benchmark results.

Input ID	#CPUs	#MBs	#NICs	#Disks
tiny	3	2	3	5
small	6	5	8	9
17_17_28_20	17	17	28	20
34_34_42_40	34	34	42	40
51_51_56_50	51	51	56	50

Table 4.1: HW inventory inputs for benchmarking

Using default settings for the framework and the acts.jar generator, the different HW inventory inputs produce the following numbers of generated systems to be evaluated and are selected eventually:

Input ID	#Generated Systems	#Selected Systems
tiny	9	2
small	71	4
17_17_28_20	569	13
34_34_42_40	1597	23
51_51_56_50	3177	36

Table 4.2: Base configuration numbers and final system numbers produced by HW inventory inputs for benchmarking

The important parameters are $t = 2$ for $t - way$ generation of base configuration, $t' = 4$ for $t' - way$ generation of complete configurations (systems), both using acts.jar's default *IPOG* algorithm for generation of combinations and default *forbiddentuples* algorithm to apply the constraints.

The measurement was done on two test systems with mainly different number of logical processors to enable us to investigate potential scaling with logical processors available:

- CPU: Intel Core i5-4690K 4 cores/4 threads @4.0 GHz, RAM: 32 GB DDR3, SSD storage; labeled in tables and graphs as *haswell_4c4t*

- CPU: AMD Ryzen 7 5800X 8 cores/16 threads @4.0 GHz, RAM: 64 GB DDR4, SSD storage; labeled in tables and graphs as *zen3_8c16t*

For HW inventory inputs *tiny*, *small*, *17_17_28_20* and *34_34_42_40* we ran 3 iterations on each system and we list the average from the three iterations for each input and system. For input *51_51_56_50* only single iteration was run on each system because of how long it takes to process.

4.4.2 Results

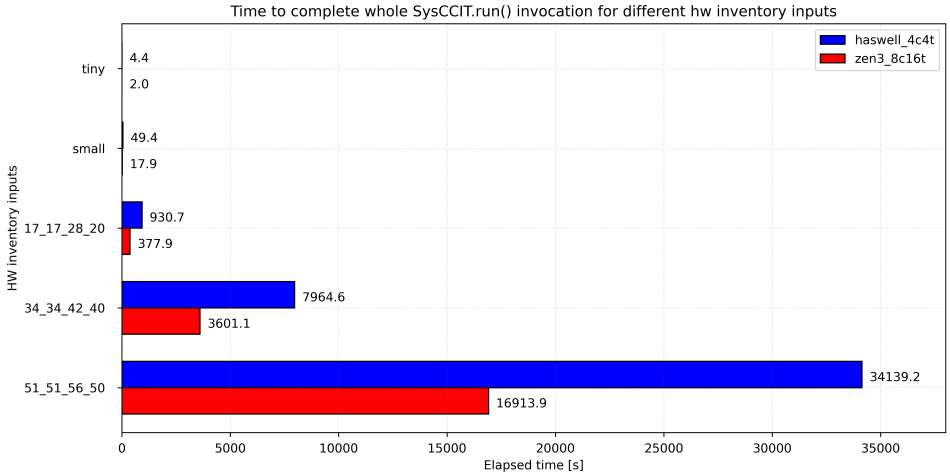


Figure 4.4: Time to complete whole SysCCIT.run() invocation for different hw inventory inputs

Here depicted in figure 4.4 we start with the runtime required to complete the whole procedure, from inputs ingress to having recommended base configurations to be built with complete test configurations associated on the output of the framework.

As we can see from the graph and as is expected for combinatorial problems, the runtime required for completion grows drastically with the size of the input. For the biggest input in the form of HW inventory *51_51_56_50* the required time is almost 10 hours on 4 core 4 thread system and just a bit under half of that for newer 8 core 16 thread system.

The HW inventory size represented by the *51_51_56_50* input could be considered realistic. With respect to discussion in the analysis and requirements part, where we do not expect a need to run the computation on weekly or daily basis, but less often, this is still usable performance, even if not particularly great.

We explore the details of how the particular components contribute to the total runtime in the next figures in this section.

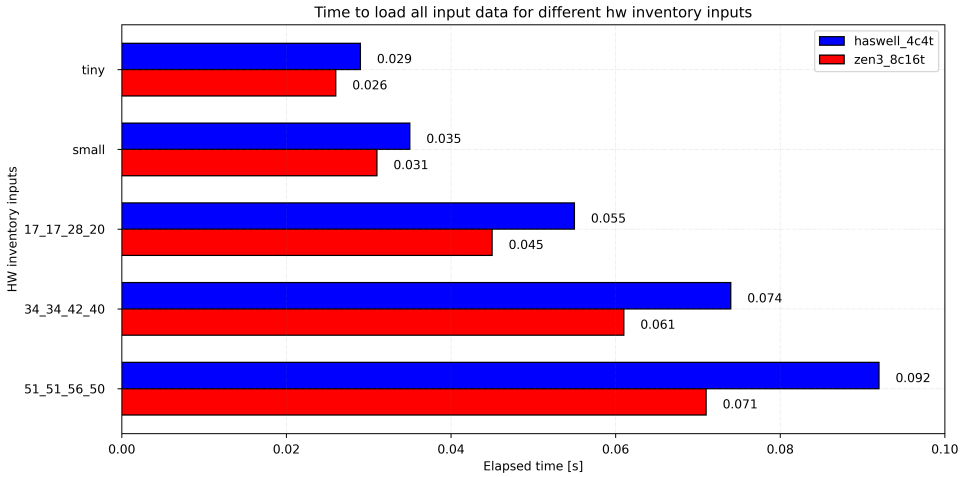


Figure 4.5: Time to load all input data for different hw inventory inputs

The data in figure 4.5 shows the time required to load and parse the inputs to construct parameters and constraints for the CCIT generator do grow with the size of the input, but not in any drastic fashion, actually the trend resembles linearity with respect to total input size, and that is expected.

This shows we have not made any oversights when implementing the *SwParamProvider* and *HwParamProvider* components. Also, the total runtime is so small in contrast to the complete runtime of the framework invocation, that we can effectively neglect it's presence when looking for areas of improvement. It could be made faster by at least parallelizing the ingress of inputs for sw model and hw components, but even that does not seem worth the effort currently.

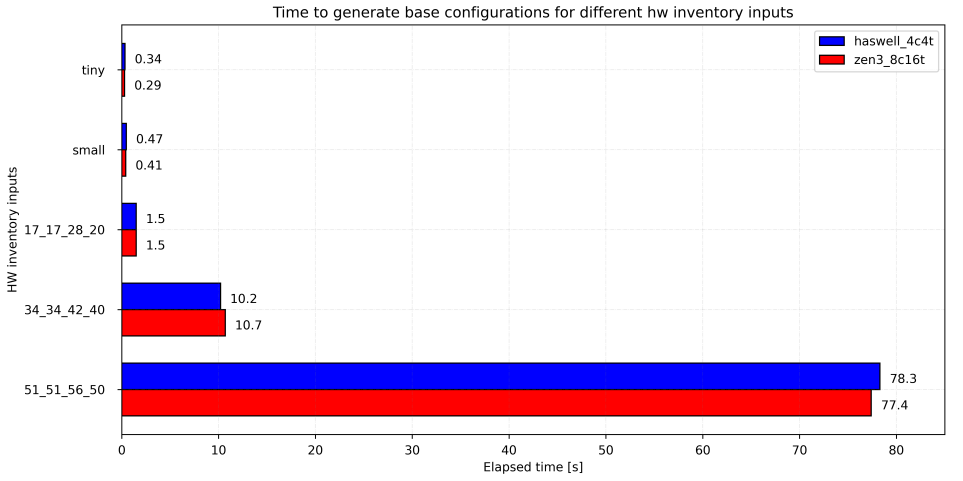


Figure 4.6: Time to generate base configurations for different hw inventory inputs

In figure 4.6 we see the time needed to generate the base configurations from the HW inventory available. Basically this encompasses the serialization of parameters to acts-style text file to form the input to acts.jar, then single invocation of acts.jar, and parsing of the output csv with generated base configurations.

It should be noted that acts.jar in its current implementation is multi-threaded and does try to use all available processors. However, from experience gained from observations made during the development of the SysCCIT framework we can say, it is not able to fully utilize the available computing resources.

From the data we do not observe any actual scaling with available processors, and even the slight general performance advantage of the more modern zen3 cpu micro-architecture does not demonstrate itself.

It could be that for the input sizes we are working with the single run of acts.jar is not able to take advantage of the more resourceful system in terms of available processors, because input parsing to form internal representation of the data and producing the output csv could be non-trivial amount of work in contrast to the work required to find the t-way combinations themselves.

Also we are certainly paying some price for the sub-process start and serialization of data on persistent storage. We are basically starting a Java virtual machine to do a bit of work and then shut it down immediately, but in general that is a very expensive operation, Java applications can perform much better if running for a while, allowing just in time compilation and runtime optimizations to take place for long-running services, which is not our case.

Again though as in the previous case with data ingress, the time spent in this particular step can be very much neglected for the input sizes we are working with in contrast to the total runtime of the whole framework. What we need to keep in mind is, that if we were to significantly increase the input sizes and try to employ even more processors, despite acts.jar being multi-threaded implementation, it might not help, because we see no evidence of performance scaling with number of processors available, so it might become a problem in such case.

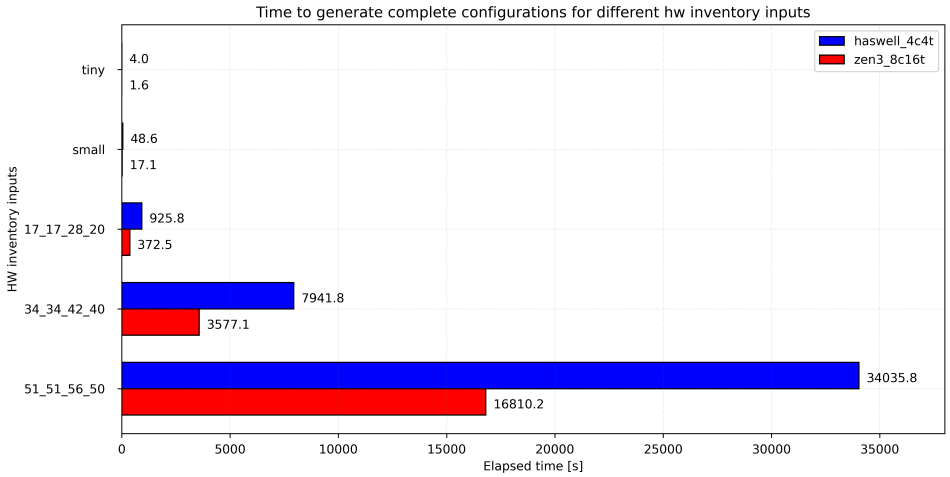


Figure 4.7: Time to generate complete configurations for different hw inventory inputs

The generation of all complete configurations that we later rate and choose from is the main workload the framework has to handle and as seen from the data depicted in figure 4.7 it has by far the biggest impact on the total run time.

The procedure consists of constructing more constraints for the HW inventory model to select particular base configuration and merging it with the parameters and constraints derived form application model inputs, thus constructing new intermediate complete model of the system targeting specific base configuration. Then that data is passed to acts.jar and output loaded back into the framework.

For each base configuration this computation is independent and thus we have perfectly data partitioned input and we should be seeing good scaling with the number of processors available to the framework, which tries to take advantage of all available processors for scheduling of acts.jar runs, regardless of how well is each particular acts.jar invocation capable of using the multiprocessing environment available to it.

We are seeing scaling factor of a bit over 2 between the two test systems, which corresponds with the increase of physical cores of the zen3 processor. That means horizontal scaling with number of processing cores is a viable strategy for getting faster run times, but there also seems some performance to be left untapped, as we would rather expect scaling factor of around 3 (the simultaneous multi-processing implementation on zen3 cpu micro-architecture is generally thought to make two logical cores equivalent to about 1 and a half of physical core). This suggests more tuning could be done to the way how the framework schedules acts.jar jobs.

Also the other points about acts.jar process re-spawning and nature of the Java based implementation discussed above for data in figure 4.6 apply here as well.

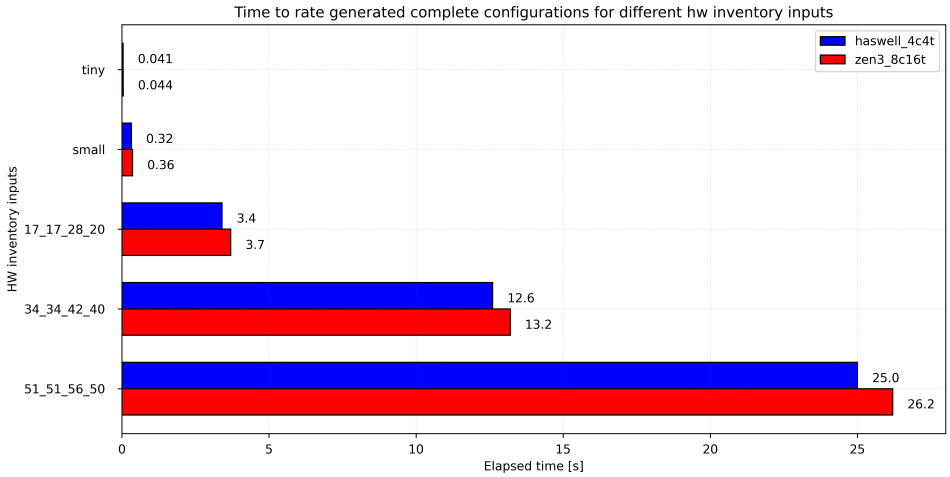


Figure 4.8: Time to rate generated complete configurations for different hw inventory inputs

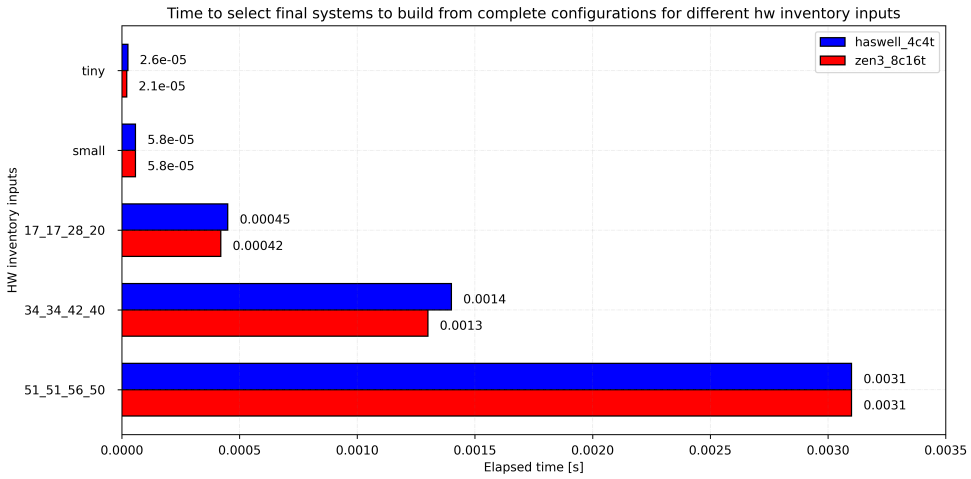


Figure 4.9: Time to select final systems to build from rated complete configurations for different hw inventory inputs

The data depicted in figures 4.8, 4.9 and 4.10 shows a very similar behavior of the rating service component, the final systems selection process and the results processing as is discussed for the data for data ingress in figure 4.5.

All of these parts of the computation are single-threaded and their impact on the total runtime is minuscule for the input sizes used, it could even be argued we do not expect their current implementations to become problem for some significantly larger inputs.

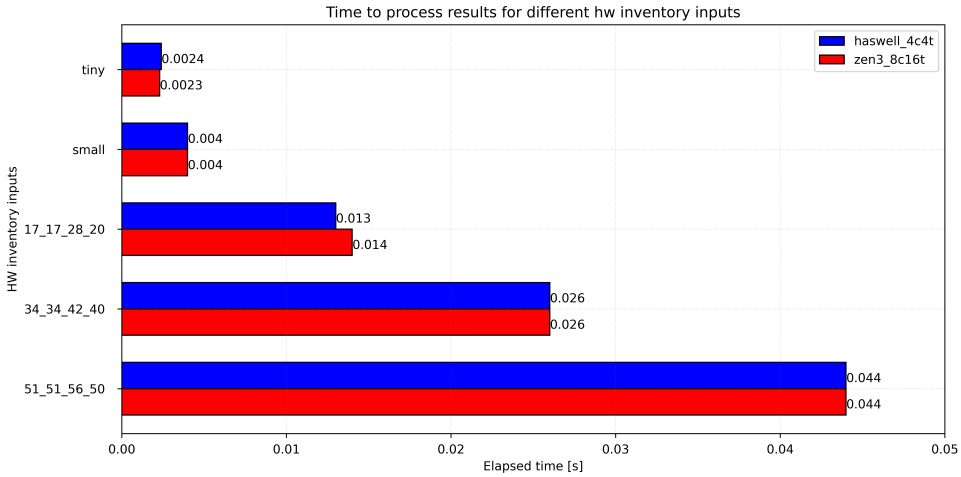


Figure 4.10: Time to process results for different hw inventory inputs

4.5 Possible future expansion and improvements

Integrations

The future of SysCCIT framework certainly lies in integrations with the target environments, without that it is just another single-purpose tool.

The integration with a HW inventory database is already available through the HwParamProvider and a similar integration could be done for getting the application model to the framework, especially for projects that can make use of intelligent software testing mechanisms such as generated application inputs/configuration parameters from source code or model/specification.

On the output side also more integrations to different CI/CD environments and servers such as TeamCity, Github/Gitlab CI and others would be beneficial, as well as integration with automatic test generation tools from provided application model and tests combinations from the output of the framework.

On the part of rating and selection of the final base configurations to be built, a more intelligent rating schema is probably desirable. Current rating procedure cares about the number of covered tuples by the particular combinations. That works quite well but has an inherent bias in preference of systems with less constraints, so these, where more edge case scenarios might be prevalent and we would probably prefer their coverage, may end up underrepresented. In general the rating procedure would benefit from more sophistication by taking into account also th actual relationships between parameters.

The modular architecture enables these additions and so can become a good basis or an integration point for different areas of testing using techniques of combinatorial interaction with constraints.

Performance and different CCIT backends

Having option to choose from different CCIT generator backends would be desirable and possibly necessary for different organizations, for many reasons starting with availability of the tools (ACTS is distributed on request, but that is not guaranteed), through potential licensing issues, supported environments and required features to the plain performance offered by different implementations of different algorithms.

The used ACTS implementation has become a standard tool in the industry, but we do not have much ways to tune its performance and behavior according to the environment where it is used and with respect to the batch-of-jobs nature of the workload in SysCCIT framework. An implementation, that would work more like a service deployable to private or public clouds leveraging horizontal compute resources scaling might be more desirable.

Another way to improve performance would be to consider different and smarter algorithm for generating and selecting best configurations to build, as the current implementation is quite naive. It is usable, but not stellar, and there certainly are areas for more improvement.

Chapter 5

Discussion and conclusion

In this work we've given an overview of the current possibilities in combinatorial interaction testing landscape and how it could be used to solve a problem of selecting good HW component combinations for testing in a resource constrained environments when dealing with software exhibiting complex dependencies and interactions with the underlying HW.

We've designed and implemented SysCCIT framework to make decisions about what configurations of lab computers should we build from given component inventory while taking into account different combinations coverage on different potential HW combinations for a software application under test.

Using the approach of combinations generation with the help of constrained combinatorial interaction testing generator ACTS we were able to avoid an exponential explosion of combinations stemming from the number of parameters defining the base HW configurations as well as the application model.

Although not stellar, the performance of SysCCIT framework implementation brought by this work can be considered usable in real world. Its modular architecture also makes it a good starting point for future development, capabilities and integration extensions as well as performance improvement from better CCIT generators and smarter rating procedures.

We've also shown that the performance of the main workload for the core solving routine does positively scale with more processing cores, so for even bigger instance inputs then we have tested, horizontal scaling of the available computing resources (read adding more processors) is a viable strategy to tackle high run times. As that part of the SysCCIT framework run time has the majority of contribution to total run time, future performance improvement efforts should be concentrated specifically into faster work with combinations and CCIT generators in general.

Bibliography

[1] Bestoun S Ahmed, Taib Sh Abdulsamad, and Moayad Y Potrus. Achievement of minimized combinatorial test suite for configuration-aware software functional testing using the cuckoo search algorithm. *Information and Software Technology*, 66:13–29, 2015.

[2] Bestoun S Ahmed and Kamal Z Zamli. A variable strength interaction test suites generation strategy using particle swarm optimization. *Journal of Systems and Software*, 84(12):2171–2185, 2011.

[3] Yazan A Alsariera, Hussam Alddin S Ahmed, Hammoudeh S Alamri, Mazlina A Majid, and Kamal Z Zamli. A bat-inspired testing strategy for generating constraints pairwise test suite. *Advanced Science Letters*, 24(10):7245–7250, 2018.

[4] Abdul Rahman A Alsewari and Kamal Z Zamli. Design and implementation of a harmony-search-based variable-strength t-way testing strategy with constraints support. *Information and Software Technology*, 54(6):553–568, 2012.

[5] Joshua Bonn, Konrad Fögen, and Horst Lichter. A framework for automated combinatorial test generation, execution, and fault characterization. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 224–233. IEEE, 2019.

[6] Miroslav Bures and Bestoun S Ahmed. On the effectiveness of combinatorial interaction testing: A case study. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 69–76. IEEE, 2017.

[7] Myra B Cohen, Charles J Colbourn, and Alan CH Ling. Augmenting simulated annealing to build interaction test suites. In *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, pages 394–405. IEEE, 2003.

[8] Jaczek Czerwonka. Pairwise independent combinatorial testing tool. [online, accessed 2021-02-23] available at https://github.com/microsoft/pict.

[9] Alan Hartman and Leonid Raskin. Problems and algorithms for covering arrays. *Discrete Mathematics*, 284(1-3):149–156, 2004.

[10] Imad H Hasan, Bestoun S Ahmed, Moayad Y Potrus, and Kamal Z Zamli. Generation and application of constrained interaction test suites using base forbidden tuples with a mixed neighborhood tabu search. *International Journal of Software Engineering and Knowledge Engineering*, 30(03):363–398, 2020.

[11] Richter Jan. Avocado framework plugin for generating combinatorial interaction tests. Master's thesis, České vysoké učení technické v Praze. Vypočetní a informační centrum., 2020.

[12] Hao Jin and Tatsuhiro Tsuchiya. Deriving fault locating test cases from constrained covering arrays. In *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 233–240. IEEE, 2018.

[13] Raghu N Kacker, D Richard Kuhn, Yu Lei, and James F Lawrence. Combinatorial testing for software: An adaptation of design of experiments. *Measurement*, 46(9):3745–3752, 2013.

[14] D Richard Kuhn, Renee Bryce, Feng Duan, Laleh Sh Ghandehari, Yu Lei, and Raghu N Kacker. Chapter one–combinatorial testing: Theory and practice. volume 99 of advances in computers, 2015.

[15] D Richard Kuhn, Raghu N Kacker, and Yu Lei. *Introduction to combinatorial testing*. CRC press, 2013.

[16] R. Kuhn, Y. Lei, and R. Kacke. Practical combinatorial testing: Beyond pairwise. *IT Professional*, 10(3):19–23, May 2008.

[17] Jim Lawrence, Raghu N Kacker, Yu Lei, D Richard Kuhn, and Michael Forbes. A survey of binary covering arrays. *the electronic journal of combinatorics*, pages P84–P84, 2011.

[18] Yu Lei, Raghu Kacker, D Richard Kuhn, Vadim Okun, and James Lawrence. Ipog: A general strategy for t-way software testing. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*, pages 549–556. IEEE, 2007.

[19] Microsoft. Servicing differences between windows 10 and older operating systems - windows deployment. [online, accessed 2021-02-03] available at https://docs.microsoft.com/en-us/windows/deployment/update/waas-servicing-differences.

[20] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2):1–29, 2011.

[21] Duc Truong Pham, Afshin Ghanbarzadeh, Ebubekir Koç, Sameh Otri, Shafqat Rahim, and Muhamad Zaidi. The bees algorithm—a novel tool for complex optimisation problems. In *Intelligent production machines and systems*, pages 454–459. Elsevier, 2006.

[22] R Venkata Rao, Vimal J Savsani, and DP Vakharia. Teaching–learning-based optimization: a novel method for constrained mechanical design optimization problems. *Computer-Aided Design*, 43(3):303–315, 2011.

[23] Jan Richter, Bestoun S Ahmed, Miroslav Bures, and Cleber R Rosa Junior. Avocado: Open-source flexible constrained interaction testing for practical application. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 185–190. IEEE, 2020.

[24] Toshiaki Shiba, Tatsuhiro Tsuchiya, and Tohru Kikuno. Using artificial life techniques to generate test cases for combinatorial testing. In *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, pages 72–77. IEEE, 2004.

[25] Neil JA Sloane. Covering arrays and intersecting codes. *Journal of combinatorial designs*, 1(1):51–63, 1993.

[26] Jose Torres-Jimenez and Eduardo Rodriguez-Tello. New bounds for binary covering arrays using simulated annealing. *Information Sciences*, 185(1):137–152, 2012.

[27] Michael Wagner, Kristoffer Kleine, Dimitris E Simos, Rick Kuhn, and Raghu Kacker. Cagen: A fast combinatorial test generation tool with support for constraints and higher-index arrays. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 191–200. IEEE, 2020.

[28] J. Watada, A. Roy, R. Kadikar, H. Pham, and B. Xu. Emerging trends, techniques and open issues of containerization: A review. *IEEE Access*, 7:152443–152472, 2019.

[29] Linbin Yu, Yu Lei, Raghu N Kacker, and D Richard Kuhn. Acts: A combinatorial test generation tool. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 370–375. IEEE, 2013.

[30] Kamal Z Zamli, Fakhrud Din, Salmi Baharom, and Bestoun S Ahmed. Fuzzy adaptive teaching learning-based optimization strategy for the problem of generating mixed strength t-way test suites. *Engineering Applications of Artificial Intelligence*, 59:35–50, 2017.

Appendix A

List of acronyms

- ACTS — Advanced Combinatorial Testing System
- API — Application Programming Interface
- CA — Covering Array
- CCIT — Constrained Combinatorial Interaction Testing
- CI/CD — Continuous Integration / Continuous Delivery
- CIT — Combinatorial Interaction Testing
- CLI — Command Line Interface
- CPU — Central Processing Unit
- CSV — Comma Separated Values
- DB — Database
- DSL — Domain Specific Language
- GB — Giga Bytes
- GUI — Graphical User Interface
- HW — Hardware
- IPOG — In Parameter Order Generation
- JAR — Java Archive
- JVM — Java Virtual Machine
- MB — Motherboard
- NIC — Network Interface Card
- NIST — The National Institute of Standards an Technology
- ORM — Object Relational Mapping

- OS — Operating System
- QA — Quality Assurance
- RAM — Random Access Memory
- SDLC — Software Development Life Cycle
- SSD — Solid State Drive
- SUT — System Under Test
- SW — Software

Appendix B

SysCCIT framework complete component diagram

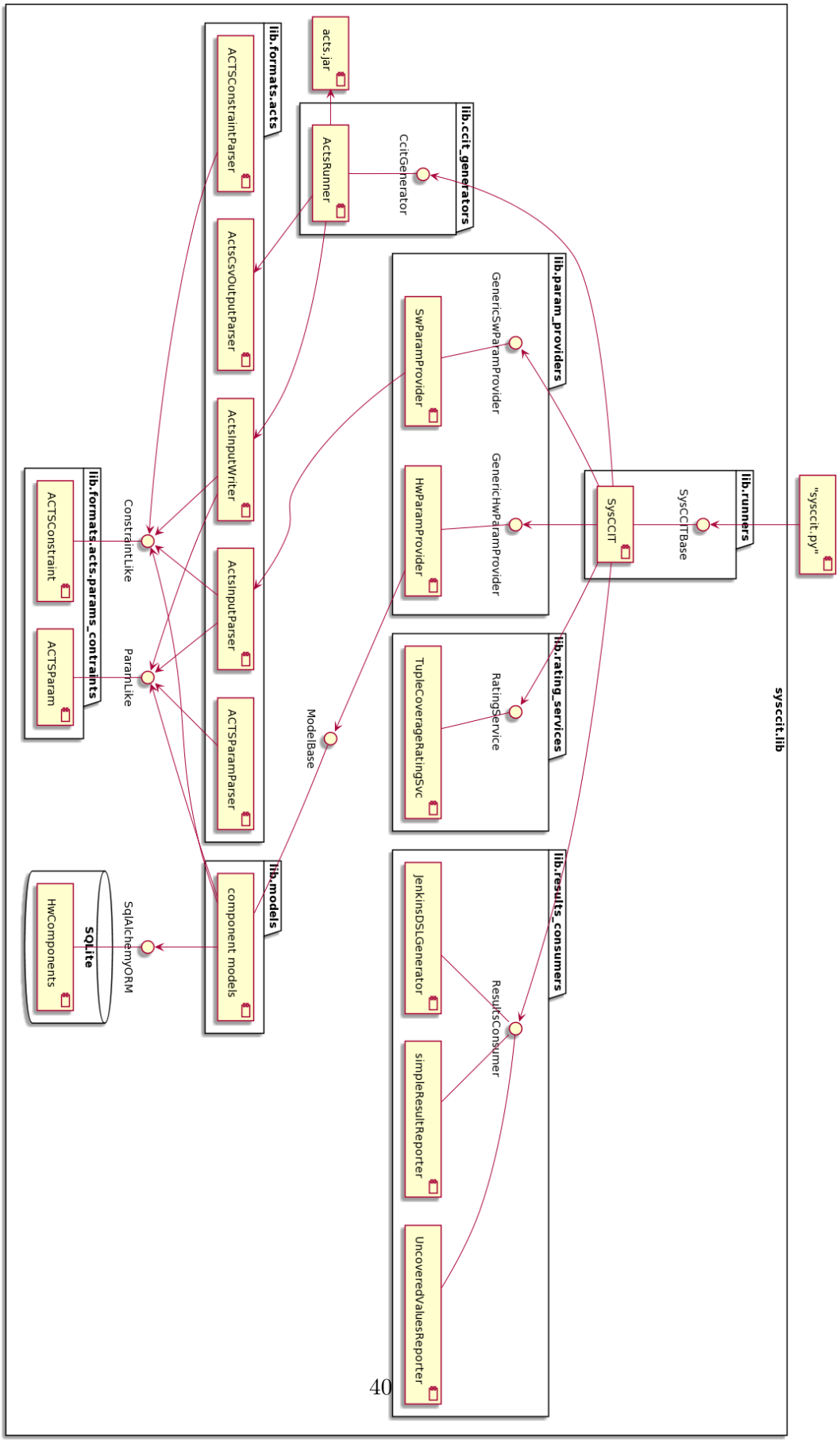


Figure B.1: Complete SysCCIT framework implementation component diagram in bigger more readable edition

Appendix C

SysCCIT framework complete class hierarchy diagram

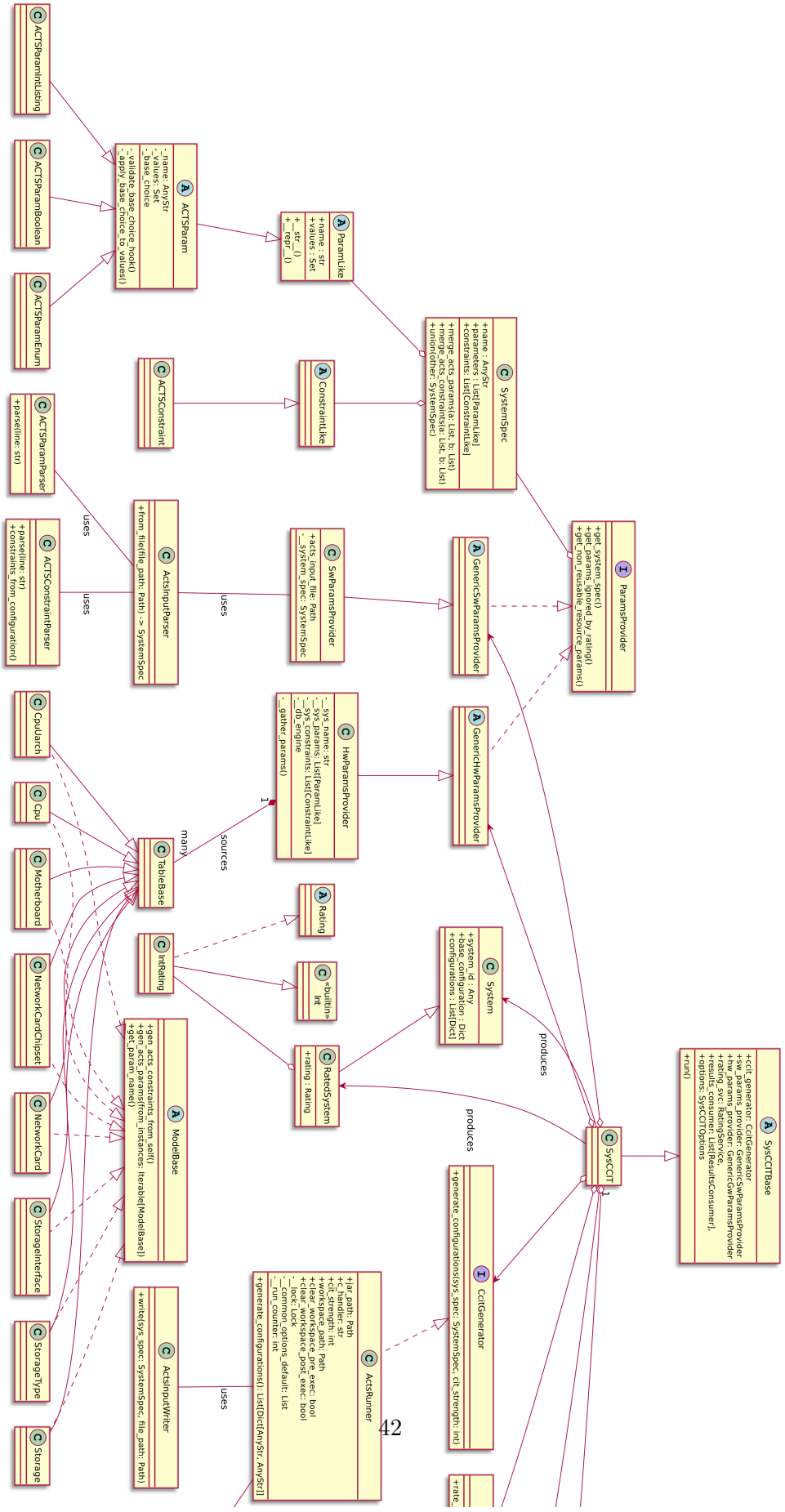


Figure C.1: Complete SysCCIT framework implementation class diagram in bigger more readable edition pt. 1/2

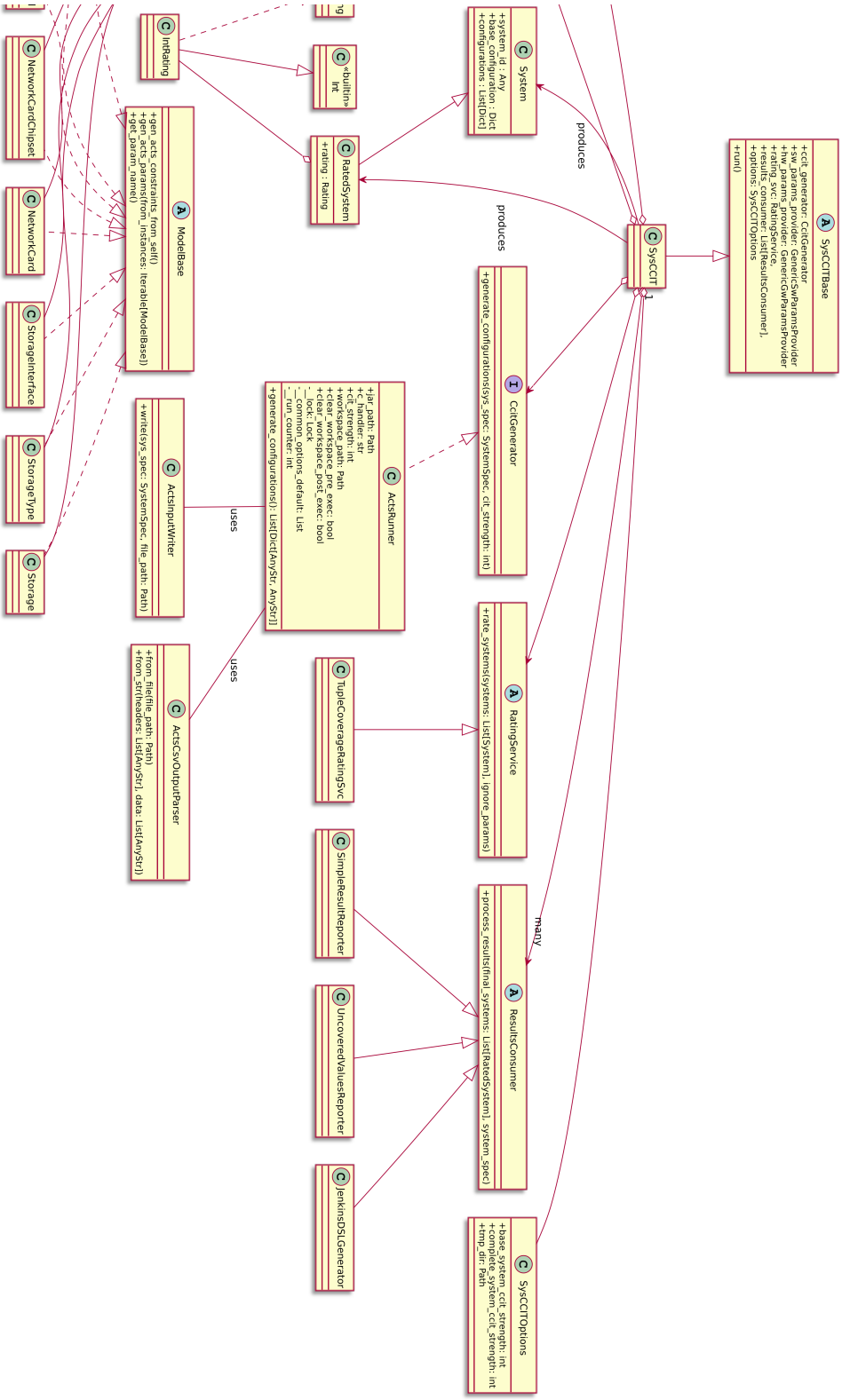


Figure C.2: Complete SysCCIT framework implementation class diagram in bigger more readable edition pt. 2/2