A comparative study of invariants generated by Daikon and user-defined design contracts

by

Mst Farhana Rahman

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Carleton University
Ottawa, Ontario

Abstract

A lot of progress has been made towards the reverse-engineering of program specification under the form of contracts. Ensuring the quality of such reverse-engineered contracts, referred to as likely invariants when one uses Daikon, is paramount since those contracts are used in several other contexts. One aspect that can influence the "quality" of the reverse-engineered contracts is the configuration being used when executing Daikon. In this thesis we evaluate the impact of two such configuration parameters, namely `--nesting-depth` and `--std-visibility`, which help the user control in two different ways how many variables of the program are considered by Daikon when inferring likely invariants. We perform a case study with a program equipped with test cases and high-level design contracts (i.e., design contracts produced before implementation) and systematically compare likely invariants reverse-engineered by Daikon to those contracts, thanks to a comparison framework we devised. We performed our experiment with different settings in Daikon and observed that Daikon is able to infer 57% to 68% of our high-level design contracts. Results confirm and complement previous works, whereby we show that a good proportion of contracts are correctly identified by Daikon as likely invariants, and therefore that many interesting contracts are not discovered by Daikon, but these likely invariants are lost in a mass of incorrect ones.

Acknowledgement

Thanks to Almighty Allah for helping me to achieve my dreams. I would like to thank my supervisor Dr. Yvan Labiche for his guidance, his help and his financial support throughout the last two years. I like to thank the department of System and Computer Engineering for providing me such a wonderful environment to do my research. All the departmental staffs are helpful and cordial, thanks to their support in the last two years. Thanks to the committee members Dr. Babak Esfandiari, Dr. Daniel Amyot and Dr. Roy Gregory Franks for their valuable comments and time.

I take the opportunity to thank the SQUALL members for their help in the last two years. Special thank goes to Google for their wonderful products to help me in doing my research.

I want to thank my parents and my siblings for everything they did in my life.

Finally thank to my little kid little Ruhan for making wonderful noises around when I was writing my thesis.

Table of contents

List of figures

List of tables

Chapter 1 Introduction

Design by Contract (DbC) is a program construction approach that relies on the definition of contracts between caller and callee methods whereby the callee method specifies what it expects as input (i.e., the precondition), and therefore has to be satisfied by the caller method, and what it provides as output (i.e., the post-condition), and therefore is expected by the caller [11]. DbC has been shown to provide many advantages during various software development phases, from analysis [12], to design [2], including low-level design, i.e., program construction [10, 11], as well as verification and validation (e.g., [1]). DbC also asks that invariants be defined. In an object-oriented context, these are called state and class invariants, and they specify a condition that applies to every instance of a class, or loop invariants, and they specify a condition that must be true prior to entering a loop and is then preserved by iterations of the loop [11].

When such contracts have not been (entirely) specified during analysis, design or program construction, they can be reverse-engineered. A well-known approach and tool for the reverse-engineering of contracts is Daikon [6]. In a nutshell, Daikon instruments the code to collect information on variable values during executions of the instrumented program. Daikon then searches for instances of predefined patterns of relations between variables in the recorded execution traces and reports on those it believes (by means of some statistical analysis) hold in the program. Using the Daikon terminology, these expressions are referred to as likely invariants. These likely invariants can correspond to any of the contracts we mentioned above (i.e.,

preconditions, post-conditions, …), depending on where in the program the likely invariants have been found to hold: e.g., a likely invariant that holds prior to the execution of a method is the likely precondition of that method. Daikon detects likely invariants in programs written in several programming languages, including C, C++, Java, Eiffel and Perl.

Detecting (likely) invariants can really help programmers understanding and debugging a program. Likely invariants have also been used in many other aspects of software development, including: theorem proving, repairing data structures, generating test cases, detecting errors and avoiding bugs (see the Daikon web site for a extensive list of such uses of likely invariants [4]).

Regardless of the goal of the reverse-engineering of (likely) invariants, a critical question is whether they are accurate. This is an important question since programmers struggle to classify likely invariants generated by Daikon as being correct or incorrect [16]. The accuracy of likely invariants critically depends upon several factors among which the two most obvious ones are: the set of executions that result in execution traces used by Daikon to discover those invariants; and the configuration of Daikon itself. Indeed, it is intuitive that the more we execute the instrumented program with diverse inputs the more chances we have to collect a large set of accurate invariants. However, the number of executions is not the only characteristic of the set of executions that matters. One can also intuitively recognize that as structural coverage increases, we have more chances to collect a larger number of accurate invariants. This has already been investigated (e.g., [7, 15]): see Chapter 2 for details.

One Daikon parameter of interest to us controls the amount of variables in the program whose values will be monitored and involved in likely invariants. This is interesting since the

larger the number of variables that can be involved in invariants the more likely it is we will obtain a larger set of (accurate) invariants. Another parameter of interest to us restricts the set of variables to consider in invariants to those that are visible to a location where an invariant is to be discovered (e.g., public attributes of class B are visible to class A if the classes belong to the same package). This is equally interesting since restricting the analysis this way may prevent Daikon from discovering invariants that designers may be interested in. (Indeed, when writing contracts in OCL in the context of UML, one does not need to worry about visibility.) In this thesis, we report on an analysis of the impact of these two parameters on the collected likely invariants.

In order to identify whether likely invariants returned by Daikon are accurate, we need a benchmark, that is, software for which we have accurate invariants (or at least invariants that can be deemed accurate enough), and more importantly we need a mechanism to perform the comparison. The comparison cannot be syntactic (i.e., lexicographic) for obvious reasons. It has to account for the semantics of the invariants. In this thesis, we report on a framework for the comparison of such sets of logical expressions: likely invariants against benchmark contracts. Our comparison framework is an extension of a previously published one [14] (see Chapter 2 for details).

Results of our experiments show that Daikon is able to reverse-engineer a large part of contracts originally created during high-level design. However, not all such contracts are discovered by Daikon: a good number of them are not identified by Daikon. Also, our experiments show that identifying those likely invariants that are design contracts is a challenge

since they are lost in a mass of incorrect invariants and that the large majority of these incorrect invariants simply compare apples and oranges. We observe that this search of a needle in a haystack is heavily influenced by the configuration setting of Daikon by means of parameters `--nesting-depth` and `--std-visibility`. Our results identify that the correct likely invariants are lost in a mass of incorrect invariants.

1.1 Summary of our contributions

The contributions of this thesis include:

- A framework to compare likely invariants produced by Daikon to design contracts (preconditions, post-conditions, class invariants) produced in early phases of software development.
- The application of this framework on a case study: comparison was mostly performed by hand; automating as much as possible whatever aspect (i.e., repeated comparisons) could be automated.
- Demonstration of the impact of different values for two different parameters on the quality of produced likely invariants using a case study system.
- Identification of a number of contracts that should have been identified by Daikon since these contracts are instances of variable relations patterns Daikon uses.

1.2 Organization of this thesis document

The rest of the thesis is structured as follows. Chapter 2 discusses related work. Chapter 3 presents the setup of the experiment we conducted, including the benchmark software (with its contracts and test cases), the framework for comparing design contracts produced during high-level design with likely invariants generated by Daikon, and the configuration of Daikon we used. (This chapter also provides an introduction to Daikon.) Chapter 4 reports on the results of our experiment. Chapter 5 discusses the threats to the validity of our study while we draw conclusions in Chapter 6.

Chapter 2 Related Work

The problem of reverse-engineering specifications is very diverse and Daikon is one example solution in this endeavor. We first discuss technologies to detect invariants (section 2.1). We then discuss work related to the validation of Daikon's output by comparisons to other sources of invariants (section 2.2), and work on improving the quality of invariants (section 2.3).

2.1 Daikon and other technologies

Daikon is a popular tool to detect likely invariants dynamically in the research community; the introductory paper [5] of Daikon has over one thousand citations[1]. Apart from Daikon there exist other invariant detection tools. Relevant tools can be categorized into industrial and academic ones.

Sun Microsystems' IODINE [21] is an invariant detector for hardware designs. IODINE executes test vectors or real loads on a simulator that in turn generates the invariants to verify the properties of the design. The Axiom Meister tool [22] performs exhaustive symbolic executions of the actual code and generates specifications from that search. The output is used to generate the test cases for the input system. Agitar Software, Inc. uses the tool Agitator [23], which was inspired by Daikon. The goal of the tool is to detect the dynamic invariants to inform the users about tests and improve the tests.

[1] http://scholar.google.ca/ (Last accessed March 2014)

The most relevant industrial tool could be used in our thesis to generate the likely invariants is Agitator. While the IODINE is used for the verification of hardware design, the Axiom Meister is more of a model checker than an invariant detector. Agitator can be considered as the industrial implementation of Daikon.

Among the tools resulting from academic efforts, the most closely related tool to Daikon is Dynamic Invariant Detection Union Checking Engine DIDUCE [24]. DIDUCE dynamically formulates hypotheses of invariants obeyed by the program while the program is running. While Daikon is a great tool to detect likely invariants dynamically, DIDUCE works great if we want to detect program errors on the fly. By observing the program behavior, DIDUCE initially guesses about the program invariants. It reports any violations of these invariants to the user in order to identify the bugs in the software. As we are comparing the likely invariants to the designers specified DbC instated of identifying bugs in a program we choose Daikon over DIDUCE.

The framework proposed by Ammons, Bodík, and Larus [25] describes a system for inferring temporal specification with the goal to determine the legal sequences of procedure calls. Whaley, Martin, and Lam [26] propose a combination of static and dynamic methods to meet the same goal. The Carrot dynamic invariant detection system is partially based on Daikon's likely invariants [27]. The author pointed out that the tool was still premature at the time of publication, and the status of the tool release date is unknown.

Yang and Evans described a tool [28] that takes execution traces as input and publishes the set of likely temporal properties. This approach is interesting but for our experiment, we needed to generate the likely invariants.

Daikon itself has been used in a number of application contexts (e.g., software testing). These research activities are not in the scope of this paper, and are therefore not discussed here. A good starting point to find more information on these topics is the web site maintained by the authors of Daikon [4].

2.2 Comparing reverse-engineered invariants to benchmarks

In this section we discuss research activities that are related to the objective of our work; that is comparing likely invariants produced by Daikon to some kind of benchmark invariants.

Meyer and colleagues studied the quality of programmer-written contracts by comparing likely invariants produced by Daikon to contracts written by programmers and embedded in Eiffel programs [14]. To perform the comparison, the authors defined four different categories for likely invariants: the likely invariant specifies a relation between unrelated variables; it specifies equality with a constant; it is redundant with another likely invariant at the same program point; it belongs to another program point (it is misplaced). The authors found that Daikon produced many more relevant expressions than what programmers did, that only around 60% of the programmer contracts were correctly identified by Daikon, and that one third of the likely invariants were incorrect or irrelevant. Their notions of invariant correctness and relevance are the following.

They consider a likely invariant to be correct if it is true regardless of the test suite being used, and a likely invariant is relevant if it is correct and interesting, i.e., it makes sense, semantically, in the context of the reverse-engineered software [14]. We do not have any

category similar to these since we are comparing likely invariants to design level contracts rather than studying likely invariants without a basis for comparison.

This study is highly related to ours, though there are important differences. First, we believe there might be a difference between high-level design contracts, possibly coming from OCL expressions, and programmer written contracts embedded in the code, which we would refer to as low-level design contracts. Meyer and colleagues studied the latter kind of contracts whereas we study the former. Another difference is the comparison framework: although there is an overlap (we also have a class for irrelevant likely invariants), we have a refined classification to identify reasons for mismatches between contracts and likely invariants. We do not have a class for misplaced invariants though. As shown later in the thesis, we found that 57% to 68% of contracts, depending on the experiment's configuration settings for Daikon, were correctly identified by Daikon, which is consistent with Meyer and colleagues' findings.

2.3 Improving the quality of invariants

Common sense tells us that there must be a relation between the amount of structural coverage and the quality of reverse-engineered invariants: e.g., if we do not execute a path in a method, no tool can infer post-conditions specific to this path. This common sense has been confirmed experimentally [15].

However, some studies show that standard structural coverage criteria, such as statement coverage or branch coverage, are not good enough for accurate detection of likely invariants [7]. In an attempt to improve accuracy, Gupta and Heidepriem defined a new coverage criterion,

called Invariant-coverage. They use the basic structural coverage (branch coverage and definition-use coverage) suites as test suites for Daikon to generate likely invariants for five different programs. As these programs do not contain any programmer written contracts, they use Daikon generated likely invariants to obtain initial guesses about program invariant properties. They generate definition-use chains for each of these likely invariant properties and then create invariant-coverage suites using inputs that exercise the generated definition-use chain. The authors then showed that using an invariant-coverage adequate test suite for generating likely invariants using Daikon increased the quality of likely invariants: fewer spurious invariants and new accurate invariants. Since the new criterion requires a significant analysis of control and data flow in methods and is not yet supported by an available tool (to the best of our knowledge), we decided to not use it. This is not a threat to the validity of our work and results since our objective is to compare reverse-engineering (likely) invariants to design ones instead of studying accuracy of a set of likely invariants.

Daikon generated likely invariants do not contain logical operators (e.g., AND, OR) or universal quantifiers. They are similar to the predicates of the first order logic. But contracts generated during DbC usually contain those logical operators and quantifiers. Several attempts have been made to improve Daikon generated likely invariants and introduce those operators and quantifiers. Meyer and colleagues devised a completely automated technique, supported by a tool named AutoInfer, which relies on the presence of low-level programmer contracts, and is able to infer contracts with implication or universal quantification [18]. In their paper, they use this technique to focus on inferring postconditions of commands only (i.e., postconditions of methods that change the state of an object). The technique they describe cannot infer postconditions for

queries, i.e., postconditions for methods with no side effect. The tool has been shown to provide 75% of complete specifications (under the form of contracts) for a series of standard data structures. ContExt [19] can automatically infer invariants in a format of "A or B" for Daikon. It is a prototype implementation of the state partitioning technique which adds program-specific disjunctive properties to the language of dynamic constraint detection. This technique is applicable to only one class at a time (state interactions between composed classes are not considered). Dillig and colleagues [20] propose a technique that infers invariants under the form of Boolean combinations of linear integer constraints.

To the best of our knowledge, attempts at providing rich contracts, in fact contracts that would be richer than what Daikon can produce for instance, only consider sub-problems (e.g., universal quantifiers). We do not know of any technology that can produce contracts with all those characteristics together: using universal quantifiers, using the Boolean "or", using the Boolean "and", using linear integer constraints. We therefore decided to not use these technologies and only use Daikon.

Chapter 3 Experimental setup

Our experiment consisted in executing Daikon with different configurations on a case study system for which we had high-level contracts and test cases. Section 3.1 discusses the characteristics of this case study, including contracts and test suite, while section 3.2 discusses the configuration we used when executing Daikon. Section 3.2 also provides some additional information on Daikon. Section 3.3 then specifies how we compared the high-level design contracts to the likely invariants returned by Daikon. Section 3.4 discusses about precision and recall is determined, and section 3.5 specifies how we collect likely invariants (LIs) and measure them.

3.1 Case study system

To perform our experiment, we needed a software system for which we had access to high-level design contracts. We also needed test cases which would exercise functionalities of this software. For these reasons, we selected an ATM system implementation we were familiar with [1] though we did not design or implement it. This is a Java application that simulates typical operations customers perform on an ATM machine. The implementation is made of 20 classes divided into four packages (Table 3.1), with a total of 2,200 lines of code (not counting comments and blank lines).

Among the 20 classes, 14 were further specified with contracts: pre-conditions, post-conditions and class invariants; initially specified along with the class diagram for the ATM system and then as JavaDoc comments using the JContract tool from Parasoft. The contracts were therefore specified prior to the construction of the software, i.e., the creation of the source code. They are therefore different from what a programmer would produce when working on the source code (e.g. programmer might write some loop invariants). These classes are: `ATM`, `Bank`, `CardReader`, `CashDispenser`, `EnvelopeAcceptor`, `Money`, `ReceiptPrinter`, `Session`, `Transaction`, `DepositTransaction`, `TransferTransaction`, `InquiryTransaction`, `OperatorPanel`, and `WithdrawlTransaction`. It is important to note that those contracts had originally been defined in OCL 1.x [17] and then transformed into JContract contracts, and that the notation used in those JContract contracts is very similar to that of JML [8, 9] (though JML has a richer set of features), which is now the standard for specifying contracts in Java. As a consequence, the results of our experiment should extend to a situation where OCL or JML are used.

Table 3.1 ATM system's packages and classes

Package	Class
atm	ATM
	Bank
	Session
	AtmMain
atmparts	CardReader
	CashDispenser
	Display
	OperatorPanel
	ReceiptPrinter
	EnvelopeAcceptor
	GUILayout
	Keyboard
	QuestionDialog
transaction	Transaction
	DepositTransaction
	TransferTransaction
	InquiryTransaction
	WithdrawlTransaction
util	Money
	Status

Table 3.2 provides some structural details about those 14 classes: specifically, the number of attributes and the number of methods (The last two columns of Table 3.2 are explained in section 3.3).

Table 3.2 Structural details about the case study system

Class	No. of attributes	No. of methods	No. of DbC	No. of predicates (JEs)
ATM	12	15	22	22
Bank	16	12	32	32
Transaction	3	4	4	10
Money	1	12	17	18
WithdrawlTransaction	2	4	4	5
TransferTransaction	3	4	6	6
InquiryTransaction	1	4	3	3
DepositTransaction	2	4	4	5
Session	6	6	12	17
OperatorPanel	4	4	4	4
ReceiptPrinter	0	2	11	12
CardReader	3	5	6	10
EnvelopeAcceptor	3	2	2	2
CashDispenser	1	4	5	5

A set of test cases was created for the ATM system [1] using the Category-Partition black-box testing technique [13]. The Category-Partition testing technique is based on the equivalent class partitioning and boundary value analysis testing techniques, and provides additional features. The interested reader is referred to the original paper for more details about the Category-Partition testing technique [13]. More specifically, Category-Partition was used to devise a test suite for each of the eight most important use cases of the ATM system, resulting in eight test suites referred to as TS1 to TS8, a ninth test suite (TS9) was created by combining different use cases (i.e., different transactions are performed by the customer), for an overall total of 51 test cases. Table 3.3 provides a brief identification of each test suite as well as the structural coverage achieved by these test suites in terms of coverage of all uses (data flow) and basic block (control flow) coverage criteria. Data flow and control flow coverage was measured

by Coverlipse[2] and Emma[3], respectively. Note that Table 3.3 does not show the coverage of the 51 test cases on the 20 classes. Instead we report on the coverage for packages of classes: package `atm` includes classes like `ATM` and `Bank`, package `atmparts` includes classes like `CardReader` and `CashDispenser`, package `util` includes classes `Money` and `Status`, package `transaction` includes abstract class `Transaction` and its concrete sub-classes (i.e., `WithdrawalTransaction`, `DepositTransaction`, `TransferTransaction` and `InquiryTransaction`). As expected, since it exercises several functionalities, test suite TS9 achieves the highest overall control-flow and data-flow coverage of the nine test suites. Similarly, AllTS, which is the union of all the other test suites, achieves a higher coverage than any of the individual test suites. We discuss at the end of section 3.2 why we selected TS9 instead of AllTS for our experiments with Daikon.

Table 3.3 Test suites description and coverage (percentage)

		`atm`		`atmparts`		`util`		`transaction`	
		All uses	Basic block	All uses	Basic block	All uses	Basic block	All uses	Basic block
TS1	System start-up	36	56	54	59	27	22	0	32
TS2	Customer session	65	62	88	82	95	89	91	38
TS3	Withdrawal	62	61	77	69	95	64	37	42
TS4	Deposit	60	61	83	83	77	67	50	28
TS5	Transfer	57	62	77	80	73	63	65	28
TS6	Balance inquiry	52	56	70	73	50	39	84	29
TS7	Invalid PIN	67	66	90	83	95	93	86	42
TS8	Failed transaction	49	56	65	63	27	22	54	32
TS9	Combination of transactions	69	67	90	83	95	93	90	57
All-TS	TS1 to TS9 all-together	71	70	95	84	98	95	93	63

[2] http://coverlipse.sourceforge.net/

[3] http://emma.sourceforge.net/

The test suites are designed for specific operations:

1. TS1 verifies the system start up and setting up of the initial amount of currency bills in the ATM.
2. TS2 verifies a typical customer session with the ATM. The customer inserts the card, inputs the PIN, chooses a transaction type and after the successful transaction, the customer chooses another transaction or chooses to terminate the session.
3. TS3 verifies the cash withdraw transaction. The customer chooses an account, and then chooses an amount to withdraw the money from the account. This is a more detailed behaviour than TS2 in the sense that alternative scenarios in use cases are triggered: e.g., a scenario whereby a user has reached his/her daily withdrawal limit or the available balance is not sufficient.
4. TS4 verifies the deposit transaction operation. The customer chooses an account, and enters an amount to deposit the money into the account. This test suite triggers scenarios where a customer chooses an incorrect account type or chooses to deposit an amount and does not insert an envelope.
5. TS5 verifies the transfer transaction operation. The customer chooses an account to transfer the money from, and then chooses the second account to transfer the money into, as well as an amount. This test suite exercises scenarios where a customer chooses an incorrect account type or has reached his/her daily withdrawal limit.
6. TS6 verifies the balance inquiry transaction operation. The customer chooses an account to check the amount/balance on the account. This test suite also exercises a scenario where a customer chooses an incorrect account type.

7. TS7 verifies the invalid PIN extension. If the PIN is incorrect, the customer is asked to re-enter the PIN. If incorrect after three attempts, the card is retained.
8. TS8 verifies the failed transaction extension (use case), i.e., whether the transaction has failed for other reason (e.g. invalid card) apart from an incorrect PIN.
9. TS9 is a combination of all transactions (i.e., `deposit transaction`, `withdraw transaction`, `transfer transaction` and `inquiry transaction`). This suite also exercises other scenarios where there is an insufficient available balance or a user has reached his/her daily withdrawal limit.

3.2 Daikon description and set up

Daikon is an analysis tool that dynamically discovers likely program invariants. Daikon observes the values of variables that the program manipulates and then reports on properties involving those variables that were true at specific program points over an observed set of executions.

Daikon works in three steps as shown in Figure 3.1. This Figure is reproduced from [5].

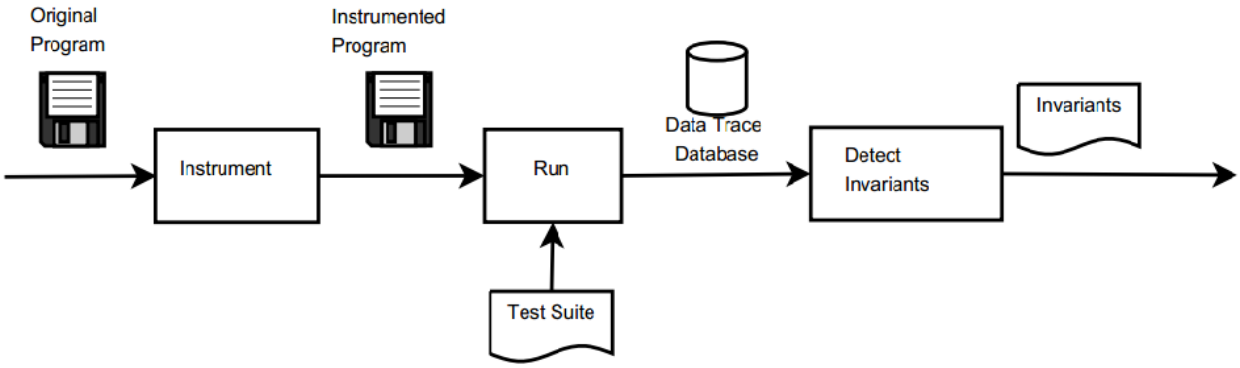

Figure 3.1 Daikon infrastructure [5]

First, a front-end to Daikon is used to instrument the code: We used the Chicory front-end for our ATM system case study. Chicory comes with the Daikon distribution and is used to instrument Java programs. In a second step, the instrumented program is executed and execution traces are created. In a third step, Daikon analyses the execution traces to infer likely invariants. Specifically, Daikon searches for instances of predefined patterns of relations between variables in the recorded execution traces and reports on those it believes (by means of some statistical analysis) hold in the program. Daikon uses program points to organize its output. A program point is a specific place in the source code where a condition has been found to hold over a series of executions. Daikon uses four program points: `:::CLASS`, `:::OBJECT`, `:::ENTER` and `:::EXIT`. The `:::ENTER` program point specifies the entry to a method and the `:::EXIT` specifies a method exit. For example, `session():::ENTER` specifies the entry to method `session()`. The likely invariants at that point are therefore the pre-conditions for the `session()` method. The likely invariants at program point `:::EXIT` are (some of the) the post-conditions of a method. A method may have multiple exit points as it is possible to have multiple

return statements: each exit point likely invariant therefore specifies only part of the overall post-condition, and all exit point likely invariants together specify the overall post-condition. Daikon differentiates these different exits by using the exit line numbers as a suffix to string `:::EXIT`. For example, `dollar():::EXIT153` specifies the likely invariants (some of the post-conditions) that are true at exit point (line number) 153, i.e., post-conditions that are true after the execution of the return statement at line number 153 of method `dollar()`. Program points `:::CLASS` and `:::OBJECT` are used by Daikon to specify class invariants, i.e., expressions that hold true for any instance of a class at entry and exit of every public method of the class. The `:::OBJECT` program point specifies class invariants over all the member fields and static fields of the class. The program point `:::CLASS` is similar to `:::OBJECT`. The only difference is that it specifies invariants over static variables.

Chicory, the Daikon front-end for Java programs has five parameters that impact its output, besides allowing the user to control which program points to be considered (e.g., only preconditions, or only program points for a set of methods). Those are `--omit-var`, `--linked-lists`, `--purity-file`, `--nesting-depth` and `--std-visibility`. The `--omit-var` parameter allows Daikon to omit variables whose name matches a given regular expression from being considered in likely invariants. Using the `--linked-lists` parameter, Chicory outputs user-defined linked lists as sequences; its default value is true. The `--nesting-depth` parameter determines the depth of a structure component (and objects) Daikon will examine from a given program point. `--std-visibility` tells Daikon to traverse those variables that are visible from a given program point. The `--purity-file` parameter allows the user to tell

Chicory which methods in the code are pure methods, that is methods that do not have side effects though perform some computations. Chicory will then consider the result of invoking each pure method at each program point to be studied as a variable to involve in likely invariants.

In our experiment, we always used the default value for the parameter that selects the kinds of program points Daikon considers: Daikon reports on all the different program points. We did not use the `--omit-var` parameter because we did not want to omit any variable for a given expression. We used the default value (true) of the `--linked-lists` parameter. We did not specify pure methods (parameter `--purity-file`) since the source code does not contain many such methods and they are typically attribute accessor methods (`get()` methods) which would not help Chicory and Daikon further since Chicory and Daikon already have access to the attributes. For our experiment we experimented with three different values for the `--nesting-depth` parameter and two different values for the `--std-visibility` parameter.

Table 3.4 Illustrating Daikon parameter `--nesting-depth` and `--std-visibility`

`Class Person {`
`  String name;`
`  int age;`
`  Address address;`
`}`
	`Class Address {`
	`  String city;`
	`  private int roadNo;`
	`}`

`Person P;`	

With `--nesting-depth=0`, Daikon only considers object references. With `--nesting-depth=1`, Daikon also considers the attribute values of those object references. With `--nesting-depth=2`, Daikon, in addition to object references and object references' attributes,

considers the attribute values of the attributes values of those object references. (And so on as the value of `--nesting-depth` increases.) Consider the simple example code of Table 3.4. With `--nesting-depth=0`, only reference `P` will be used in likely invariants. With `--nesting-depth=1`, `P.name`, `P.age`, and `P.address` will also be used. With `--nesting-depth=2`, `P.address.city` will also be used. The higher the nesting depth value, the more variables are monitored and considered when matching relation patterns, and therefore the more chances we have to obtain a larger set of likely invariants.

Using parameter `--std-visibility`, which is either true or false, we can also control the amount of produced likely invariants by Chicory. This option is turned off by default. When turned on, only the variables that are visible from a program point are considered. Referring again to Table 3.4, at any program point in class `Person`, variable `roadNo` in class `Address` is not visible, as this variable is private to `Address`. However, attribute `city` being public, it will be visible to any program point in class `Person`. Thus Chicory will only traverse variable `city` from a program point in class `Person` if `--std-visibility` is turned on. If turned off, Chicory will also consider `roadNo`.

Both parameters `--nesting-depth` and `--std-visibility` can therefore be used to increase or decrease the number of returned likely invariants by controlling the amount of variables Daikon will consider when trying to identify instances of its predefined variable relation patterns. The problem, and research question that we study in this paper is that by increasing the amount of variables Daikon considers using either one or the other of these two parameters, or both of them, one will likely obtain more interesting likely invariants at the likely

expense of spurious likely invariants. To what extent increasing the amount of variables Daikon considers is useful, i.e., to what extent more interesting likely invariants are produced without getting lost among countless spurious ones? Trying to identify interesting likely invariants from Daikon's output might turn out to be similar to looking for a needle in a haystack. In our experiment we have used values 0, 1 and 2 for the `--nesting-depth` parameter, and values true and false for the `--std-visibility` parameter.

Additionally, Daikon includes an automatic theorem prover called Simplify [3] to reduce the amount of identified likely invariants. Even though Daikon attempts to reduce the number of likely invariants through its own internal algorithms, using Simplify gives the whole approach a better capability to reduce the number of redundant likely invariants. For example, if the following two likely invariants are discovered by Daikon for a given program point: `x>=1` and `x>0`; Simplify allows Daikon to report only `x>=1`. In our experiment, we systematically used the Daikon `--suppress_redundant` option, which triggers Simplify.

Last, it is worth mentioning we observed that Daikon does not scale up very well for programs that generate large numbers of traces, which is our case as we will discuss in Chapter 4: with large traces, there are risks of running out of memory. In our experiment, when trying to use all the test suites at once (Table 3.4) with `--nesting-depth=2`, Daikon runs out of heap space. The technique mentioned in Daikon's user guide to deal with this issue is to increase the amount of memory available to the Java Virtual Machine. We tried to increase the memory available to the JVM to 256 megabytes, as suggested in the Daikon user guide. We also tried to increase it to 4 gigabytes. These values did not help resolving the out of heap space problem with

`--nesting-depth=2`. We also tried the same amount of memory available to the JVM with `--nesting-depth=1`, without success. We were using a Linux Ubuntu machine, with a 1.33 GHz, Intel core i5, 64 bit processor and 4 GB memory. Daikon successfully generated likely invariants for all the test suites with `--nesting-depth=0`. We also considered the DynComp feature of Daikon as it is designed to reduce the problem of generating invariants for unrelated values in Java programs. But this option did not help with the memory usage problems. Actually we obtained more likely invariants when using DynComp than without using it. We therefore decided to not use it. We concluded that we would not be able to use all the test cases together without setting `--std-visibility` to true, and indeed, with `--std-visibility` set to true, Daikon successfully generated likely invariants for all the test suites together with `--nesting-depth=0, 1, 2`.

As we were unable to use all the test cases together to study the performance of Daikon with the six planned pairs of values for `--nesting-depth` and `--std-visibility`, i.e., (0,true), (1,true), (2,true), (0,false), (1,false), and (2,false), we selected one of the test suites that would exercise as much as possible the ATM functionalities. We chose TS9 as it is a combination of all transactions (i.e., deposit transaction, withdraw transaction, transfer transaction and inquiry transaction). It is the test suite for which the highest level of coverage is obtained (Table 3.3), and we successfully ran Daikon with the abovementioned six configurations.

3.3 High-level design contract vs. likely invariant

Once the case study system has been run through Daikon (instrumentation and execution on test cases), regardless of the configuration setting of Daikon, we obtain likely invariants which are, using first order logic terminology, predicates (i.e., an expression that evaluates to a Boolean value and does not contain logical operators (e.g., AND, OR). On the other hand, a DbC contract (e.g., a JContract contract) typically contains those operators. In order to compare DbC contracts to likely invariants, we therefore split the DbC contracts into predicates, and compared DbC predicates and likely invariants. Each such DbC predicate is referred to as a JContract Expression (JE) in the rest of this thesis. The numbers of DbC contracts and JEs for classes in the case study are provided in Table 3.3. Then we compared each JE one-to-one to the Daikon's reported likely invariants.

For example a DbC JContract "`A and (B or C)`" is split into three JEs: `A`, `B` and `C`. At a given program point Daikon might report likely invariants (LIs) `A`, `B` and `C` separately. Daikon is not able to report one LI equal to "`A and (B or C)`". It does not even report a LI like "`(A and B) or (A and C)`". If there are several exit points of a method then Daikon might generate LIs `A, B` and `A, C` under two different exit points. For example in the case study a post-condition (JContract) reads: `($result == UNREADABLE_CARD) || ($result== CARD_HAS_BEEN_READ && _cardNumberRead == number)`. In one of our experiments Daikon reports two separate LIs: `atm.atmparts.CardReader.CARD_HAS_BEEN_READ == return` and `this._cardNumberRead == orig(number)`. Admittedly, by splitting high level (JContract) contracts into JEs and comparing JEs to LIs, we lose the information about how

the JEs are related to one another into a contract. This is however necessary because, as mentioned earlier, Daikon is not capable of returning complex expressions (with the Boolean "and" and "or" or with universal quantifiers).

We then needed to devise a precise procedure to perform the comparison between JEs and likely invariants (LIs). We opted for a framework that is based on first order logic, whereby we are looking for implications or equivalences between JEs and LIs. With this objective in mind, to identify an adequate comparison framework, we tried to identify the situations we could encounter. (We did that without considering the situations that would occur when we would actually perform the comparison, i.e., the framework was defined a priori.) The most obvious situation is that one JE is logically equivalent to exactly one LI. When this does not happen, we consider the following situations: one JE is logically equivalent to several LIs; one LI is logically equivalent to several JEs. When equivalency cannot be found, logical implications may happen instead: one JE implies one LI; one LI implies one JE; one JE implies several LIs; one LI implies several JEs. Last, in the worst-case scenario, we would have one JE that does not match any LI, or one LI that does not match any JE. This is illustrated in Figure 3.2.

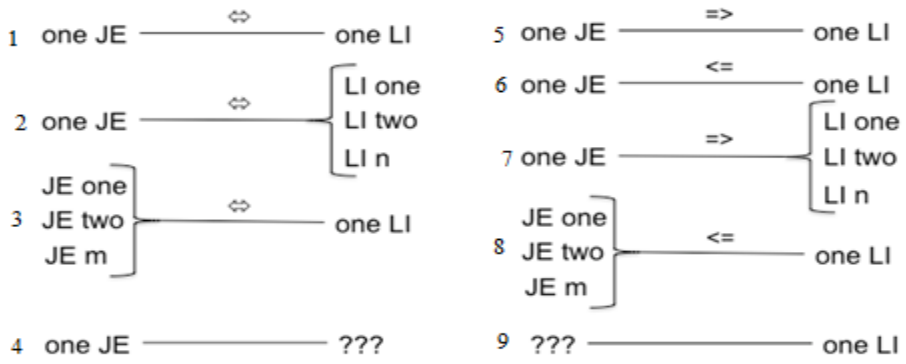

Figure 3.2 Illustrating the comparison framework

Relying on first order logic ensures that our comparison will be complete and unbiased. In the latter cases, i.e., when there is no matching between a JE and an LI, we tried to identify categories of mismatch. This led us to identify the following ten categories for comparing JEs and LIs, and label each LI accordingly.

Equiv: A JE is equivalent to one (and only one) LI. If such a case is identified, we label the LI as *Equiv*. As an example, in one of our experiments, Daikon returned the LI `this._number == orig(number)` for the end of execution of a method, and we found `_number == number` in a JE post-condition. These two expressions are logically equivalent.

A JE may be equivalent to two or more LIs. If such a case is identified, we label the LIs as *Equiv* too. And we record which JE they are all equivalent to, which allows us to distinguish this situation from the one-to-one equivalence above. As an example, in one of our experiments, Daikon returned the following LIs at the end of a method execution: `this.currentCash == return` and `this.currentCash.cents == return._cents`. Given that the returned

value is an instance of class `Money` and `_cents` is one of its attributes, each one of these two likely invariants is equivalent to the following JE: `$result.equals(_currentCash)`. If a LI is equivalent to two or more JEs then we label the LIs as *Equiv* too. However we did not find any instance where several JE are equivalent to one LI.

Likely Equiv: A JE is equivalent to the conjunction of several LIs. In such a case, we label each LI as *Likely Equiv*. In other words, the JE implies each LI separately, but each LI does not imply the JE; only their conjunction does. We did not find any instance of Likely Equiv although we were very close to finding some. Specifically, in the case study a post-condition reads: `$result == Password.equals("123456")`. In one of our experiments, Daikon returned `(password.toString == "123456") <==> (return == true)` and `(password.toString == "2") <==> (return == false)`. Clearly the conjunction of the two LI's is not equivalent to the post-condition (JE). The following two LIs would be labeled as Likely Equiv: `(password.toString == ¬ "123456") <==> (return == false)` and `(password.toString == "123456") <==> (return == true)`. But Daikon failed to report `(password.toString == ¬ "123456") <==> (return == false)`. The characteristics of the test case inputs, i.e., the fact that only two password strings are used, likely explains the LIs returned by Daikon.

Partially Imply: If Daikon fails to return an expression which, in conjunction with what it already returns, would imply the JE, we label the LI as *Partially Imply*. For instance, for JE `$result == insert.equals("insert")`, Daikon returns only `insert.toString ==`

`"insert" and return == true`, but does not return `insert.toString == ¬ "insert" and return == false`. We label these two LIs as *Partially Imply***.**

JE_Imply: One or more JEs imply one LI. If such a case is identified, we label the LI as *JE_Imply*. We did not find any instance of JE_Imply. We nevertheless discuss this case to have a list of situations as complete as possible.

Semantically Incorrect: This is a mismatch category whereby an LI does not match any JE simply because Daikon tries to compare apples and oranges: we label the LI as *Semantically Incorrect*. As an example, in one of our experiments, Daikon returned expression `atm.Bank.CHECKING <= atm.Bank._PIN[this._state]`, comparing a type of account (`atm.Bank.CHECKING`) and a PIN number (`atm.Bank._PIN[this._state]`), which does not make sense. It simply happens that the two values have the same type and, in a large enough number of executions, the value of enumeration `atm.Bank.CHECKING` is always lower or equal to specific PIN numbers. Therefore Daikon returns the inequality.

GUI Element: This is a mismatch category whereby Daikon returns an expression about elements of the graphical user interface. Since the designer did not specify any contract for the GUI, we did not find JEs similar to those LIs. We label such LIs as *GUI Element*. For instance, Daikon returns expression `atm._envelopeAcceptor._display == null` at the end of the execution of a method.

Obvious: This is a mismatch category whereby Daikon returns an expression that we deem so obvious that a designer would unlikely specify it. We label such a case as *Obvious*. For

instance, in one of our experiments, Daikon returned the expression `atm.Bank.accountNames.getClass() == java.lang.String[].class`, which means that the names of the accounts at the bank is an array of strings.

Frame: This is a mismatch category whereby Daikon returns a frame expression. A frame expression specifies in a post-condition that something (for instance an attribute value) is not changed by the method. Since the designer who generated the JEs did not write any frame expression, we consider this a mismatch. We label such a LI as *Frame*. As an example, in one of our experiments, Daikon returned expression `this.password == orig(this.password)` at the end of execution of a method.

Dummy: This is a mismatch category whereby Daikon returns an expression about a variable that the designer has hard coded in the program. We label such a LI as *Dummy*. For instance, in one of our experiments, Daikon returns expression `atm.Bank._PIN[] == [0, 42, 1234]` since only those PIN values were used in our test suite.

Not useful: This is a mismatch category whereby Daikon returns an expression that is true in the set of executions we used but other executions can be found during which the expression would be false. We label such LI expression as *Not useful*. As an example, in one of our experiments, Daikon returned the expression `amount._cents == 4000`. We consider `amount._cents == 4000` as incorrect since `amount._cents` specifies the amount of money (in cents) a customer wants to withdraw from the ATM machine. Therefore `amount._cents`

`== 4000` specifies that a customer always wants to withdraw 4000 cents, which is in general not true.

The mapping between different situations that we considered (Figure 3.2) in our framework is as follows: case 1, 2 and 3 from Figure 3.3 fall in category Equiv, case 6 and 8 fall in category JE Imply, case 5 and 7 fall in category Partially Imply and case 7 also falls in category Likely Equiv. Case 9 is the combination of mismatched categories (i.e. Semantically Incorrect, GUI Element, Dummy, Obvious, Frame, Not useful**).** We do not have any category for case 4 as we are analyzing the LIs.

3.4 Precision and Recall

In our context, precision and recall can help us evaluate how well Daikon performs, that is how well Daikon is able to reverse-engineer high-level design contracts. Precision and recall can be explained with the notion of confusion matrix, which helps us visualize and reason about the performance of Daikon. Each column of the matrix represents the instances (i.e., invariants) in a predicted class, that is, in our case, what Daikon tells us, while each row represents the instances (i.e., invariants) in an actual class, that is, in our case, the high-level design contract expressions. The confusion matrix for our situation is in Table 3.5: capital letters `A`, `B`, `C`, and `D` refer to elements of the Venn diagram of Figure 3.3 which is discussed below.

In the confusion matrix, we indicate that Daikon may return a LI (column "correct") that we know (from the list of JEs) is either one we expect (row "correct") or not (row "incorrect"). These are called the True Positive instances (TP) and False Positive instance (FP), respectively.

If Daikon does not return an expression we expect (row "correct"), although Daikon says nothing (it does not return anything), we consider that Daikon identified that an expression was incorrect (column "incorrect"). These are False Negative instance (FN). The remaining instances are called True Negative (TN).

Table 3.5 Confusion matrix for evaluating the performance of Daikon

	Daikon prediction	
	Correct C	Incorrect A \ C
Correct contracts B	True Positive (TP) D	False Negative (FN) B \ C
Incorrect contracts A \ B	False Positive (FP) C \ B	True Negative (TN) A \ (B ∪ C)

Let us now illustrate those notions with the Venn diagram of Figure 3.3: A is the set of all possible contracts that can be imagined[4]; B is the set of JEs specified by the designer; C is the set of LIs reported by Daikon; and D is the set of LIs for which we have matching JEs (the intersection between C and B). True positive (TP) instances are expressions that are returned by Daikon and that match expressions we expect (JEs). As recorded in Table 3.5, referring to Figure 3.3, TP is set D. False positive (FP) instances are incorrect expressions that are returned by Daikon. As recorded in Table 3.5, referring to Figure 3.3, FP is set C\B. False negative (FN) instances are correct expressions that are missed by Daikon. As recorded in Table 3.5, referring to Figure 3.3, FN is set B\C.

[4] We cannot build set A in practice but this is not an issue for our discussion and the definition of precision and recall from the confusion matrix, as mentioned later in this section.

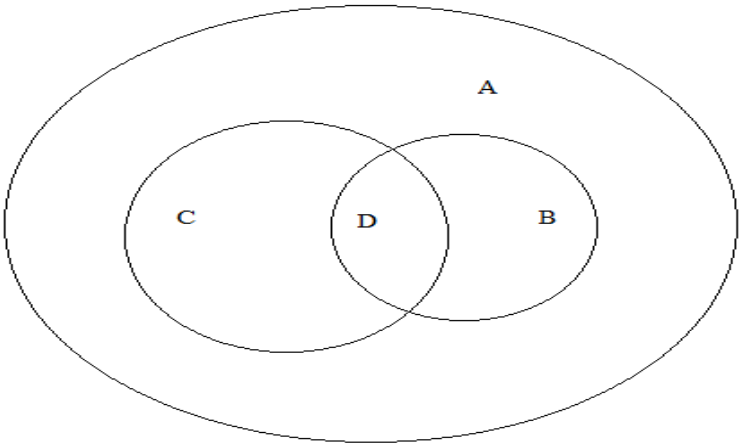

Figure 3.3 Venn diagram for Precision and Recall

We calculate the precision and recall for the likely invariants reported by Daikon using the confusion matrix. Precision is defined as the ratio of the number of correct identifications of high-level design contracts reported by Daikon to the total number expressions reported by Daikon. And recall is defined as the ratio of the number of correct identifications of high-level design contracts reported by Daikon to the total number of JEs specified by designers. Using the confusion matrix, we obtain:

$$Precision = \frac{|TP|}{|TP| + |FP|}$$

$$Recall = \frac{|TP|}{|TP| + |FN|}$$

Precision and recall are usually expressed as percentages. A high precision, the highest value being 100%, means that Daikon returned substantially more correct results than incorrect, while high recall, the highest value being 100%, means that Daikon returned most of the correct

results. Recall is also often called the true positive rate (TPR). Another common metric used is the false positive rate (FPR), which is computed as follows:

$$FPR = \frac{|FP|}{|FP| + |TN|}$$

The TPR and FPR metrics are commonly used to plot the Receiver Operating Characteristics (ROC) graph, which is considered useful for visualizing the performance of a classifier [29]. A ROC graph is a two-dimensional graph that plots TPR as a function (y-axis) of FPR (x-axis). For our experiment TN is uncountable which results in FPR to equal to 0 in the limit, regardless of FP. Hence we decided not to use ROC curves. Instead we choose to study the performance by plotting Precision as a function of Recall. Also Goadrich and colleges show that there exist strong connections between the Precision-Recall curve (PR curve) and the ROC curve [30]. They show that a curve dominates in ROC space if and only if it dominates in PR space.

Determining sets FP, FN, and TP, and computing Precision and Recall assumes that what we evaluate, i.e., what Daikon predicts, and what we know, i.e., high-level contracts, are part of a same larger set (`A` in Figure 3.3) of Boolean expressions, i.e., we can evaluate with certainty whether what Daikon predicts is correct given what we know. In other words, we need to evaluate whether a LI (Daikon) equals a JE (high-level contracts). We therefore need to qualify what it means for an LI to equal a JE. This is not necessarily an easy task if we want to be fair to Daikon, if we want to evaluate as much as what Daikon returns. And this issue is very much related to the comparison framework we discussed in section 3.3.

Indeed, the notion of equality between an LI and a JE could first be thought as string equality: an LI equals a JE if and only if the string representing the LI is lexicographically equal to the string representing the JE. This is obviously not adequate, and should not be used, since a given constraint can be written in lexicographically different ways. Instead, we can use a logical equivalence: i.e., an LI predicted by Daikon is correct, i.e., in set `D` (Figure 3.3) if and only if it is logically equivalent to one JE. This may however not be considered entirely fair to Daikon since Daikon may return LIs that are almost equivalent to what we expect (the JEs). This was the whole purpose of the procedure we devised to split high-level contracts into predicates and the comparison framework we devised and discussed in section 3.3. To be fairer to Daikon we should at least account for the LIs that are implied by some JEs.

We therefore propose to determine sets FP, FN, and TP, and compute Precision and Recall in two different stages, relying again on our comparison framework, each stage relying on one precise notion of equality between what Daikon returns (LIs) and what we know (JEs).

The first stage is to only account for LIs that are found logically equivalent to some of the JEs: an LI belongs to set `D` (Figure 3.3) if and only if there exists a JE that is logically equivalent to that LI; every other LI, even those that are found to be implied by some JE are considered incorrect and belong to `C\B`. These LIs were labeled *Equiv* in our comparison framework (section 3.3). We also consider the cases where several LIs are equivalent to one JE; in such a case we count the JE once and each equivalent LI once. An example where several LIs are equivalent to one JE is as follow: in one of our experiments Daikon returns `this._currentCash._cents == initialCash._cents` and `this._currentCash ==`

`initialCash`. Given that the compared values are instances of the class `Money` and `_cents` is one of its attributes, each one of these two likely invariants is equivalent to the following JE: `_currentCash.equals(initialCash)`, since equality between `Money` instances is the equality of their `._cents` attribute values. In this example we count the JE once and the two equivalent LIs individually once (i.e., the number of LIs is 2). We determine the set `D` and then we determine the number of elements of the sets FP_{equiv}, FN_{equiv}, and TP_{equiv}, and compute $Precision_{equiv}$ and $Recall_{equiv}$ as follows:

$$Precision_{equiv} = \frac{|TP_{equiv}|}{|TP_{equiv}| + |FP_{equiv}|}$$

$$Recall_{equiv} = \frac{|TP_{equiv}|}{|TP_{equiv}| + |FN_{equiv}|}$$

In a second stage, we not only account for LIs that are logically equivalent to some of the JEs but we also account for those LIs that are implied by some of the JEs and for which Daikon fails to return an expression which, in conjunction with these LIs, would imply the JE. These LIs were labeled *Equiv* and *Partially Imply*, respectively, in our comparison framework (section 3.3). In other words, an LI belongs to set `D` (Figure 3.3) if and only if either (i) it is labeled *Equiv* or (ii) it is labeled *Partially Imply*. (Remember that we did not find any Likely Equiv instances; As a result we do not develop a specific solution for them; However, they can easily be accounted for as further discussed later in this section.)

Determining the contents of set `D` is not as simple since, if we do not pay attention, we can obtain Precision and Recall values that would make no sense. Suppose for instance that we have

100 JEs (set B) and 1000 LIs (set C); Suppose that for 50 LIs we can find a logically equivalent JE (these are labeled *Equiv*); Suppose that for 25 JEs, Daikon was not able to produce a logically equivalent LI but instead produces, for each of those 25 JEs, three LIs that are labeled *Partially Imply*; Further assume that for each one of these 25 JEs, Daikon fails to return one LI that together (conjunction) with the three LIs Daikon returns would be equivalent to the JE. Using the definition of set D in the previous paragraph, D would contain all the *Equiv* and all the *Partially Equiv* LIs, that is 50 + 25*3 = 125 elements. Then, $Recall = {|D|}/{|B|} = {125}/{100} = 1.25$, which does not make sense.

To devise an accurate identification of set D and a meaningful computation of Precision and Recall, while accounting for LIs that are not equivalent to any JE, we proceed as follows. When a JE is found to not be equivalent to one LI but instead a number of (*Likely Equiv*) LIs, we consider that the JE can be decomposed into building blocks (expressions) that together (conjunction) would be equivalent to the JE. In such a situation, Daikon may be able to return all these expressions, in which case these (LI) expressions are labeled *Likely Equiv* in our framework, or Daikon may only be able to return a subset of those expressions, in which case the returned (LI) expressions are labeled *Partially Imply* in our framework. In the general case, when a JE is equivalent to the conjunction of g expressions, Daikon returns f expressions (LIs), f≤g. For instance, a typical situation we have encountered is that Daikon returns two LIs (f=2) and fails to return one LI such that all these LIs together (conjunction) would be equivalent to one JE (g=3). It is those three building block expressions, and their possible combinations (conjunctions) that we will use to identify the number of elements in set D. In the case Daikon

returns two out of three LIs, assuming that the conjunction of those three LIs is equivalent to one JE, we consider that Daikon can potentially return any one of the three LIs (three such possibilities), any pair of those LIs (three such possibilities), the three LIs together (there is one such triplet), or could have returned nothing. In other words, Daikon could have returned elements of a set of 3+3+1+1 = 8 elements (the empty element being part of this set), i.e., 2^3 elements. This is the number of distinct subsets that can be constructed from g elements (here g=3), which is 2^g. In more general terms, when one JE is equivalent to the conjunction of g LIs and Daikon returned f LIs ($f \leq g$), each such LI is labeled *Partially Imply* according to our comparison framework; we therefore consider that the JE in fact represents 2^g elements and that Daikon returned f of those elements.

Let us reuse the example above, where we have 100 JEs (set `B`) and 1000 LIs (set `C`), where for 50 LIs we can find a logically equivalent JE (these are labeled *Equiv*), where for 25 JEs, Daikon was not able to produce a logically equivalent LI but instead produces, for each JE, three LIs that together (i.e., conjunction) with one LI Daikon fails to return would be equivalent to one JE (i.e., g=4 and f=3), and where 25 JEs do not find any counterpart LI. In this example, we consider that set `B` contains $50 + 25 + 25*2^4 = 475$ elements (set `B`): term 50 represents the JEs for which we find an equivalent LI; term 25 represents JEs that do not match any LI; term $25*2^4$ represents the building blocks that make up the JEs for which we find mapping *Partially Imply* LIs. In such a case, Daikon was able to correctly identify $50 + 25*3 = 125$ of those elements (set `D`). This results in a Recall of $|D|/|B| = 125/475 = 0.26$.

In more general terms, let us assume the following: n JEs and m LIs, e of the JEs are equivalent (*Equiv*) to LIs, p of the JEs are *Partially Imply* to pairs of LIs and Daikon returns pd LIs of the pairs for each JE (pd≤2), t of the JEs are *Partially Imply* to triplets of LIs and Daikon returns td LIs of the triplets for each JE (td≤3), q of the JEs are *Partially Imply* to quadruples of LIs and Daikon returns qd LIs of the quadruples (qd≤4). We compute the number of elements of sets $FP_{\text{partiallyImply}}$, $FN_{\text{partiallyImply}}$, and $TP_{\text{partiallyImply}}$, as follows. (Note that formulas can be extended for any higher arity than four for tuples of *Partially Imply* LIs. Also note that formulas account for LIs that are in fact *Likely Equiv*, instead of *Partially Imply*, since in the former cases we have either pd=2 or td=3 or qd=4 … and in the latter cases we have either pd<2 or td<3 or qd<4 …):

$$\left|TP_{\text{partiallyImply}}\right| = e + p * pd + t * td + q * qd$$

$$\left|FN_{\text{partiallyImply}}\right|$$

$$= \{e + [n - (e + p + t + q)] + p.2^2 + t.2^3 + q.2^4\}$$

$$- \{e + p * pd + t * td + q * qd\}$$

$$= n + p.(2^2 - pd - 1) + t.(2^3 - td - 1) + q.(2^4 - qd - 1) - e$$

$$\left|FP_{\text{partiallyImply}}\right| = m - (e + p * pd + t * td + q * qd)$$

The formulae defining $FN_{\text{partiallyImply}}$, is made of two terms, each identified within curly brackets: The first term specifies the number of elements in set `B` whereas the second term

specifies the number of elements in set D. The first term is itself made of three parts which correspond to three subsets of set B, which are: (1) the number of JEs for which we can find an *Equiv* LI, i.e., e; (2) the number of JEs that do not match any LI, i.e., $n - (e + p + t + q)$; (3) the number of expressions Daikon can potentially return, according to the discussion we had previously, i.e., $p.2^2 + t.2^3 + q.2^4$. The second main term in the overall formulae is the number of elements in set D.

Then, we compute $\text{Precision}_{\text{partiallyImply}}$ and $\text{Recall}_{\text{partiallyImply}}$ as follows.

$$Precision_{\text{partiallyImply}} = \frac{\left|TP_{\text{partiallyImply}}\right|}{\left|TP_{\text{partiallyImply}}\right| + \left|FP_{\text{partiallyImply}}\right|}$$

$$Recall_{\text{partiallyImply}} = \frac{\left|TP_{\text{partiallyImply}}\right|}{\left|TP_{\text{partiallyImply}}\right| + \left|FN_{\text{partiallyImply}}\right|}$$

Referring to the running example above, where n=100, m=1000, e=50, p=t=0, pd=td=0, q=25, qd=3, we obtain the following:

$$\left|TP_{\text{partiallyImply}}\right| = 50 + 0 + 0 + 25 * 3 = 125$$

$$\left|FN_{\text{partiallyImply}}\right| = 100 + 0 + 0 + 25.(2^4 - 3 - 1) - 50 = 100 + 25.12 - 50 = 350$$

$$\left|FP_{\text{partiallyImply}}\right| = 1000 - (50 + 0 + 0 + 25 * 3) = 875$$

and:

$$Precision_{\text{partiallyImply}} = \frac{125}{125 + 875} = 0.125$$
$$Recall_{\text{partiallyImply}} = \frac{125}{125 + 350} = 0.26$$

3.5 Collecting and measuring LIs

The comparison between Daikon generated likely invariants and design contracts are mostly performed by hand. We tried to automate as much as possible by using Microsoft Excel. Specifically, we used the VLOOKUP function to maintain consistency: this function searches for a value in a column of a table and returns a value in the same row from another column in the table. By using this function we avoided repeated evaluations of the same (JE, LI) comparisons and then ensured consistency of the evaluation. More specifically, we started with nesting depth 2 for test suite TS9 with the `--std-visibility` parameter set to false as this configuration returned the largest number of LIs, and we performed all the evaluations manually. We refer to this set of evaluations as Eval_0. Then, before making any new comparison between a JE and an LI, we looked (using VLOOKUP) for the result of this evaluation in Eval_0. If we did not find it, then we performed the evaluation manually. If the evaluation had already been performed, then we reused the previous result automatically, thanks to the use of VLOOKUP. This way we obtained new evaluations, which we refer to as Eval_1. Further evaluations proceeded similarly, by looking for the results in Eval_0 and Eval_1, using the result if it were found or manually performing the evaluation otherwise. We proceeded this way when decreasing the nesting depth, and since the LIs at nesting depth levels 0 and 1 are also found at nesting depth 2, we saved a lot of effort while maintaining consistency.

Table 3.6 Sample likely invariants of `Money` class

	A	B	C	D	E	F	G
1			**Contract**	**TS9 noSTD ND 2**	**Manual**	**TS9 noSTD ND 1**	**Automatic**
2	**Class Invariant**	**Inv**	**_cents >= 0**	this._cents >= 0	Equiv	this._cents >= 0	Equiv
3							
4	**Money()**	**Post**	**_cents==0**	**atm.util.Money.Money():::EXIT**		**atm.util.Money.Money():::EXIT**	
5				this._cents == 0	Equiv	this._cents == 0	Equiv
6							
7	**Money (int dollars)**	**Pre**	**dollars >= 0**	**atm.util.Money.Money(int):::ENTER**		**atm.util.Money.Money(int):::ENTER**	
8				dollars >= 0	Equiv	dollars >= 0	Equiv

Table 3.6 shows a portion of the LIs that Daikon reported. This table is actually a small excerpt of the real data we collected in Excel. In column A, B and C of Table 3.6 we store information about Method names, the types of contract ("inv" for invariant, "pre" for precondition, and "post" for post-condition), and the corresponding DbC contracts expressions (i.e., the JEs). Column D shows Daikon generated LIs with TS9 for depth 2 and `--std-visibility` set to false. In the next column (E) we record the result of the manual comparison between the contents of column C and column D for the corresponding row, identifying the category of the LI. In column F, we store LIs returned with test suite TS9 for depth 1 and `--std-visibility` set to false. In the next column (G) the category of generated likely invariants is identified automatically using VLOOKUP and the contents of columns C, D, and E: typically, VLOOKUP searches in column D for the contents of a cell in column F for the same JE (column C). For instance, it searches for the contents of F2 in column D, row 2. If the search succeeds,

VLOOKUP returns the contents of column E at the same row. For instance, for F2, it returns `Equiv`, and stores this in G2. The syntax for the VLOOKUP function is VLOOKUP (`Value, table_array, index_no, [not_exact_match])`, where `Value` is the value to search for in the first column of the `table_array`, `table_array` is the column(s) of data and `index_no` is the column number in `table_array` from where the matching value must be returned. The first column of `table_array` is at index 1. The `optional [not_exact_match]` parameter determines whether we are looking for an exact match based on the `Value` parameter or not. The logical `[not_exact_match]` parameter has two possible values: `FALSE` in which case VLOOKUP finds an exact match and `TRUE` in which case VLOOKUP finds an approximate match. As an example, the formula of cell G2 is (F2, $D$2:$E$2, 2, `FALSE`).

Daikon traces for every class are stored in a separate sheet of the Excel file. We count the number of likely invariants in each category considering every program point separately. Finally, we count the total number of likely invariants (summation of class invariants, preconditions and postconditions) of a class.

Chapter 4 Result Analysis

In this Chapter, we will start the discussion by reporting on the number of LIs generated by Daikon for different configurations (section 4.1). We then turn our attention to the application of our comparison framework (section 4.2). Finally we will identify reasons for missing JEs (section 4.3) and analyze results in terms of precision and recall (section 4.4)

4.1 Initial analysis

Table 4.1 shows, for each of the 14 classes for which we have JEs (recall these are JContract expressions), the number of JEs available in that class (sum of the preconditions, post-conditions and invariants), and the numbers of LIs (likely invariants) obtained with different configurations for Daikon (recall the end of section 3.2): test suite TS9 for different values of `--nesting-depth` (`--nesting-depth = 2, 1, 0` are labeled D2, D1, and D0, respectively) while setting `--std-visibility` to false (noSTD); test suite TS9 for different values of `--nesting-depth` while setting `--std-visibility` to true (STD); all test suites together (AllTS) for different values of `--nesting-depth` while setting `--std-visibility` to true; all test suites together for `--nesting-depth=0` while setting `--std-visibility` to false.

Recall that to compare DbC contracts to likely invariants we have split the DbC contracts into predicates, i.e., JE's that does not contain logical operators (e.g., AND, OR). Hence the total number of JE's is larger than the total number of DbC (Table 3.2). For example, the precondition

of method `Money(int dollars, long cents)` of class `Money` counts for one DbC but two JEs: `dollars >= 0 && cents >= 0`.

What is striking in Table 4.1 is the huge amount of likely invariants returned by Daikon, even at depth 0 when the `--std-visibility` parameter is set to false. Another lesson to learn from the table is the jump of total number of LIs from depth 0 to depth 1 and from depth 1 to 2, again when the `--std-visibility` parameter is set to false: For example with TS9, there is a 479% increase from D0 to D1 and a 52% increase from D1 to D2. On the contrary, when the `--std-visibility` parameter is set to true, there is only a minor increase in number of LIs when increasing the value of the `--nesting-depth` parameter: With TS9, there is a 2.3% increase from D0 to D1 and from D1 to D2; with AllTS, there is a 2.7% increase from D0 to D1 and a 0.4% increase from D1 to D2. The latter observation is due to the fact that good programming practices were followed when constructing the software, whereby encapsulation is enforced by making fields private or protected. As a result, setting `--std-visibility` to true prevents Daikon from using many more variables (only a few are public) as `--nesting-depth` increases.

Table 4.1 Number of JEs and LIs (depending on the nesting depth) for each class

Class Name	No of JE's	TS9 noSTD			TS9 STD			AllTS STD			AllTS no STD
		D2	D1	D0	D2	D1	D0	D2	D1	D0	D0
ATM	22	8386	8067	325	322	322	319	439	439	419	419
Bank	32	3894	3299	2887	1802	1745	1745	3245	3245	3207	7403
Transaction	10	7319	3379	39	45	45	39	48	48	38	38
Money	18	87	87	37	87	87	37	90	90	42	42
WithdrawlTransaction	5	1764	946	22	22	22	22	36	36	36	36
TransferTransaction	6	1393	812	25	25	25	25	28	28	28	28
InquiryTransaction	3	1231	665	9	9	9	9	13	13	13	13
DepositTransaction	5	1115	599	14	14	14	14	28	28	28	28
Session	17	6225	2711	91	91	91	91	168	168	168	168
OperatorPanel	4	44	44	43	43	43	43	57	57	57	57
ReceiptPrinter	12	77	77	34	34	34	34	33	33	33	33
CardReader	10	39	39	39	39	39	39	57	57	57	57
EnvelopeAcceptor	2	11	11	11	11	11	11	22	22	22	22
CashDispenser	5	22	22	9	9	9	9	12	12	12	12
Total	151	31607	20758	3585	2553	2496	2437	4276	4276	4160	8356

When the value of parameter `--nesting-depth` is fixed and the value of parameter `--std-visibility` varies, we can make similar observations (using TS9). For `--nesting-depth=0`, setting `--std-visibility` to true reduces the number of LIs by 1.6 times; for `--nesting-depth=1`, setting `--std-visibility` to true reduces the number of LIs by 8.3 times; For `--nesting-depth=2`, setting `--std-visibility` to true reduces the number of LIs by 12.3 times. This reduction in number of LIs is again attributable to good (encapsulation) practices in the source code.

Similar observations can be made when using all the test suites together (AllTS). With the `--std-visibility` parameter set to true, increasing the value of the `--nesting-depth` parameter does not significantly increase the number of LIs: 2.7% increase from D0 to D1; 0.04% from D1 to D2. For AllTS setting the `--std-visibility` parameter to false and the `--`

nesting-depth parameter to 0, the total number of LIs is twice as much as when setting the --std-visibility parameter to true for the same depth.

From Table 4.1 we observe similarities in the number of generated likely invariants for AllTS and TS9. For example when --std-visibility parameter is set to true, for both test suites the number of reported LIs does not increase with an increase in depth for class WithdrawlTransaction, TransferTransaction, InquiryTransaction, DepositTransaction, Session, OperatorPanel, ReceiptPrinter, CardReader, EnvelopeAcceptor and CashDispenser. Also Daikon reports the same number of likely invariants for depth 2 and 1 for both test suites when the --std-visibility parameter is set to true for class ATM, Transaction and Money.

From our observations we can reasonably foresee that, for AllTS, Daikon will report the same number of LIs for class CardReader, Money, OperatorPanel, ReceiptPrinter CashDispenser and EnvelopeAcceptor at depth 2 and 1 for the --std-visibility parameter being set to false. For other classes we might observe a huge increase in the total number of reported LIs with the increase of depth. From our observations we are also predicting that most of those LIs will be over unrelated variables: they will compare apples and oranges. Therefore, we conclude that TS9 can be considered representative of what we would obtain with AllTS for other configurations of Daikon that we were not able to execute with respect to these general observations (recall section 3.2).

Another observation can be made from Table 4.1. The total numbers of LIs generated by Daikon using different configurations with TS9 is smaller than with AllTS. This is not entirely

surprising since the additional coverage reached with AllTS provides additional opportunities (i.e., more variables being observed) for Daikon to discover new LIs. However, one could also argue that AllTS and the associated increase in coverage gives Daikon more traces for the code covered by TS9, which should help Daikon discard LIs that are identified when using TS9. Studying to what extent those two aspects play a role in the results is out of the scope of this thesis.

One can also observe that the results vary a great deal from class to class. Consider for instance TS9 and noSTD. For class `EnvelopeAcceptor`, the number of LIs does not change as `--nesting-depth` increases; On the contrary, the number of LIs at depth 1 for class `InquiryTransaction` is 94 times the one obtained at depth 0. Also, in some cases, using a depth of 2 does not increase the number of LIs from depth 1 (e.g., class `OperatorPanel`) whereas for others the number of LIs increases a lot from depth 1 to depth 2 (more than doubles for class `Session`). Variations of increase with depth are due to the characteristics of the classes. For instance, class `EnvelopeAcceptor` does not have any attribute (even inherited), other than a few attributes related to the GUI, and its variables (local variables, parameters, return values) are all of primitive types (actually either `String` or `Boolean`). So increasing depth does not increase the number of variables to be monitored and compared in the context of class `EnvelopeAcceptor`, i.e., for program points in this class, and there is therefore no difference in terms of number of reported LIs. On the contrary, class `InquiryTransaction` has methods with parameters of type `ATM`, `Session`, and `Bank`, which all have attributes. As a consequence, increasing depth from 0 to 1 drastically increases the number of variables to

compare and therefore the chances of finding relations between these variables. In other words, the many relationships between classes provide as many opportunities to increase the number of variables to consider and therefore the number of reported LIs. We also observe from the table that both classes `ReceiptPrinter` and `Session` have the same number of JEs but the behavior of Daikon is very different. The reason is that `ReceiptPrinter` does not have any attribute, even private ones, and it has only two methods: one is the empty constructor and the other simply sets values to an array representing information to print on the receipt. On the other hand, class `Session` has attributes of type `ATM`, `Bank`, and `Transaction` and three private static attributes of type `int`. Also, it has six methods with complex bodies compared to class `ReceiptPrinter`. Class `CardReader` has five attributes of type `int` and one attribute related to the GUI but no attribute of other class type, and class `EnvelopeAcceptor` has three attributes related to the GUI. Since both classes do not have any coupling with other classes, a change of depth does not affect results for these classes.

Another interesting thing to observe is that the number of LIs increases much more as depth increases for class `ATM` than for class `Bank` when using test suite TS9 and setting parameter `--std-visibility` to false. Class `ATM` has private attributes of all of the other main class types whereas class `Bank` has only one private attribute of type `Money`. The setting for the `--nesting-depth` parameter therefore drastically affects results: class `ATM` as fewer LIs at D0 than class `Bank` but then there is a larger increase for `ATM` than for `Bank` when depth increases.

Yet another interesting observation from the table is that, when the `--std-visibility` parameter is set to true, the total number of LIs returned by Daikon is different only for the `ATM`,

`Bank`, `Transaction` and `Money` classes at different nesting depth for TS9 and AllTS. These classes have one or more attributes whose type is another class in the class diagram, which would give more opportunities to Daikon to involve more variables in LIs. However, the `--std-visibility` parameter being set to true, Daikon is largely prevented from doing so since attributes of those classes are private. The variation in numbers of LIs in those classes compared to other classes is rather due to the many more local variables, parameters and return types they involve since these are the classes at the core of the business logic of the ATM software and their methods are more complex than other classes' methods.

Looking at those four classes again, i.e., `ATM`, `Bank`, `Transaction` and `Money`, and results when the `--std-visibility` parameter is set to true, we observe different results with TS9 and AllTS for class `Bank` compared to classes `ATM`, `Transaction` and `Money`: AllTS allows Daikon to report on many more LIs than TS9 for `Bank` than for the other three classes. This is also due to the many local variables, parameters and return types in class `Bank`, rather than attributes of other classes, which, being private, are not used by Daikon (`--std-visibility` parameter being set to true), because `Bank` is a central class in the software. Plus, the increase in coverage of AllTS compared to TS9, although small for class `Bank` in package `atm` (Table 3.3), provides a larger list of those variables to Daikon and therefore more opportunities to involve them in LIs.

4.2 Comparison framework

Given the large difference between the number of JEs and the number of LIs at different depth values, we suspected that many LIs do not match any JE. So we turned our attention to the classification of the LIs according to our comparison framework. Table 4.2 shows, for each depth level, the number of LIs that Daikon reports as class invariant (inv), precondition (pre) or post-condition (post) and that fit our comparison framework (the ten categories discussed in section 3.3) for test suite TS9 without `--std-visibility`. A class invariant LI (inv) is one that Daikon reports for a program point `:::CLASS` or `:::OBJECT`, a precondition LI (pre) is one that Daikon reports for a program point `:::ENTER`, and a post-condition LI (post) is one that Daikon reports for a program point `:::EXIT`. For example, at depth 2, six (class) invariant LIs are found to be equivalent to JE class invariants, 13 precondition LIs are found to be equivalent to JE preconditions, and 61 post-condition LIs are found to be equivalent to JE post-conditions. The table also shows that we did not find any LI that falls in categories JE_Imply and Likely Equiv. The table also sums up the total number of matches (the first four categories of our comparison framework) and the total number of mismatches (the last six categories): the two rows above the last row of the table. These rows confirm our initial intuition: a large majority of the LIs do not match any JE. This is due to an overwhelmingly large number of Semantically Incorrect LIs: a total of 27,219 Semantically Incorrect LIs at depth 2 (86% of LIs), 17,543 at depth 1 (84% of LIs), and 2,571 at depth 0 (71% of LIs); especially in post-conditions: 68% of semantically incorrect LIs at depth 2 are post-conditions, they are 69% and 73% at depth 1 and 0, respectively.

Table 4.2 Classification of LIs at different depth values when compared to JE class invariants (inv), preconditions (pre), and post-conditions (post) for TS9 without `--std-visibility`.

Category	Depth 2			Depth 1			Depth 0		
	Inv	Pre	Post	Inv	Pre	Post	Inv	Pre	Post
Equiv	6	13	61	6	13	56	5	10	43
Likely Equiv	0	0	0	0	0	0	0	0	0
JE_Imply	0	0	0	0	0	0	0	0	0
Partially Imply	20	16	66	19	16	65	21	13	29
Obvious	183	262	1423	158	218	1200	42	60	347
Semantically Incorrect	3068	5667	18484	1797	3629	12117	274	405	1892
GUI Element	103	216	1181	43	55	575	10	0	122
Frame	0	0	27	0	0	27	0	0	14
Dummy	57	143	224	52	152	211	13	28	64
Not useful	26	95	266	19	92	238	4	38	151
Total matches	26	29	127	25	29	121	26	23	72
Total mismatch	3437	6383	21605	2069	4146	14368	343	531	2590
Total LI obtained	3463	6412	21732	2094	4175	14489	369	554	2662

Another observation we can make is that Daikon returns many more post-condition LIs than precondition LIs and class invariant LIs. One reason we put forward and that could explain the fact that Daikon returns many more post-conditions than preconditions, more specifically the fact that Daikon does not return too many preconditions, is the possibility that the programmers have followed a defensive programming approach while implementing the code. With a defensive programming approach, the programmer of a function makes no assumptions regarding conditions under which its method will be called. This typically results in several if-then-else statements at the beginning of the method body so that the method behaves in a

predictable manner despite unexpected inputs. In that case the precondition of a method is reduced to true since as any input is considered valid. Daikon would then observe that the method can be called in whatever situation and would typically not produce preconditions for such a method. However, we did not find an overwhelming number of methods implemented according to a defensive programming approach (we only found six such methods) in our case study system.

Yet another possible explanation for this difference is due to the kinds of expressions one (including Daikon) can write in pre and post-conditions. Depending on the value of `--std-visibility` and `--nesting-depth`, Daikon can typically compare attribute values of the current object, links and attributes values of linked objects, and arguments in a precondition. In a post-condition, Daikon can typically compare attribute values of the current object, links and attributes values of linked objects, arguments, as well as the return value and values of attributes, linked objects' attributes and arguments as they were prior to the execution of the method. Daikon has therefore typically many more values of variables to compare in a post-condition than in a precondition. We identified that with TS9, using a `--nesting-depth` value of 2 and setting `--std-visibility` to false, Daikon reports on 14,289 LIs that use the `orig` keyword, typically used to relate the value of a variable after the execution of a method to the value that variable had prior to the execution of that method. In other words, 14,289 of the 21,732 LIs (Table 4.2) use the `orig` keyword. Therefore 7443 LIs do not use the `orig` keyword; this is in the same order of magnitude as the number of preconditions LIs (i.e., 6412) returned by Daikon

under the same conditions (Table 4.2). We conclude that Daikon returns many more post condition LIs than precondition LIs simply because of the use of the `orig` keyword.

One other observation we can make from the matching cases (i.e., the first four rows in Table 4.2) is that increasing the depth increases the number of matches but not drastically so, especially from depth 1 to depth 2: the total of matching LIs goes from 58 at depth 0, to 75 at depth 1 (29% increase), and 80 at depth 2 (6.6 increase from depth 1). Regardless of the kind of invariant, it pays more (more equivalent hits) to go from depth 0 to depth 1 than from depth 1 to depth 2.

Table 4.3 Classification of LIs at different depth values when compared to JE inv, pre, and post for TS9 with `--std-visibility`

Category	Depth 2			Depth 1			Depth 0		
	Inv	Pre	Post	Inv	Pre	Post	Inv	Pre	Post
Equiv	4	11	45	4	11	45	4	10	43
Likely Equiv	0	0	0	0	0	0	0	0	0
JE_Imply	0	0	0	0	0	0	0	0	0
Partially Imply	21	16	50	21	16	50	21	13	28
Obvious	42	66	337	42	60	328	42	59	327
Semantically Incorrect	103	242	1232	103	238	1218	100	228	1209
GUI Element	10	0	124	10	0	120	10	0	113
Frame	0	0	22	0	0	22	0	0	14
Dummy	8	10	39	8	4	33	8	4	33
Not useful	2	36	133	2	32	129	2	32	137
Total match	25	27	95	25	27	95	25	23	71
Total mismatch	165	354	1887	165	334	1850	162	323	1833
Total LI obtain	190	381	1982	190	361	1945	187	346	1904

A slightly different observation can be made when the `--std-visibility` parameter is set to true for test suite TS9 (Table 4.3) with different nesting depth values. Although the mismatch category with the highest number of elements is still the Semantically Incorrect

category (1,577, 1,557 and 1,537 Semantically Incorrect LIs at depth levels 2, 1, and 0, respectively), these differences due to varying the depth are not as important as in the previous table. The number of Semantically Incorrect LIs contributes to a large extent to the number of LIs, but less than in the previous table: 61% at depth 2, 62% at depth 1, 63% at depth 0. Post-conditions are still contributing a lot to the Semantically Incorrect category: 78% regardless of the depth. One other observation we can make from the matching cases (i.e., the first four rows in Table 4.3) is that increasing depth from 1 to 2 does not increase the number of matches (it stays at 147) whereas increasing depth from 0 to 1 increase matches by 23%. A similar observation can be made when setting the `--std-visibility` parameter to true for all the test suites together (AllTS): Table 4.4.

Table 4.4 Classification of LIs at different depth values when compared to JE inv, pre, and post for AllTS with and without --`std-visibility`

Category	AllTS STD									AllTS noSTD		
	Depth 2			Depth 1			Depth 0			Depth 0		
	Inv	Pre	Post	Inv	Pre	Post	Inv	Pre	Post	Inv	Pre	Post
Equiv	4	8	56	4	8	56	4	6	54	5	8	51
Likely Equiv	0	0	0	0	0	0	0	0	0	0	0	0
JE_Imply	0	0	0	0	0	0	0	0	0	0	0	0
Partially Imply	18	22	55	18	22	55	18	22	37	18	14	39
Obvious	43	106	380	42	106	380	42	106	371	42	105	422
Semantically Incorrect	94	548	2330	94	548	2330	90	536	2286	242	1488	5196
GUI Element	10	4	125	10	4	125	10	4	124	10	4	124
Frame	0	0	27	0	0	27	0	0	20	0	0	20
Dummy	7	4	80	7	4	80	7	4	73	13	30	147
Not useful	4	53	298	4	53	299	4	50	292	6	59	313
Total matches	22	30	111	22	30	111	22	28	91	23	22	90
Total mismatch	158	715	3240	157	715	3241	153	700	3166	313	1686	6222
Total LI obtained	180	745	3351	179	745	3352	175	728	3257	336	1708	6312

Table 4.5 reports data on LIs returned by Daikon that match JEs (data from Table 4.2 and Table 4.3) and allows us to comment on the impact of parameter `--std-visibility`. In general setting the value of parameter `--std-visibility` to true makes Daikon miss a number of correct LIs (i.e., ones that match JEs), and the number of missed LIs increases as depth increases (and especially when going from depth 0 to depth 1): two at depth 0, 30 at depth 1, 36 at depth 2. This is not surprising since setting the value of parameter `--std-visibility` to true (drastically) reduces the set of variables Daikon can compare and many of these variables are involved in JEs. As depth increases, the number of private attributes Daikon can involve in LIs increases and these are variables that Daikon can no longer involved in LIs when setting the value of parameter `--std-visibility` to true.

It is worth also noting that new correct LIs are identified when setting parameter `--std-visibility` to true at depth 1 (two new correct LIs) and depth 2 (one new LI). To explain this, let's have a look at an example. Class `CashDispenser` has the following JE: `$result.equals(_currentCash)`. For TS9 (noSTD) Daikon returned `this._currentCash == return`, `this._currentCash._cents == return._cents`, `return._cents == orig(this._currentCash._cents)`, and `return == orig(this._currentCash)` at depth values 2 and 1. The four LIs returned by Daikon correctly identify the JE: the first two LIs fall in category Equiv and the second two LIs fall in category Partially Equiv. But for TS9, STD at depth values 2 and 1, we obtain `this._currentCash == return` and `return == orig(this._currentCash)` (only two LIs). Hence we missed two LIs but the JE is still correctly identified. This explains

differences in numbers in Table 4.5 and Table 4.6. One thing to notice from this example is that in the latter case both the missing LIs contains private attribute `_cents` of class `Money`. As parameter `--std-visibility` is set to true, Daikon fails to access this attribute from class `CashDispenser`.

Table 4.5 Impact of parameter `--std-visibility` on LIs matching JEs

	Category	Depth 2			Depth 1			Depth 0		
		Inv	Pre	Post	Inv	Pre	Post	Inv	Pre	Post
noSTD	Equiv	6	13	61	6	13	56	5	10	43
	Partially Imply	20	16	66	19	16	65	21	13	29
	Total	26	29	127	25	29	121	26	23	72
STD	Equiv	4	11	45	4	11	45	4	10	43
	Partially Imply	21	16	50	21	16	50	21	13	28
	Total	25	27	95	25	27	95	25	23	71

So far we have counted and categorized likely invariants. We now turn our attention to the high-level design contracts. Table 4.6 reports on the number of high-level design contracts (JEs) for which Daikon returns a matching LI, i.e., a likely invariant that is either one of Equiv, Likely Equiv, JE_Imply, or Partially Imply. For instance, 14 of the 22 contracts (precondition, post-condition or class invariant) of class `ATM` have been identified (at least partially) at depth levels 1 and 2, and 0 for test suite TS9 for both values of the `--std-visibility` parameter, and this number drops to 13 at depth level 2, 1 and 0 for test suite AllTS. Overall, Daikon was able to (partially) recognize a minimum of 57% of the high-level design contracts at depth level 0 for test suite TS9 with `--std-visibility` set to true and a maximum of 68% at depth levels 1 and 2 for test suite TS9 with `--std-visibility` set to false. An interesting observation can be made from Table 4.5 is that increasing the depth level from 1 to 2 did not improve the correct

identification of high-level design contracts, regardless of the test suite used and Daikon configuration.

4.3 Reason for missing JEs

Table 4.6 indicates that at least 45 (32%) of the high-level design contracts have been missed at level 2 and 1 by Daikon for test suite TS9 with `--std-visibility` set to false. We identified these missed contracts and investigated plausible reasons explaining why Daikon missed them. We identified the following cases.

Missed by Daikon: These are contracts that Daikon had the capability to find (i.e., traces exercised the code, the contracts are instances of patterns Daikon uses) but it did not. For instance, Daikon failed to identify class invariant `ATMnumber == 42`, although this kind of expression is part of the patterns it is searching for [5] and Daikon is able to involve `ATMnumber` in other (although semantically incorrect) invariants such a `ATMnumber == atm.Bank._PIN[atm.Bank.SAVINGS]` (it is comparing the identifier of an ATM with a PIN number, which accidently happens to be equal during executions). Another example is the contract post-condition `result.getValue() == first.getValue() + second.getValue()` which is missed by Daikon (although it should be able to identify such a linear relation). Daikon instead identifies two post-conditions, `first._cents < return._cents` and `second._cents <= return._cents`. In yet another example, Daikon fails to report on a method precondition that reads: `_newBalance.getValue() >= _amount.getValue()`; in fact Daikon does not report any precondition for the method. In all

the cases we identified, we checked that the code was adequately executed by our test cases, specifically that each decision was executed and that each exit point was triggered. We identified that 44 of the 49 missed contracts fall in this category. Investigating further why these contracts were missed by Daikon would require looking into the algorithms actually used by Daikon. This was out of the scope of our study.

Table 4.6 Total number of correct identification of high-level design contracts

Class Name	No of JE's	TS9 noSTD			TS9 STD			AllTS STD			AllTS noSTD
		D2	D1	D0	D2	D1	D0	D2	D1	D0	D0
`ATM`	22	14	14	14	14	14	14	13	13	13	13
`Bank`	32	21	21	20	20	20	19	18	18	18	19
`Transaction`	10	6	6	5	5	5	5	5	5	5	5
`Money`	18	15	15	7	15	15	7	17	17	7	7
`WithdrawlTransaction`	5	5	5	4	4	4	4	4	4	4	4
`TransferTransaction`	6	4	4	4	4	4	4	4	4	4	4
`InquiryTransaction`	3	3	3	3	3	3	3	3	3	3	3
`DepositTransaction`	5	3	3	3	3	3	3	4	4	4	4
`Session`	17	10	10	10	10	10	10	12	12	12	12
`OperatorPanel`	4	3	3	2	2	2	2	1	1	1	1
`ReceiptPrinter`	12	5	5	4	4	4	4	4	4	4	4
`CardReader`	10	7	7	7	7	7	7	9	9	9	9
`EnvelopeAcceptor`	2	2	2	2	2	2	2	2	2	2	2
`CashDispenser`	5	4	4	2	2	2	2	2	2	2	2
Total	151	102	102	87	95	95	86	98	98	88	89
Total missing		49	49	64	56	56	65	53	53	63	62
Recognized in %		68	68	58	63	63	57	65	65	58	59

Additional test cases needed: Five of the 49 missed contracts were missed because of inadequate structural coverage of our test suite TS9. For instance, methods `retainCard()` and `doInvalidPINExtension()` are exercised in TS7 but not in TS9. (These methods are exercised when a customer fails to provide a correct PIN after three attempts and as a

consequence the card is kept in the ATM.) Therefore Daikon cannot reverse-engineer the corresponding preconditions and post-condition.

At depth 0 Daikon missed 64 (42%) of the high-level design contracts for test suite TS9 with `--std-visibility` set to false. 59 out of the 64 fall in category *Missed by Daikon* and five out of 64 fall in category *Additional test cases needed*. Six are missed due to the use of nesting depth 0.

When the `--std-visibility` parameter is set to true and test suite TS9 is used, Daikon missed 56 (37%) of the high-level design contracts at depth values 2 and 1. Among the 56, 44 fall in category *Missed by Daikon* and five out of the 56 fall in category *Additional test cases needed*. The remaining seven are missed due to the setting parameter `--std-visibility` to true (fewer variables are involved in likely invariants). At depth 0, 65 (43%) of the high-level design contracts are missing. 53 of them fall in category *Missed by Daikon* and five fall in category *Additional test cases needed*. Seven are missed due to the use of nesting depth 0 along with the value of the `--std-visibility` parameter.

When all test suites are used together (AllTS) with parameter `--std-visibility` set to true, at depth 2 and 1, 53 (35%) high-level design contracts have been missed by Daikon: 44 fall in category *Missed by Daikon* and nine are missed due to the setting of the `--std-visibility` parameter**.**

It is worth noting that we did not find a high-level design contract that was missed by Daikon because the contract had a structure that Daikon would not be able to recognize. In other

words, the relation patterns that Daikon is trying to instantiate with variable values monitored during executions should have allowed Daikon to correctly identify all the missed high-level contracts.

4.4 Results on precision and recall

We first compute the number of elements of the sets TP_{equiv}, FN_{equiv}, and FP_{equiv} and then compute $Precision_{equiv}$ and $Recall_{equiv}$ for two different test suites (i.e., TS9 and AllTS) with different configuration settings of Daikon (Table 4.7). Figure 4.1 shows $Precision_{equiv}$ vs. $Recall_{equiv}$ graphically for different configuration settings of Daikon. A high precision implies that Daikon returned more relevant (i.e., correct) LIs than irrelevant (i.e., incorrect) LIs and a high recall indicates that most of the LIs returned by Daikon have matching JEs (at least partially). In a Precision vs. Recall curve, desirable plots are located at the upper right corner of the graph; in other words, a configuration that leads to a curve above another configuration's curve is considered more desirable.

Table 4.7 Calculation of $Precision_{equiv}$ and $Recall_{equiv}$ (as percentages)

	TS9 noSTD			TS9 STD			AllTS STD			AllTS noSTD
	D2	D1	D0	D2	D1	D0	D2	D1	D0	D0
No. of equivalent LIs	80	75	58	60	60	57	68	68	64	64
TP_{equiv}	80	75	58	60	60	57	68	68	64	64
FN_{equiv}	31527	20683	3527	2493	2436	2380	4208	4208	4096	8292
FP_{equiv}	71	76	93	91	91	94	83	83	87	87
$Precision_{equiv\%}$	0.3	0.4	1.6	2.4	2.4	2.3	1.6	1.6	1.5	0.8
$Recall_{equiv\%}$	53	50	39	40	40	38	45	45	42	42

From Table 4.7 we observe that for TS9 and `--std--visibility` being set to false, at depth 2, 1 and 0 $Precision_{equiv}$ is 0.3%, 0.4% and 1.6%, respectively. That means for these configurations only 0.3%, 0.4% and 1.6% of the returned LIs are correct and therefore most of

the returned LIs are incorrect. Also for the same test suite and the same configurations $Recall_{equiv}$ equals 53%, 50% and 39%, respectively. That means only 53%, 50% and 39% of the high level design contracts are inferred by Daikon.

Another interesting observation can be made from Table 4.7. Parameter `--std--visibility` being true, at depth 2 and 1 the values of $Precision_{equiv}$ and $Recall_{equiv}$ remain same regardless of the test suites and they decrease very marginally at depth 0. We observe that Daikon successfully returned 53% to 39% of high-level contracts.

We also compute the number of elements of sets $FP_{partiallyImply}$, $FN_{partiallyImply}$, and $TP_{partiallyImply.}$ Then we compute $Precision_{partiallyImply}$ and $Recall_{partiallyImply}$. Table 4.8 shows that with `--std--visibility` set to false, for TS9 at depth 2, 1 and 0, $Precision_{partiallyImply}$ equals 0.4%, 0.7% and 3%, respectively. This means that very few correct LIs are returned by Daikon. Table 4.8 shows that Daikon partially infers 34% to 30% of the high level design contracts. Figure 4.2 is the graphical representation of $Precision_{partiallyImply}$ vs. $Recall_{partiallyImply}$ for different configuration setting of Daikon.

Table 4.8 Calculation of $Precision_{partiallyImply}$ and $Recall_{partiallyImply}$ (as percentages)

	TS9 noSTD			TS9 STD			AllTS STD			AllTS noSTD
	D2	D1	D0	D2	D1	D0	D2	D1	D0	D0
$TP_{partiallyImply}$	147	143	116	135	135	114	143	143	126	127
$FP_{partiallyImply}$	31460	20615	3469	2418	2361	2323	4133	4133	4034	8229
$FN_{partiallyImply}$	293	294	264	322	322	262	321	321	277	279
$Precision_{partiallyImly\ \%}$	0.4	0.7	3	5	5	4.6	3.4	3.4	3	1.5
$Recall_{partiallyImply\ \%}$	34	33	31	30	30	30	31	31	31	31

We observe from Table 4.8 that the `--std--visibility` parameter being true, at depth 2 and 1 the value of $Precision_{partiallyImply}$ remains the same regardless of the test suite and it

decreases very marginally at depth 0. But for the same configuration setting the values of $Recall_{partiallyImply}$ remains the same at depth 2, 1 and 0 regardless of the test suite. That means the number of returned correct LIs by Daikon do not increase with an increase in depth.

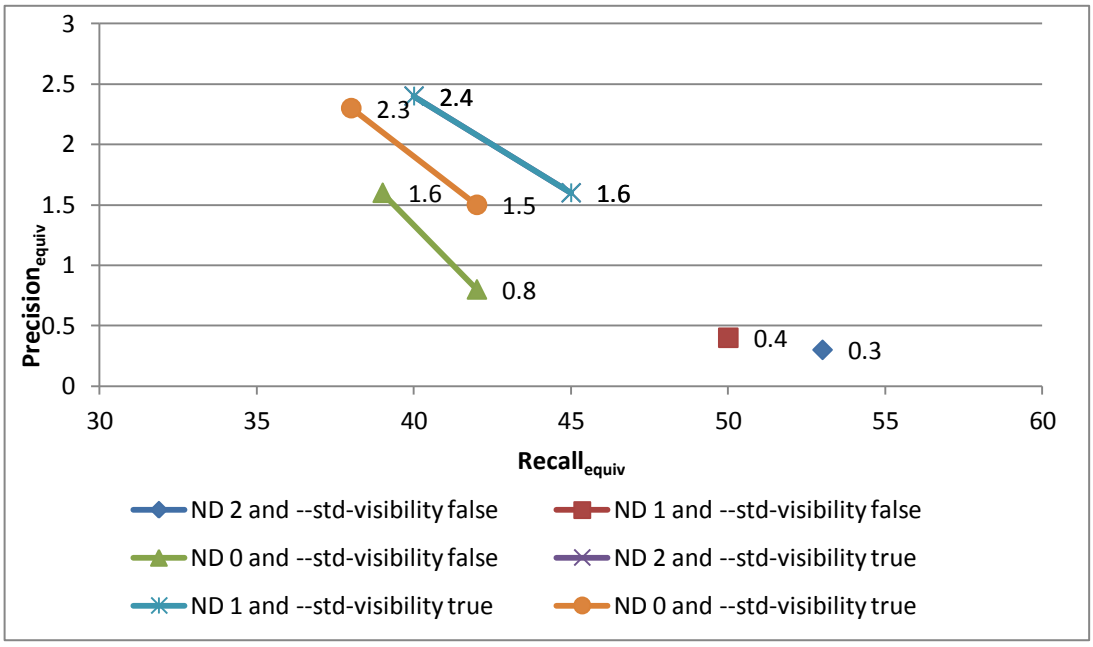

Figure 4.1 $Precision_{equiv}$ vs. $Recall_{equiv}$ for different configuration settings of Daikon

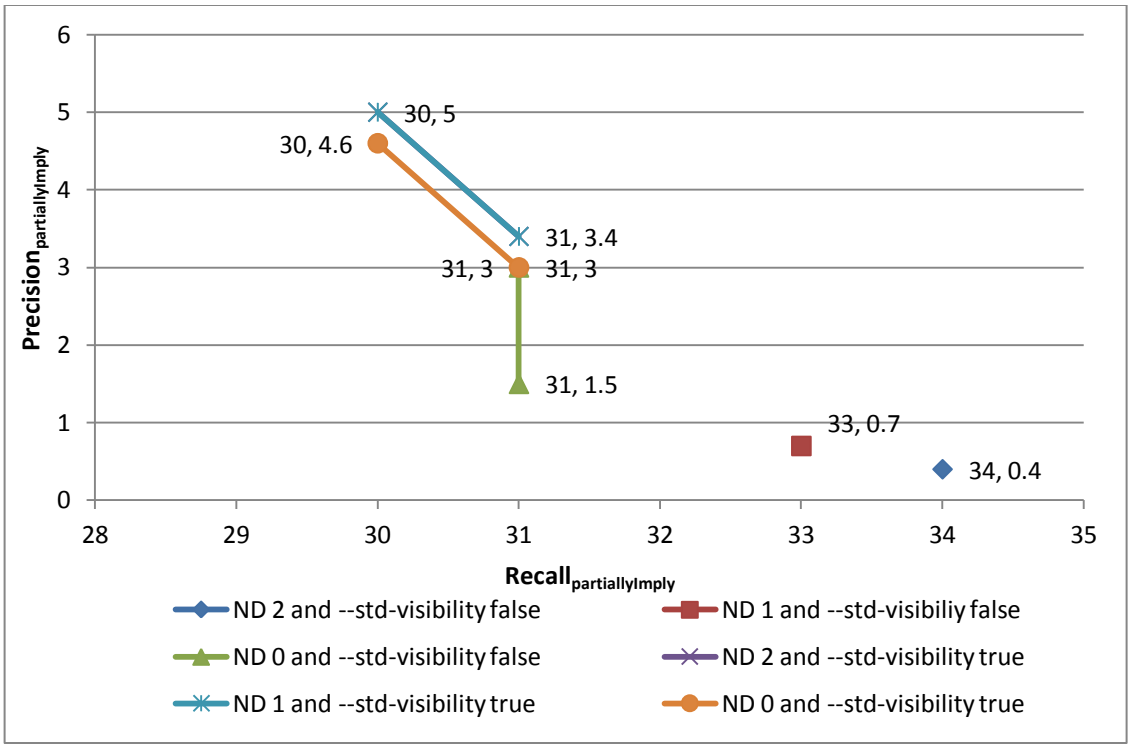

Figure 4.2 Precision$_{partiallyImply}$ vs. Recall$_{partiallyImply}$ for different configuration settings of Daikon

Another observation can be made from Table 4.7 and Table 4.8. That is Precision$_{equiv}$ and Precision$_{partiallyImply}$ have low values for their corresponding Recall$_{equiv}$ and Recall$_{partiallyImply}$ values respectively. The fact that most of the LIs returned by Daikon are incorrect in our experiments leads to the very poor values of Precision$_{equiv}$ and Precision$_{partiallyImply}$.

The value for pair Precision$_{equiv}$ and Recall$_{equiv}$ and also Precision$_{partiallyImply}$ and Recall$_{partiallyImply}$ can be used to determine the quality of any configuration setting of Daikon. Any setting of Daikon with high values for Precision$_{equiv}$ and Recall$_{equiv}$ and also Precision$_{partiallyImply}$ and Recall$_{partiallyImply}$ is considered as good. High values for Precision$_{equiv}$ and Recall$_{equiv}$ and

$Precision_{partiallyImply}$ and $Recall_{partiallyImply}$ is desired i.e., any curve in the upper right corner in the Precision vs. Recall graph is desired. For plotting the $Precision_{equiv}$ vs. $Recall_{equiv}$ (Figure 4.1) and $Precision_{partiallyImply}$ and $Recall_{partiallyImply}$ (Figure 4.2) we consider the obtained values for $Precision_{equiv}$, $Recall_{equiv,}$ $Precision_{partiallyImply}$ and $Recall_{partiallyImply}$ for both test suites TS9 and AllTS together for a configuration setting of Daikon (i.e., each curve has two coordinates). We combined data in order to have more data to compare.

From Figure 4.1 we observe that for `--nesting-depth` 2, 1 (i.e., ND 2 and ND 1) and `--std-visibility` set to false we obtain two points rather than two $Precision_{equiv}$ vs. $Recall_{equiv}$ curves. Similar observation is made from Figure 4.2. This is due to the fact that for AllTS we were unable to study the performance of Daikon with the two pairs of values for "`--nesting-depth 2` and `--std-visibility` false" and "`nesting-depth 1` and `--std-visibility` false". Although we have highest values for $Recall_{equiv}$ and $Recall_{partiallyImply}$ for these two configuration settings, the corresponding values of $Precision_{equiv}$ and $Precision_{partiallyImply}$ respectively are lowest.

From Figure 4.1 we also observe that the curves for `nesting-depth 2` and `--std-visibility` true (ND 2) and `nesting-depth 1` and `--std-visibility` true (ND 1) overlap since they have exactly the same coordinates. Theses curves are in the upper right corner. Also highest values for $Precision_{equiv}$ and $Precision_{partiallyImply}$ are obtained for these configuration settings (from Table 4.7 and Table 4.8). And the corresponding values of $Recall_{equiv}$ and $Recall_{partiallyImply}$ are high. So for our experiment we can consider these configuration settings as good, better than the others.

Chapter 5 Threats to validity

As any experimental activity, what we report in this thesis is subject to threats that can jeopardize the validity of the results and conclusions. We try in this section to discuss them in (hopefully) an as honest as possible way. Threats to the validity of our experiment pertain to (1) our choice of a case study system, (2) the way we used Daikon, and (3) the way we performed the comparison. We discuss these in sequence below.

The case study we used is a simplification of an ATM system. Admittedly, it is not representative of real, industrial size software systems. However, we believe it is nevertheless useful for our experiment and the results we obtained are instructive. Since the high-level design contracts were created in an academic context and have been shown to be adequate substitute for hard-coded oracles for testing purposes [1], thereby suggesting they are of high quality, we believe they constitute a good benchmark for studying the correctness of likely invariants returned by Daikon. Test suite TS9 does not achieve 100% structural coverage (we used control-flow and data-flow criteria). However, the level of coverage we obtain is significant and we found that only five of the 151 JEs (corresponding to two of the 134 design contracts) were missed because of this inadequate coverage. When we used all the test suites together (AllTS), Daikon does successfully identify these two design contracts. Also, the general observation we make about the percentage of likely invariants returned by Daikon that are considered meaningful is similar to what others have observed [14].

With respect to using Daikon, we used the default setting, except for parameters `--nesting-depth` and `--std-visibility`. We encountered difficulties when using all the test suites together with the `--std-visibility` parameter set to false. As discussed earlier, we lacked computer memory, although we tried our best to resolve the issue. We do not believe this is a threat to the validity of our results since (1) the results of AllTS for the `--std-visibility` parameter being true are similar to those with TS9, (2) the results with AllTS and the `--std-visibility` parameter set to true at depth 0 are similar to the ones for TS9 under the same configuration, (3) the structural coverage obtained with TS9 is close to the one for AllTS, and (4) we show that a very low number of JEs are missed because of lack of coverage. The main results discussed in this paper, which are based on TS9, should therefore largely extend to AllTS.

The data we discussed show that we had to evaluate a tremendous number of logical expressions: 31,607 likely invariants were produced with nesting depth 2; and there is therefore a high risk of making mistakes.

Chapter 6 Conclusion

The present thesis described the result of an experiment whereby we studied to what extent likely invariants produced by Daikon match contracts (precondition, post-condition, or class invariants) produced by designers during high-level design (i.e., prior to implementation). Daikon infers instances of pre-defined relations between program variables at different program points (corresponding to the notions of pre-condition, post-condition, and class invariants). Studying the accuracy of such likely invariants is crucial since designers/programmers have a hard time classifying them as correct or incorrect [16].

It is well-known that the accuracy of likely invariants heavily depends on the "quality" of the executions of the program we use, a "quality" that is often measured in terms of structural coverage achieved by those executions (e.g., [7], [15]). Another aspect that affects the quality of those invariants is the execution configuration of Daikon. We studied the impact of two configuration parameters of Daikon: `--nesting-depth` specifies the amount of indirection that is allowed when considering variables whose values need to be monitored and compared to find instances of pre-defined relations; `--std-visibility` is used to restrict (or not) the analysis to variables that are visible from a given program point. In this paper we studied, using a case study system, the impact of different values for these parameters on the quality of produced likely invariants. In doing so we devised a comparison framework to identify to what extent likely invariants match high-level design contracts.

The results we obtain show that overall Daikon is able to infer 57% to 68% of our high-level design contracts, which is in accordance with previous work performed by others [14]. Although this study [14] is related to ours there are important differences as well. We use the high-level design contracts coming from OCL expressions whereas Meyer and colleagues used low-level design contracts (i.e. programmer written contracts embedded in the code). Also the comparison framework is different. We refined classification to identify reasons for mismatch between contracts and likely invariants.

However, our results also show that the correct likely invariants are lost in a mass of incorrect invariants and that the large majority of those incorrect invariants simply compare apples and oranges: between 70% and 80% of the likely invariants reported by Daikon compare apples and oranges. Results show that when setting the `--std-visibility` parameter to false, the amount of incorrect invariants increases drastically as we move from a `--nesting-depth` value of 0 to 1 and 2: for instance with TS9, at depth 1 Daikon reports on 479% more likely invariants than at depth 0, at depth 2 Daikon reports on 52% more likely invariants than at depth 1. Our experiment suggests that setting the `--nesting-depth` parameter to a maximum of 1, regardless of the setting for the `--std-visibility` parameter, is sufficient since the number of high-level design contracts correctly identified by Daikon does not increase significantly when further increasing the depth. We also show that setting the value of parameter `--std-visibility` to true makes Daikon miss a number of correct LIs (i.e., ones that match JEs), and the number of missed LIs increases as depth increases (and especially when going from depth 0 to depth 1). Last, we identified a number of contracts that should have been identified by Daikon:

we believe there was enough coverage; the contracts are instances of variable relation patternsDaikon uses. We calculated $Precision_{equiv}$, $Recall_{equiv}$, $Precision_{partiallyImply}$ and $Recall_{partiallyImply}$. Our result shows that Daikon fully infers 53% to 39% high level design contracts and partially infers 34% to 30 high level design contracts. Our results for $Precision_{equiv}$ and $Precision_{partiallyImply}$ also demonstrate that most of the LIs returned by Daikon are incorrect.

Our result suggests that Daikon should not be used for invariant detection for Software that has huge number of Lines of Code (LOC). The manual analysis of generated likely invariants is time consuming. It reports many likely invariants that are over unrelated variables. When using Daikon, one may perhaps focus on a few classes. But still, there is a risk of trying to find a needle in a haystack. Further modification is required to prevent Daikon from reporting on huge number of incorrect likely invariants.

Future work will necessarily involve replications of our experiment, for instance by applying our comparison framework to case studies reported by others [14]. We also plan to apply the Invariant-coverage criterion proposed by Gupta and Heidepriem [7] to assess the matching quantity and quality of generated likely invariants with contracts produced by designers during high-level design. Additional research is needed to use the AutoInfer tool [18] to generate new contracts and then apply our comparison framework on these newly reported contracts. Another interesting extension of the work can be the analysis of the likely invariant generated by Daikon that runs on combination of test suites, instead of a single test suite or all the test suites. For example, we can combine TS1, TS7, TS8 and TS9 and try executing Daikon

to generate the likely invariants. The result may differ and also we will possible overcome the scaling issue that Daikon has.

References

[1] Briand L. C., Labiche Y. and Sun H., "Investigating the Use of Analysis Contracts to Improve the Testability of Object-Oriented Code," *Software - Practice and Experience*, vol. 33 (7), pp. 637-672, 2003.

[2] Bruegge B. and Dutoit A. H., *Object-Oriented Software Engineering Using UML, Patterns, and Java*, Prentice Hall, 2nd Edition, 2004.

[3] Detlefs D., Nelson G. and Saxe J. B., "Simplify: a theorem prover for program checking," *Journal of the ACM*, vol. 52 (3), pp. 365-473, 2005.

[4] Ernst M. D., The Daikon invariant detector web site, http://groups.csail.mit.edu/pag/daikon/, 2010 (Last accessed December 2012).

[5] Ernst M. D., Cockrell J., Griswold W. G. and Notkin D., "Dynamically discovering likely program invariants to support program evolution," *IEEE Transaction on Software Engineering*, vol. 27 (2), pp. 1-25, 2001.

[6] Ernst M. D., Perkins J. H., Guo P. J., McCamant S., Pacheco C., Tschantz M. S. and Xiao C., "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69 (1--3), pp. 35-45, 2007.

[7] Gupta N. and Heidepriem Z. V., "A new structural coverage criterion for dynamic detection of program invariants," *Proc. IEEE International Conference on Automated Software Engineering*, pp. 49-58, 2003.

[8] Leavens G. T., Baker A. L. and Ruby C., "Preliminary design of JML: A behavioral interface specification language for Java," *ACM SIGSOFT Software Engineering Notes*, vol. 31 (3), pp. 1-38, 2006.

[9] Leavens G. T., Poll E., Clifton C., Cheon Y., Ruby C., Cok D., Müller P. and Kiniry J., "JML Reference Manual," Department of Computer Science, Iowa State University, http://www.jmlspecs.org, 2007.

[10] Liskov B. and Guttag J., *Program development in Java*, Addison-Wesley, 2000.

[11] Meyer B., *Object-Oriented Software Construction*, Prentice Hall, 2nd Edition, 1997.

[12] Mitchell R. and McKim J., *Design by Contract, by Example*, Addison-Wesley, 2001.

[13] Ostrand T. J. and Balcer M. J., "The Category-Partition Method for Specifying and Generating Functional Test," *Communications of the ACM*, vol. 31 (6), pp. 676-686, 1988.

[14] Polikarpova N., Ciupa I. and Meyer B., "A comparative study of programmer-written and automatically inferred contracts," *Proc. ACM International Symposium on Software Testing and Analysis*, pp. 93-104, 2009.

[15] Rad S. K., *Can structural test adequacy criteria be used to predict the quality of generated invariants?*, Thesis, University of Antwerp, Department of Mathematics and Computer Science, 2005

[16] Staats M., Hong S., Moonzoo K. and Rothermel G., "Understanding user understanding: determining correctness of generated program invariants," *Proc. ACM International Symposium on Software Testing and Analysis*, pp. 188-198, 2012.

[17] Warmer J. and Kleppe A., *The Object Constraint Language*, Addison-Wesley, Errata at http://www.klasse.nl/english/boeken/errata.html, 1999.

[18] Wei Y., Furia C., Kazmin N. and Meyer B., "Inferring better contracts," *Proc. ACM International Conference on Software Engineering*, pp. 191-200, 2011.

[19] Dillig, I., Dillig, T., Li, B., & McMillan, K., "Inductive invariant generation via abductive inference", In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications,* pp. 443-456, 2013.

[20] Kuzmina, N., Paul, J., Gamboa, R., & Caldwell, J.,. "Extending dynamic constraint detection with disjunctive constraints", In *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis,* pp. 57-63, 2008.

[21] Hangal, S., Narayanan, S., Chandra, N., & Chakravorty, S., "IODINE: a tool to automatically infer dynamic invariants for hardware designs", In Proceeding of *Design Automation Conference,* pp. 775-778, 2005.

[22] Tillmann, N., Chen, F., & Schulte, W., "Discovering likely method specifications. In *Formal Methods and Software Engineering*", pp. 717-736, 2006. Springer Berlin Heidelberg

[23] Boshernitsan, M., Doong, R., & Savoia, A., "From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing", In *Proceedings of the 2006 international symposium on Software testing and analysis*, pp. 169-180, 2006.

[24] Hangal, S., & Lam, M. S., "Tracking down software bugs using automatic anomaly detection", In *Proceedings of the 24th international conference on Software engineering,* pp. 291-301, 2002.

[25] Ammons, G., Bodík, R., & Larus, J. R., "Mining specifications. In *ACM Sigplan Notices* ",Vol. 37, No. 1, pp. 4-16, 2002.

[26] Whaley, J., Martin, M. C., & Lam, M. S., "Automatic extraction of object-oriented component interfaces", In *ACM SIGSOFT Software Engineering Notes,* Vol. 27, No. 4, pp. 218-228, 2002.

[27] Pytlik, B., Renieris, M., Krishnamurthi, S., & Reiss, S. P., "Automated fault localization using potential invariants", *arXiv preprint cs/0310040*, 2003.

[28] Yang, J., & Evans, D., “Automatically inferring temporal properties for program evolution”, In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on* pp. 340-351, 2004.

[29] Fawcett, Tom., "An introduction to ROC analysis.", *Pattern recognition letters* 27, vol 8 pp. 861-874 , 2006.

[30] Davis, J., and Goadrich, M., "The relationship between Precision-Recall and ROC curves", *Proceedings of the 23rd international conference on Machine learning*, ACM, 2006.