Informatics

Consistency-based Software Fault Localization with Multiple Observations

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Lukas Graussam, BSc
Matrikelnummer 01636304

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Georg Weissenbacher, D.Phil.
Mitwirkung: Sarah Sallinger, MSc

Wien, 5. Oktober 2024

Lukas Graussam   Georg Weissenbacher

Informatics

Consistency-based Software Fault Localization with Multiple Observations

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Lukas Graussam, BSc
Registration Number 01636304

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Georg Weissenbacher, D.Phil.
Assistance: Sarah Sallinger, MSc

Vienna, October 5, 2024

Lukas Graussam   Georg Weissenbacher

Erklärung zur Verfassung der Arbeit

Lukas Graussam, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich nicht an generativen KI-Tools bedient habe.

Wien, 5. Oktober 2024

Lukas Graussam

Danksagung

Zuerst möchte ich mich zutiefst bei meinem Betreuer Prof. Georg Weissenbacher bedanken. Vor allem für seine Verfügbarkeit, Geduld, sowie alle Hilfsstellungen, Diskussionen, Inputs und Feedback, die mir das Weiterarbeiten immer wieder ermöglicht und mir Ansporn gegeben haben. Auch seine exzellenten vorausgehenden Lehrveranstaltungen - vor allem Software Model Checking - die für Vorkenntnisse sorgten und mein Interesse in diesem Gebiet weckten, dürfen nicht unerwähnt bleiben. Außerdem gilt Sarah Sallinger mein großer Dank für ihr detailliertes Feedback, speziell bei der schriftlichen Arbeit, sowie für die motivierenden Gespräche.

Weiters bedanke ich mich bei meiner gesamten Familie, insbesondere bei meinen Eltern Herta & Paul und meinen Großeltern Gertrude & Ernst und Maria. Ohne eure Unterstützung - sowohl mental als auch finanziell - wäre mein gesamter Bildungsweg in dieser Art nicht möglich gewesen.

Zu guter Letzt gilt mein Dank allen Freund:innen, auf die ich mich stets verlassen kann und maßgeblich zu meiner Motivation beigetragen haben.

Kurzfassung

Existierende modellbasierte *Software Fault Localization* (*SFL*) Implementationen haben starke Einschränkungen in ihren Fehlermodellen. Programmstellen, die als Fehlerursache in Frage kommen, sind meistens auf einfache Teilmengen der Programmiersprache *ANSI-C* beschränkt. Weiters sind aktuelle *SFL* Algorithmen nicht in der Lage minimale Ergebnisse zu liefern, wenn mehrere Observations (z.B. Unit Tests) bearbeitet werden. In Praxisanwendungen ist es jedoch üblich, dass eine große Anzahl an Observations verfügbar ist, besonders durch den immer stärkeren Einsatz von Continuous Integration Frameworks, etc.

Um die Einschränkungen von existierenden Algorithmen zu bewältigen, präsentieren wir einen neuen *SFL* Ansatz für *ANSI-C* mit den folgenden zwei wichtigsten Verbesserungsmerkmalen. Erstens wird ein klar definiertes Fehlermodell für *ANSI-C* eingeführt, welches auch komplexe Instruktionen wie Pointers und Arrays umfasst. Zweitens ermöglicht uns der Einsatz eines aktuellen *Model Based Diagnosis* (*MBD*) Algorithmus eine effiziente Verarbeitung von mehreren Observations und garantiert, dass die Ergebnismengen minimal sind unter Berücksichtigung aller Observations.

Unsere Auswertungen anhand von konstruierten Programmen, sowie an *TCAS* - das am häufigsten verwendete Benchmark Programm für *SFL* - zeigen wesentliche Verbesserungen im Vergleich zu existierenden Methoden. Eine komplette Implementation unseres Ansatzes ist verfügbar und wurde eingesetzt, um die experimentellen Ergebnisse zu erlangen.

Abstract

State-of-the-art implementations of model-based *Software Fault Localization* (*SFL*) have strong limitations in terms of their fault-model, with candidate locations mostly being restricted to a simple subset of the programming language *ANSI-C*. Moreover, with respect to multiple observations, recent *SFL* algorithms are unable to deliver minimal result sets. However, with the emergence of continuous integration frameworks, etc., it is common that a large number of observations are available in real world applications.

In order to tackle the limitations of existing algorithms we present a novel approach for *SFL* on *ANSI-C*, which has the following two major contributions. First, we introduce a well defined fault-model for *ANSI-C* which also addresses complex features of the language including pointers and arrays. Second, our approach features efficient handling of multiple observations by deploying a recent *Model Based Diagnosis* (*MBD*) algorithm which guarantees that result sets are minimal with respect to all given observations.

We demonstrate significant improvements compared to existing methods on hand-crafted examples as well as on *TCAS* - the most commonly used benchmark for *SFL*. A full implementation of our approach is available and was used to derive the experimental results.

Contents

CHAPTER 1

Introduction

Manually finding bugs in programs with ever growing complexity is a tedious and expensive task for developers. With the rapid advancements of automated verification and testing in software, the discipline of fault localization has attracted a lot of research attention in recent years. Given a bug, the goal is to automatically derive a subset of locations in the program which are candidates for being altered in order to prevent the bug. There is a broad spectrum of techniques tackling this task, though many popular approaches depend on *Model Based Diagnosis* (*MBD*) [WGL+16].

State-of-the-art implementations of model-based *Software Fault Localization* (*SFL*) have strong limitations in terms of their fault-model, with candidate locations mostly being restricted to a simple subset of the programming language *ANSI-C*. For example, concepts like pointers or arrays and the respective modeling for fault localization, i.e., their fault-models, are not addressed in the literature. A single observation, i.e., a single configuration of inputs or a trace leading to a bug, is processed by the majority of currently available solutions. However, in the real world software industry it is common that a large number of observations are available due to the deployment of continuous integration frameworks, regression and unit testing, etc. [HKKT18]. With respect to multiple observations, recent *SFL* algorithms are unable to deliver minimal result sets [IMWM19].

In order to tackle the limitations of existing algorithms, we present a novel approach for *SFL* on *ANSI-C* which has the following two major contributions. First, we present a well defined fault-model for *ANSI-C* which also addresses complex features of the language including pointers and arrays. We demonstrate that significant result improvements are achieved due to our enhanced fault-model both theoretically and practically. Second, our approach features efficient handling of multiple observations by deploying the recent *MBD* algorithm *Hitting Set Dualization* (*HSD*) [IMWM19]. In fact, our approach separates the creation of the model from the process of finding diagnoses, by creating standard instances of the *MBD* problem. Thus, any other *MBD* algorithm could also be deployed. *HSD* guarantees that results are minimal with respect to multiple observations. We also demonstrate result improvements on *SFL* caused by the improved processing of multiple observations both theoretically and practically.

A full implementation of our approach is available [Gra24] and the model creation part builds upon the popular model-checker *CBMC* [CKL04]. We evaluated our tool on hand-crafted examples, as well as on *TCAS* [HFGO94, DER05], which is the most frequently used benchmark for *SFL* [WGL+16]. Our results from the latter show significant improvements compared to related work. Due to the enhanced fault-model, our tool can handle all *TCAS* versions, including those with array errors, as opposed to other existing approaches.

This thesis is structured in the following way. We start by providing an overview of state-of-the-art *SFL* - especially model-based - in Chapter 2. Then we define the required preliminaries including notations and assumed knowledge in Chapter 3, followed by the definition of our enhanced fault-model in Chapter 4, including theoretical examples supporting each decision concerning the fault-model. Advantages of multiple observations in *SFL* are then elaborated in Chapter 5, followed by a brief presentation of our implementation in Chapter 6. We present the final results of our evaluations in Chapter 7 and conclude the thesis in Chapter 8.

CHAPTER 2

State-of-the-art

Since we use standard *MBD* algorithms for our fault localization, there are two main branches of related work to be taken into account. First, we examine *SFL* literature with the focus on model-based *SFL* on *ANSI-C*. Second, recent advances and algorithms for *MBD* are considered.

2.1 *SFL*

A classification of *SFL* techniques was recently introduced in 2018 by analyzing 273 papers in the survey [ZLAA19]. It defines the following categories in descending order by popularity: Spectrum-based, Miscellaneous, Statistical-based, Information retrieval, Hybrid, Data-mining, *Model Based Diagnosis*, Mutation-based, Slice-based, Machine learning and Program state-based. The most popular is Spectrum-based with 41% of analyzed papers, whereas 4% of papers belong to model-based techniques. The previous survey [WGL$^+$16] with less focus on categorization, but more on common *SFL* issues, examined 331 papers in 2016. They define similar categories, but 35% of their papers are counted as Spectrum-based and 19% as model-based. Each category has different advantages and disadvantages, but due to the otherwise too large scope, we solely focus on model-based techniques in this thesis.

It was first shown in 1993 that model-based diagnosis can be applied to logic programs by Console et al. [CFD93]. Wotowa et al. introduced a prototype in 2002 where a model is created from Java programs (supporting a subset of the language) and the original hitting set algorithm by Reiter [Rei87] is deployed to obtain diagnoses, i.e., fault locations [WSM02]. Although it is Java based and we focus on *ANSI-C* in this thesis, we mention it because it already implements the idea of separating the creation of the model from the diagnosis algorithm. Alex Groce presented one of the first implementations of model-based *SFL* for *ANSI-C* [Gro04]. It uses *CBMC* in its usual way to create a model and obtain a counterexample (failed execution) to a program and its specification.

It then computes a successful (not violating) execution trace in the model which is as similar as possible to the failed execution, measured by an introduced distance metric. The differences between the successful and the failed execution in the model is finally used as explanation for the fault. The implementation is tested on 5 versions of the *TCAS* benchmark. However, the process is not fully automated in the sense that the user might have to add further constraints to improve results. For example, they add an *ASSUME* constraint to the first *TCAS* version in order guide the tool to the real bug location.

A popular model-based *SFL* approach for *ANSI-C* was introduced by Griesmayer et al. [GSB06] and refined for better performance in [GSB10]. They also use *CBMC* to create a model and obtain a counterexample (failed execution) to a program and its specification. The program is then instrumented in order to create an altered model that (1) contains the values of the counterexample (2) inverts the specification assertions that lead to the counterexample and (3) introduces "abnormal variables" in order to havoc components, as is common in *MBD*. The model checker is then run iteratively on the instrumented program for generating assignments to the abnormal variables. These assignments then serve as the fault locations. While decent results are obtained on the *TCAS* benchmark, the runtimes are very high since it relies on multiple sequential calls to the model checker.

Most recently in 2019, a new *SFL* algorithm was introduced in [BFP19] which builds upon the approach from Griesmayer et al. and tackles the performance issue. While the underlying principle stays the same, individual tasks are created from each failed test case which can run in parallel on multi-core CPUs. In addition, a whitelist of viable fault locations is maintained and shared among tasks in order to reduce the runtime of new tasks right away. The runtime is in line with state-of-the-art methods and the presented results seem very good on the *TCAS* benchmarks. However, a single-fault assumption is made which obviously does not hold in real world applications. The results are also presented only for *TCAS* versions that solely have a single fault. Therefore, the approach and its results are not comparable to related works which do not assume only single faults.

In [JM10], Jose et al. introduce their *SFL* tool called BugAssist. It also uses *CBMC* to create a model of a program and its specification. The model is split into components, "healthy variables" are introduced for havocing components and a failing observation is encoded. In fact, a direct instance of the *MBD* diagnosis problem is created without referring to it as such in the paper. A MAX-SAT algorithm is then deployed to obtain diagnoses by maximizing the healthy components such that consistency is restored. The set of unhealthy components then serve as fault locations. BugAssist shows both good runtime and results on the *TCAS* benchmarks compared to the state-of-the-art.

SNIPER is a recent model-based *SFL* tool for *ANSI-C* which supports multiple faults by Lamraoui et al. [LN14, LN16]. It separates the creation of the model from finding diagnoses and treats the latter as an *MBD* problem. They introduce a novel diagnosis algorithm called *DiagCombine* (*DC*) which handles the combination of results from multiple observations (e.g. failed test cases). The result quality, as well as the runtime is evaluated on the *TCAS* benchmarks and is in line with BugAssist. While the result quality in terms of Code Size Reduction (*CSR*) is slightly weaker than BugAssist, *SNIPER* has far less misses, i.e., result sets where the actual fault location is not found.

The main takeaways from related *SFL* work concerning this thesis are the following. All discussed existing model-based *SFL* approaches for *ANSI-C* focus on the algorithms for finding diagnoses, but not on the fault-model and are therefore limited when it comes to complex instructions like arrays or pointers. The behavior when such instructions are declared faulty is unclear. Note that all mentioned papers evaluated their implementations on the *TCAS* benchmark, but omit the versions containing array faults from their results mostly without explanation. Thus, our thesis lays its focus especially on a well defined fault-model for each instruction type. Further, *SNIPER* is the only existing approach that works with multiple faults and handles multiple observations. However, it can compute a potentially large number of redundant diagnoses, as is shown in [IMWM19]. We tackle this shortcoming in our thesis by deploying the *MBD* algorithm *HSD* which delivers minimal result sets and shows performance improvements compared to *DC*. We also conclude that *SNIPER* and BugAssist are the most related and relevant existing tools and will compare their results against our tool in Chapter 7.

Note that very recently before the submission of this thesis, yet another *SFL* method for *ANSI-C* was published, namely CFaults by Orvalho et al. [OJM24]. It handles multiple observations and does not have a single fault assumption. They also compare their results to BugAssist and *SNIPER*, which confirms that these are the most related implementations. CFaults instruments the source code of input programs to encode both boolean "healthy variables" (using if statements), as well as multiple observations by creating replica blocks of the source code for each observation, sharing only the healthy variables. Finally, a single MAX-SAT instance is obtained and used to get the diagnoses. The results are claimed to be minimal with respect to multiple observations, since all observations are already encoded in the single MAX-SAT instance. While the instrumentation solely on the source code level using if statements has the advantage of being independent from the model-checker, it has the drawback that complex instruction fault-models (e.g. for arrays or pointers) on the modeling level are not possible. Also, the fault-model is not addressed in detail. The implementation of CFaults is evaluated on *TCAS* and C-Pack-IPAs (a new benchmark suite). However, the paper does not provide detailed results for each *TCAS* version like other *SFL* papers, but only compares overall performance and number of diagnoses in diagrams. This is another reason why our thesis does not include comparisons to CFaults in the following sections, besides the fact that we discovered it shortly before our submission.

2.2 *MBD*

MBD has a considerable dedicated amount of existing and ongoing research, for instance [SKFP12, MSKC14, IMM17, IMWM19, CK20, KSL21, ZOTZ23], as well as various applications in different domains. Some examples for practical implementations of *MBD* are listed in the introduction of [IMWM19] and include type error debugging [SSW03], design debugging [SMV$^+$07], debugging of web services [ACG$^+$05], spreadsheet debugging [JS16], axiom pinpointing in description logics [SHCvH07] and debugging of declarative specifications [TCJ08].

Since our implementation uses the *MBD* algorithm as a blackbox, we elaborate only briefly on the most related work on *MBD*. As mentioned in the previous Section 2.1, *DC* is the *MBD* algorithm deployed in the *SFL* tool *SNIPER* [LN14, LN16]. To the best of our knowledge it is the only *MBD* algorithm with multiple observations that also has an implementation in *SFL*. *DC* combines results from individual observations with a pairwise union approach so that fault locations cannot be discarded when it comes to multiple faults. Discarding locations can happen with a naive intersection method as we will show in Section 5.1. However, weaknesses of *DC* include that it can compute a potentially large number of redundant diagnoses, as well as runtime issues, which is shown in [IMWM19].

To tackle the issues of *DC*, the *HSD* algorithm is proposed in [IMWM19], building upon [IMM17]. As suggested by its name - *Hitting Set Dualization* (*HSD*) - it relies on implicit hitting set dualization, a well known principle in *MBD* theory already addressed in the original seminal by Reiter [Rei87], followed by several others. The duality between an explanation and a diagnosis - one being the minimal hitting set of the other - is used in the algorithm to obtain diagnoses. By construction, it does not compute redundant diagnoses and experimental results on the *ISCAS85* benchmark suite show major performance improvements compared to *DC*. Since *HSD* is the most recent *MBD* algorithm with multiple observations that has an available implementation, we deploy it in our *SFL* tool.

Most recently, there have been further developments concerning *MBD* algorithms with multiple observations. However, as no implementations of new algorithms are available, we only mention them briefly in the following. Kalech et al. propose two algorithms for solving *MBD* with multiple observations and compare them against each other using *ISCAS85* [KSL21]. However, the paper targets *MBD* in the medical domain and is therefore based on different assumptions, e.g., sequential observations and intermittent faults. Zhou et al. [ZOZT22] propose further improvements to *HSD*. They use the principle of gate domination in order to compute cardinality minimal diagnoses efficiently. If all outputs of one component A go into the inputs of another component B, then B dominates A. Dominated and non-dominated components are grouped into hard clauses and soft clauses, respectively. Performance improvements compared to *HSD* are shown on the *ISCAS85* benchmark. In the software domain, being dominated would mean that a dominating component appears down on the control flow, for example, if a variable gets overwritten with a new value down on one execution path. The effects of the algorithm on software fault locations would be an interesting topic for future work. Furthermore, Zhou et al. introduce further refinements to their approach in [ZOZZ22] and [ZOTZ23].

CHAPTER 3

Preliminaries

It is required to define a formalism which is detailed enough to reason about the modeling of the *ANSI-C* language, i.e., how different types of instructions are modeled, while avoiding unnecessary complexity. Moreover, it should enable reasoning about fault-models, complying with common notations for *MBD* [Rei87] [NPQW13] [MSKC14].

3.1 CBMC Encoding

A loop-free program P can be modeled in *Static Single-Assignment* (*SSA*) form which we call trace formula $TF(P)$. The tool *CBMC* [CKL04] uses this approach to perform Bounded Model Checking [BCCZ99] on *ANSI-C* programs. We build upon the formalism introduced by *CBMC* [CKY03] and adapt it in order to enable our reasoning about fault-models for *MBD*.

First a program P is preprocessed, such that it does not contain (1) Preprocessor Directives, (2) expressions with side-effects (for example: `a=b++;`) or (3) any loops except for *While-Loop*s. Introducing a bound $k > 0$ allows us to unwind all *While-Loop*s and recursions up to k iterations such that we get a loop-free program. Function calls can be replaced by the actual function body. The resulting simplified program can then be transformed into *SSA* form - the trace formula $TF(P)$. The transformation to $TF(P)$ renames all variables by assigning them an index which is incremented at each *writing* appearance of the variable, i.e., each time a new value is assigned.

In order to map each part of the trace formula to its originating program statement in this thesis, we partition $TF(P)$ into fragments F_i. Each F_i corresponds to a program statement in the original program P, where i denotes the line number as described in Eq. (3.1).

$$TF(P) = \bigwedge_{i \in Lines} F_i \tag{3.1}$$

To give an example for generating a trace formula, let P_1 be the program *maxPlusOne* from Listing 3.1. Since it does not contain Preprocessor Directives, loops, etc., we can transform it directly into $TF(P_1)$ as described in Eq. (3.2).

$$\begin{aligned}
&F_3 = (g_1 = a_1 < b_1) \wedge (g_1 \implies result_3 = result_1) \wedge (\neg g_1 \implies result_3 = result_2)\\
&F_4 = (result_1 = b_1 + 1)\\
&F_6 = (result_2 = a_1 + 1)\\
&F_7 = (a_1 < result_3 \wedge b_1 < result_3)\\
&TF(P_1) = F_3 \wedge F_4 \wedge F_6 \wedge F_7.
\end{aligned} \tag{3.2}$$

Fragment F_3 corresponds to the *If-Then-Else* statement of line 3. Besides the actual condition, it contains two implications, modeling a path merge of the two branches. More detailed, *Guard Conditions* g_1 and $\neg g_1$ would be assigned to lines 4 and 6, respectively. At line 7, the paths of the *If-Then-Else* block would be merged, resolving the *Guard Conditions*. Since this is caused by the *If-Then-Else* in line 3, we still assign the equations to F_3.

```
1 int maxPlusOne(int a, int b) {
2   int result;
3   if (a < b)
4     result = b+1;
5   else
6     result = a+1;
7   ASSERT(a,b < result);
8   return result;
9 }
```

Listing 3.1: Example: maxPlusOne

3.2 Instruction models

In order to reason about fault-models for each *ANSI-C* instruction type, we describe the basic *CBMC* modeling into the trace-formula, i.e., the corresponding fragment F_i for an instruction I. This enables us to argue about the necessary modifications of this basis in Chapter 4 for the purpose of fault localization. Some details are not relevant for the fault-model, for example, the exact pre-processing of expressions with side-effects. We omit anything which is not relevant for our purpose, full details are available in [CKY03].

3.2.1 Assignments

Assignment are instructions, where an *ANSI-C* expression `expr` is assigned to a variable, like in Listing 3.2. Here we assume that pointers or arrays are not contained in `expr`, since we cover these cases specifically in Section 3.2.5 and Section 3.2.6. Otherwise, all expressions within the syntax of *ANSI-C* are possible. Such expressions can be modeled trivially in $TF(P)$ as an equality. The corresponding fragment F_1 is described in Eq. (3.3), assuming it is the k^{th} *writing* appearance of variable x in $TF(P)$.

```
x = expr;
```

Listing 3.2: General Assignment

$$F_1 := (x_k = expr) \tag{3.3}$$

3.2.2 Assertions

A common way to add specification to a program is through assertions which can be checked by test cases and are typically used when it comes to model-based verification tools. A general assertion is listed in Listing 3.3 and the corresponding fragment F_1 of $TF(P)$ is given in Eq. (3.4). Adding the negation of the asserted expression to $TF(P)$ enables model-checkers like *CBMC* to verify whether a violating trace through the program exists by checking the formula for satisfiability - for which various efficient algorithms exist [GZ17]. If multiple assertions occur in $TF(P)$, say $a_1 \dots a_n$, then they would be re-arranged into a clause in order to check if one is violated $(\neg a_1 \vee \dots \vee \neg a_n)$.

```
assert(booleanExpression); // example: assert(a!=1 && b>c)
```

Listing 3.3: General Assertion

$$F_1 := \neg booleanExpression \tag{3.4}$$

3.2.3 If-Then-Else

In a general *If-Then-Else* statement the condition is added as *Guard Condition* to its respective block internally in *CBMC* to keep track of possibly nested *If-Then-Else* statements. At the end of each *If-Then-Else* these *Guard Conditions* are then resolved as we demonstrate in the following.

```
if(cond1) {
  A; // guard: cond1
}
else if(cond2) {
  B; // guard: (not cond1) and cond2
}
else {
  C; // guard: (not cond1) and (not cond2)
}
```

Listing 3.4: General *If-Then-Else*

In Eq. (3.5) we describe the corresponding fragments in the trace formula for the general *If-Then-Else* from Listing 3.4. The blocks A, B, C are evaluated recursively which we explain in more detail after the general case. The fragment F_1 contains the formula Φ, which describes the path merge after the *If-Then-Else* block, resolving the *Guard Conditions* Eq. (3.6). Informally, set V contains all variables x which appear with a *write-access* in at least one fragment $\{F_2, F_5, F_8\}$ corresponding to the blocks $\{A, B, C\}$ Eq. (3.7). Depending on the *Guard Conditions*, we assign each variable $x \in V$ with a new index to its last indexed value x_t of the respective block $T \in \{A, B, C\}$. If x is not assigned in block T, then x_t is the last value before the *If-Then-Else* Eq. (3.8).

$$
\begin{aligned}
F_1 &:= (guard1 = cond1) \wedge \Phi \\
F_2 &:= TF(A) \\
F_4 &:= (guard2 = cond2) \\
F_5 &:= TF(B) \\
F_8 &:= TF(C)
\end{aligned}
\tag{3.5}
$$

$$
\Phi := \bigwedge_{\forall x \in V} x_{max(a,b,c)+1} = \begin{cases} x_a, & guard1. \\ x_b, & \neg guard1 \wedge guard2. \\ x_c, & \neg guard1 \wedge \neg guard2. \end{cases}
\tag{3.6}
$$

$$
V := \{x : x \text{ gets } \textit{written} \text{ in one of } \{F_2, F_5, F_8\}\}
\tag{3.7}
$$

$$
x_t := \text{last appearance of } x \text{ in block } T,\ T \in \{A, B, C\}
\tag{3.8}
$$

As an example for the recursive evaluation of *If-Then-Else* blocks consider the program in Listing 3.5. When encountering the nested *If-Then-Else* in line 2, its condition *cond*2 is added as conjunction to the already existing *guard*1 from the outer *If-Then-Else*. At the end of each *If-Then-Else* block, here line 4 and line 6, the respective *Guard Conditions* can then be resolved analogously as described before.

```
1  if(cond1) {
2    if(cond2) {
3      A; // guard2: guard1 and cond2
4    }
5    B; // guard1: cond1
6  }
```

Listing 3.5: Recursive *If-Then-Else*

3.2.4 Loops

Since loops are unrolled, i.e., replaced by an *If-Then-Else* statement for every iteration, the modeling of loops in $TF(P)$ is reduced to the recursive evaluation of *If-Then-Else* described in Section 3.2.3. Every type of loop in *ANSI-C* can be re-written into an equivalent *While-Loop*. The code for a general *While-Loop* (see Listing 3.6) is unwound up to depth k as described in Listing 3.7. The loop body A is copied k times and each copy is guarded by the loop condition. Since the unwinding depth k must be chosen by the user, the *CBMC* modeling introduces an unwinding assertion at the last iteration to check that more iterations are impossible. If there are possible executions of the program with more than k loop iterations the assertion would be violated when performing model-checking and the user can increase the unwinding depth k.

```
while(cond) {
  A;
}
```

Listing 3.6: General loop

```
 if(cond) {
  A; // copy 1
  if(cond) {
    A; // copy 2
    ...
      if(cond) {
        A; // copy k
        assert(!cond); // Unwinding assertion
      }
    }
  }
}
```

Listing 3.7: Loop Unwinding

3.2.5 Pointers

In order to deal with pointers, which are commonly used in *ANSI-C*, the *CBMC* modeling introduces a function which de-references pointer occurrences in $TF(P)$. Since a lot of cases have to be considered, we only show the essence by the means of an example, which is relevant to argue about fault-models in Chapter 4. For full details we again refer to [CKY03] .

Consider Listing 3.8 containing simple pointer operations with the according fragments of $TF(P)$ in Eq. (3.9). An address is assigned to pointer p in each branch of the *If-Then-Else*. Generally, a pointer variable is re-indexed (*SSA* index) at each assignment (write operation to the pointer variable) like any other variable essentially recording all possible addresses, as in F_3 and F_5. Every time a pointer needs to be de-referenced, each possible address of the occurrence is encoded in $TF(P)$ by the de-reference function, as in F_6. Essentially, a case-split for each possible address is created and the formula always has a default case. In this example, the pointer can have either the address of a or b. The resulting formula checks whether $(p = address(a))$ or not, leaving $(p = address(b))$ as the default case.

A memory model is maintained which maps each object, i.e., dynamically allocated memory and variables, to a memory segment with respective object size [CKY00]. Variables are introduced, describing whether memory objects are active or dead, e.g., whether dynamic memory is (de-)allocated. Using this model, multiple assertions $\Pi()$ are introduced in $TF(P)$ at each pointer de-reference, verifying its safety, as in Eq. (3.10). These *Pointer Assertions* include:

- Pointer is not null
- Pointer is not invalid (e.g. uninitialized)
- Pointer is not dead (e.g. de-allocated memory)
- Pointer in object bounds (e.g. incremented too often)

We omit the encoding of the memory model itself in $TF(P)$, since it cannot be assigned to a fragment and is not required for the fault-model reasoning in Chapter 4. Further, we do not encode the *CBMC* negation and re-arrangement of each single assertion in $TF(P)$ as described in Section 3.2.2 for simplicity in this example.

```c
int a, b, *p;
if(a > b) {
  p = &b; }
else {
  p = &a; }
*p = 5;
```

Listing 3.8: Pointer Example

$$
\begin{aligned}
F_2 &:= (g_1 = a_1 > b_1) \wedge (g_1 \implies p_3 = p_1) \wedge (\neg g_1 \implies p_3 = p_2)\\
F_3 &:= p_1 = address(b_1)\\
F_5 &:= p_2 = address(a_1)\\
F_6 &:= a_2 = ((p_3 = address(a_1))\ ?\ 5 : a_1) \wedge\\
b_2 &= ((p_3 = address(a_1))\ ?\ b_1 : 5) \wedge \Pi(p_3)
\end{aligned} \tag{3.9}
$$

$$
\Pi(p_3) = in\_bounds(p_3) \wedge not\_null(p_3) \wedge not\_dead(p_3) \wedge \ldots \tag{3.10}
$$

3.2.6 Arrays

Large or dynamic memory arrays are modeled as pointers in $TF(P)$ which we already covered in Section 3.2.5. However, *CBMC* introduces a distinct modeling for constant arrays up to a definable size. A general array with constant size *c_size* is defined in Listing 3.9, with a *write* and *read* operation in lines 2 and 3, respectively. The array a is re-indexed at every *write* in $TF(P)$ and a case split sets each array position k to the assigned expression *expr* if the index i matches the position - or to its previous value otherwise Eq. (3.11). When *reading* from an array, the value at the according position is used in $TF(P)$ without modifying the array Eq. (3.12). Additionally, *CBMC* introduces

Bound Assertions $B(array, index)$ in $TF(P)$ for every array operation, i.e., *read* and *write*, in order to allow only valid indices inside the bounds given by the constant array size Eq. (3.13). Again note that we do not encode the *CBMC* negation and re-arrangement of each single assertion in $TF(P)$ as described in Section 3.2.2 for simplicity.

```c
int i, j, a[c_size], v;
a[i] = exp;
v = a[j];
```

Listing 3.9: Array Example

$$F_2 := (\bigwedge_{0\leq k<c\_size} a_2[k] = (i_1 = k\ ?\ exp : a_1[k])) \wedge B(a_1, i_1) \tag{3.11}$$

$$F_3 := (v_2 = a_2[j_1]) \wedge B(a_2, j_1) \tag{3.12}$$

$$B(a_1, i_1) = i_1 \geq 0 \wedge i_1 < c\_size \qquad B(a_2, j_2) = j_1 \geq 0 \wedge j_1 < c\_size \tag{3.13}$$

CHAPTER 4

Fault-Model

As already mentioned in Chapter 2, existing model-based approaches for *SFL* on *ANSI-C* mostly concentrate on the algorithms operating on the model and localizing the faults rather than the modeling itself. However, this leaves uncertainty about the concrete meaning of an instruction or program line that is declared faulty via the model. In order to tackle this lack, we focus on a well defined fault-model. To derive our fault-model, we examine the effects of different design decisions concerning the model for each *ANSI-C* instruction type in Section 4.2.

Furthermore, the problem of finding faulty components in a model can be treated as an instance of the general problem of *Model Based Diagnosis* (*MBD*) which has a considerable dedicated amount of existing and ongoing research, as already elaborated in Chapter 2. While some of the existing model-based *ANSI-C* fault localization implementations are not direct instances of *MBD* (e.g. [Gro04]), some are so without referring to it as such (e.g. [JM10]), and others explicitly treat the problem as an *MBD* instance (e.g. [LN14, LN16]). We make use of the wide research and implementations of *MBD* for our fault localization approach by defining our fault-model for *ANSI-C* such that it is exactly an instance of the general *MBD* problem.

4.1 *Model Based Diagnosis*

The objective of *MBD* is to find diagnoses of a faulty system, the principal underlying definitions are introduced in [Rei87] [dKW87]. Notations are similar but not always identical in different *MBD* literature, hence we describe the notations used in this work in the following. A system is split into n components as defined in Eq. (4.1).

$$COMPS := \{c_1, \dots c_n\} \tag{4.1}$$

Each component can be declared healthy or unhealthy which is represented via a predicate $h(c), c \in COMPS$. If the predicate $h(c)$ is $true$, the component c it is declared healthy

[CK20]. In literature the opposite is also commonly used such that components are declared *abnormal* via a predicate, for instance in the original work [Rei87]. In order to avoid confusion we stick to *healthy* predicates. In Eq. (4.2) the system description SD is formalized with f_c as an encoded component c and the according healthy predicate $h(c)$ which enables or disables the component by the implication.

$$SD := \bigwedge_{c \in COMPS} (h(c) \implies f_c) \tag{4.2}$$

Typically a system has a number of observable inputs and outputs. A concrete assignment of input and output values is described as an observation OBS. The diagnosis problem arises when the system is faulty such that it is inconsistent with an observations when all components are declared healthy as in Eq. (4.3).

$$SD \wedge OBS \wedge \bigwedge_{c \in COMPS} h(c) \models \bot \tag{4.3}$$

A diagnosis is a set of components Δ, such that consistency is restored when all $c \in \Delta$ are declared unhealthy as in Eq. (4.4). The task of *MBD* is to find such a diagnosis Δ with minimal cardinality.

$$SD \wedge OBS \wedge \bigwedge_{c \in \Delta} \neg h(c) \wedge \bigwedge_{c \in COMPS \setminus \Delta} h(c) \not\models \bot \tag{4.4}$$

Note that we use the *consistency-based* diagnosis model where an unhealthy component c can behave arbitrarily since its encoding f_c is havoced by the implication in Eq. (4.2). In *MBD* literature also *abductive* diagnosis models are described, where only defined behavior modes are assigned to unhealthy components [CK20].

In Chapter 3 we showed how an *ANSI-C* program P can be modeled into a trace formula $TF(P)$, which can be viewed as the system description of P. Both $TF(P)$ and the system description SD from Eq. (4.2) can be encoded to *conjunctive normal form* (*CNF*). For localizing faults in P using *MBD* algorithms we modify $TF(P)$ by arranging it in components c_i as defined in Eq. (4.1) and inserting according implications with healthy predicates $h(c), c \in COMPS$ analogously to Eq. (4.2). We call the modified trace formula $TF_M(P)$, corresponding to the *MBD* system description ($SD := TF_M(P)$). Furthermore, we define a component c_i to refer to program line i in the original program P, thus the granularity of our fault localization is based on source code lines. The fault-model, i.e., the meaning of a component c_i (line i) which is declared faulty, is then defined according to what parts of $TF_M(P)$ are havoced with the healthy predicate $h(c_i)$. We argue about the effects of our defined fault-model per instruction type in Section 4.2.

In software development, faults are commonly detected via failed test cases or counter-examples provided by model checking tools. Both provide a concrete assignment of inputs, as well as outputs that deviate from the asserted results. Such an assignment of inputs together with expected results can trivially be encoded in $TF_M(P)$ and used as observation OBS. A general instance of the diagnosis problem Eq. (4.3) is then directly described by the modified trace formula as it encodes the system description, including healthy predicates per component, as well as observations which cause inconsistency.

4.2 Instruction Fault-Models

As already mentioned, the fault-model for an instruction is defined based on its corresponding fragment in $TF_M(P)$ and especially for which parts *healthy predicates* are inserted. In our fault-model, some parts of $TF_M(P)$ are in fact never havoced, i.e., not affected by any *healthy predicate*. This applies for example to *Assertions* as described in Section 4.2.2. However, the general *MBD* system description Eq. (4.2) is solely described by components and every component is completely havoced by its assigned *healthy predicate*. Therefore, we extend the definition by a rest part f_r in Eq. (4.5) encoding all parts of our model which must always stay valid.

$$SD := f_r \wedge \bigwedge_{c \in COMPS} (h(c) \implies f_c) \tag{4.5}$$

Note that this is still a general *MBD* instance since f_r can simply be treated as part of OBS in the diagnosis problem Eq. (4.3). The fault-model $TF_M(P)$ for each instruction type is discussed in the following subsections analogously to the basic model of $TF(P)$ which was introduced in Section 3.2. We keep using the fragment notation also for $TF_M(P)$ in order to clearly map its parts to program statements in the original program P.

4.2.1 Assignments

For a general assignment instruction (see Listing 4.1), it is again assumed that pointers or arrays are not contained in `expr`, identically as in Section 3.2.1. The corresponding fragment F_1 is described in Eq. (4.6), assuming it is the k^{th} *writing* appearance of variable x in $TF_M(P)$.

```
x = expr;
```

Listing 4.1: General Assignment (copy of Listing 3.2)

$$F_1 := h(c_1) \implies (x_k = expr) \tag{4.6}$$

We use the $h(c_1)$ to havoc the complete assignment. If it is declared faulty ($h(c_1) = \bot$) any value can be assigned to x_k within its data type in order to restore consistency.

4.2.2 Assertions

Assertions in the program code are treated as specification and are therefore not havoced in our fault-model. A general assertion is listed in Listing 4.2 and its corresponding fragment F_1 of $TF_M(P)$ is given in Eq. (4.7). Since F_1 is not affected by any *healthy predicate*, it is part of the rest part f_r in the context of our introduced system description SD from Eq. (4.5). As opposed to the basic model from Section 3.2.2, the asserted expressions must not be negated in $TF_M(P)$ since the task of *MBD* is to restore consistency as defined in Eq. (4.4), whereas the basic model uses the negation to find counter-examples instead.

```
assert(booleanExpression); // example: assert(a!=1 && b>c)
```

Listing 4.2: General Assertion (copy of Listing 3.3)

$$F_1 := booleanExpression \tag{4.7}$$

4.2.3 If-Then-Else

The recursive evaluation of $TF_M(P)$ for a general *If-Then-Else* statement (see Listing 4.3) is performed identically as for $TF(P)$ in Section 3.2.3. We extend $TF(P)$ from Eq. (3.5) by inserting *healthy predicates* only for *Guard* assignments, in this case in the resulting fragments F_1 and F_4 of $TF_M(P)$ which are given in Eq. (4.8). There are no *healthy predicates* added to the path merge Φ, leaving it unchanged as defined in Eq. (3.6) for $TF(P)$. In our modified system description SD from Eq. (4.5), Φ is consequently added to the rest part f_r. We argue informally in Example 4.2.1 why this is a good choice.

```
if(cond1) {
  A;
}
else if(cond2) {
  B;
}
else {
  C;
}
```

Listing 4.3: General *If-Then-Else* (copy of Listing 3.4)

$$\begin{aligned}
F_1 &:= (h(c_1) \implies (guard1 = cond1)) \land \Phi \\
F_2 &:= TF_M(A) \\
F_4 &:= h(c_4) \implies (guard2 = cond2) \\
F_5 &:= TF_M(B) \\
F_8 &:= TF_M(C)
\end{aligned} \tag{4.8}$$

Example 4.2.1. Line 1 is declared faulty $h(c_1) = \bot$. Then *guard*1 can be assigned either $\top$ or $\bot$. Since Φ must still hold as defined in Eq. (3.6) the *MBD* solver can only choose between an execution path, i.e., either A or - depending on *guard*2 - B or C. Now assume Φ would also be havoced. Since every variable occuring in one of $\{A, B, C\}$ is re-indexed and assigned in Φ (Eq. (3.6)), all such variables could then get any value by the *MBD* solver. Therefore, the whole *If-Then-Else* block would simply be dropped, losing the possibility of more detailed solutions.

Since loops are unrolled as described in Section 3.2.4 *If-Then-Else* is the only instruction type left describing control flow. Hence, our fault-model generally leaves the control-flow skeleton of the program intact, similarly as already proposed in [LN14] (Section 4.2). By havocing only the assignment of the *guards*, informally an *MBD* solver can only change taken paths in that skeleton for restoring consistency.

4.2.4 Pointers

The basic modeling of pointers introduced in Section 3.2.5 is already complex as it involves the de-reference function, as well as *Pointer Assertions* $\Pi()$. Thus, multiple possibilities have to be considered for the fault-model. Since - to the best of our knowledge - pointers are not specifically treated in existing *ANSI-C* fault localization literature, we elaborate advantages and disadvantages for different approaches for the fault-model.

We distinguish between pointers appearing on the left or right hand side of an assignment since in the first case a pointer or its referenced object is modified, while in the latter case its value is only read. Based on the appearance, *Pointer Assertions* can be treated differently in the fault-model. Furthermore, the de-reference function has to be considered when inserting healthy variables. We summarize these possibilities in the following list consisting of three independent **design decisions (Decs. 1 to 3)** and two possibilities (a or b) for the fault-model are considered each.

1. Right hand side of assignment - *Pointer Assertions*
   a) Havoc only expression Eq. (4.10)
   b) Havoc expression and *Pointer Assertions* Eq. (4.11)
2. Left hand side of assignment - *Pointer Assertions*
   a) Havoc only expression Eq. (4.12)
   b) Havoc expression and *Pointer Assertions* Eq. (4.13)
3. Left hand side of assignment - De-reference
   a) Havoc whole expression after de-reference Eq. (4.13)
   b) Adapt De-reference function, havoc only resolved address Eq. (4.14)

All possibilities have a reference to an equation containing the corresponding fragment in $TF_M(P)$ based on the example from Listing 4.4. We use this example to describe the construction of $TF_M(P)$ since a general formal description would be too complex for pointers. The pointer p gets assigned different addresses and it is de-referenced in lines 6 and 7. The corresponding fragments of $TF_M(P)$ are described in Eqs. (4.9) to (4.14) including different variants for lines 6 and 7. These are associated to the design decisions (Decs. 1 to 3) through the equation references. Depending on the actual choice, the corresponding variant is used. In the following we analyze advantages and disadvantages for each decision.

```c
int a, b, c, *p;
if(a > b) {
  p = &b; }
else {
  p = &a; }
c = *p + a;
*p = 5;
```

Listing 4.4: Fault-Model Pointer Example

$$
\begin{aligned}
F_2 &:= (\mathbf{h}(\mathbf{c_2}) \implies (g_1 = a_1 > b_1)) \wedge (g_1 \implies p_3 = p_1) \wedge (\neg g_1 \implies p_3 = p_2) \\
F_3 &:= \mathbf{h}(\mathbf{c_3}) \implies p_1 = address(b_1) \\
F_5 &:= \mathbf{h}(\mathbf{c_5}) \implies p_2 = address(a_1)
\end{aligned} \tag{4.9}
$$

$$F_6 := \mathbf{h}(\mathbf{c_6}) \implies (c_1 = (((p_3 = address(a_1))\ ?\ a_1 : b_1) + a_1)) \wedge \Pi(p_3) \tag{4.10}$$

$$F_6 := \mathbf{h}(\mathbf{c_6}) \implies (c_1 = (((p_3 = address(a_1))\ ?\ a_1 : b_1) + a_1)) \wedge (\mathbf{h}(\mathbf{c_6}) \implies \Pi(p_3)) \tag{4.11}$$

$$
\begin{aligned}
F_7 := \mathbf{h}(\mathbf{c_7}) &\implies (a_2 = ((p_3 = address(a_1))\ ?\ 5 : a_1)) \wedge \\
\mathbf{h}(\mathbf{c_7}) &\implies (b_2 = ((p_3 = address(a_1))\ ?\ b_1 : 5)) \wedge \Pi(p_3)
\end{aligned} \tag{4.12}
$$

$$
\begin{aligned}
F_7 := \mathbf{h}(\mathbf{c_7}) &\implies (a_2 = ((p_3 = address(a_1))\ ?\ 5 : a_1)) \wedge \\
\mathbf{h}(\mathbf{c_7}) &\implies (b_2 = ((p_3 = address(a_1))\ ?\ b_1 : 5)) \wedge \mathbf{h}(\mathbf{c_7}) \implies \Pi(p_3)
\end{aligned} \tag{4.13}
$$

$$
\begin{aligned}
F_7 := ((p_3 = address(a_1))\ ?\ (\mathbf{h}(\mathbf{c_7}) \implies a_2 = 5) : (a_2 = a_1)) \wedge \\
((p_3 = address(a_1))\ ?\ (b_2 = b_1) : (\mathbf{h}(\mathbf{c_7}) \implies b_2 = 5)) \wedge \mathbf{h}(\mathbf{c_7}) \implies \Pi(p_3)
\end{aligned} \tag{4.14}
$$

Dec. 1: Right hand side *Pointer Assertions*

If a line containing any type of assignment is declared faulty, our fault-model generally havocs the RHS completely such that the written variable can have any value (as described for general assignments Section 4.2.1 and *Guard* assignments Section 4.2.3). However, choosing Dec. 1.a for the pointer model would keep *Pointer Assertions* intact, while the pointer is not used anymore. Thus, the intuitive choice is Dec. 1.b havocing the whole RHS including *Pointer Assertions* which is also the choice of our implementation (see Table 6.1). In Example 4.2.2 a major advantage against Dec. 1.a is shown.

Example 4.2.2. Consider the program in Listing 4.5 which has an error in line 2 using the uninitialized pointer c. Using the fault-model from Dec. 1.a we get the corresponding fragment F_2 of $TF_M(P)$ described in Eq. (4.15). The *Pointer Assertions* $\Pi(c_1)$ for line 2 stay intact even if the line is declared faulty ($h(c_2) = \bot$). Further, $\Pi(c_1) = \bot$ since the pointer c_1 is always uninitialized. Even without any observation (OBS is empty), the resulting MBD problem is unsolvable, since the conjunction with $\Pi(c_1)$ then always evaluates to false. However, using the fault-model from Dec. 1.b as suggested intuitively, the MBD problem is trivially solved by declaring line 2 unhealthy, since the *Pointer Assertions* $\Pi(c_1)$ would also be havoced by $h(c_2) = \bot$ and any value could be assigned to variable a_1 by the solver. Note that in this case there are no addresses assigned to the pointer c, therefore the de-reference function of *CBMC* simply assumes it points to some object *c_object* in the internal memory model of *CBMC* and since this internal object is also uninitialized, the de-referenced value is unrestricted, i.e., any value could be assigned to the target variable a_1 by a solver. This is, however, not relevant for the argument concerning the *Pointer Assertions*.

```c
int a, *c;
a = *c;
```

Listing 4.5: Reading invalid pointer

$$F_2 := h(c_2) \implies (a_1 = c\_object_1) \wedge \Pi(c_1) \tag{4.15}$$

Dec. 2: Left hand side *Pointer Assertions*

When a line is declared faulty where a pointer is de-referenced at a LHS assignment, i.e., the referenced object is written, it is not intuitive whether *Pointer Assertions* should be havoced (Dec. 2.b) or not (Dec. 2.a), since we generally do not modify the variables or objects on the LHS of assignments, but the assigned value (RHS) becomes nondeterministic. Still we can construct faulty programs where havocing *Pointer Assertions* (Dec. 2.b) is superior, as shown in Example 4.2.3, by the same argument as for Dec. 1.

Example 4.2.3. Consider the program in Listing 4.6 which has an error in line 2 writing to the de-referenced and uninitialized pointer c. Using the fault-model from Dec. 2.a we get the corresponding fragment F_2 of $TF_M(P)$ described in Eq. (4.16). The *Pointer Assertions* $\Pi(c_1)$ for line 2 stay intact even if the line is declared faulty ($h(c_2) = \bot$). Further, $\Pi(c_1) = \bot$ since the pointer c_1 is always uninitialized. Even without any observation (*OBS* is empty), the resulting *MBD* problem is unsolvable, since the conjunction with $\Pi(c_1)$ then always evaluates to $false$. However, using the fault-model from Dec. 2.b, the *MBD* problem is trivially solved by declaring line 2 unhealthy, since the *Pointer Assertions* $\Pi(c_1)$ would also be havoced by $h(c_2) = \bot$ and any value could be assigned to the unrestricted, internal object c_object_1 by the solver.

```c
int a, *c;
*c = a;
```

Listing 4.6: Writing invalid pointer

$$F_2 := h(c_2) \implies (c\_object_1 = a_1) \wedge \Pi(c_1) \tag{4.16}$$

Even though Example 4.2.3 shows that implementing Dec. 2.b enables localization of invalid pointer errors on LHS assignments, it follows that such a faulty reported line needs to be interpreted differently. In fact, it means that both the RHS or the pointer on the LHS might be the cause of the error, as opposed to general assignments, where faults are only suggested in the RHS. In other words, reported solutions might abuse that by dropping *Pointer Assertions*, writing to invalid or out-of-bounds addresses is "allowed", as illustrated in Example 4.2.4.

Example 4.2.4. Consider Listing 4.7 where pointer p can either have the address of variable a or an undefined address in line 4, due to the condition in line 2. Using $OBS_1 := \{a = b = c = 0\}$ as input for a test case the condition of the *If-Then-Else* in line 2 would be $false$ so line 3 is not executed and pointer p stays uninitialized. This could cause a segmentation fault at runtime and the test case would fail. Using the fault-model from Dec. 2.b we get the corresponding fragments of $TF_M(P)$ described in Eq. (4.17). With OBS_1 we clearly have $\Pi(p_3) = \bot$ since the pointer is uninitialized in this case. Then line 4 is a valid solution to the resulting *MBD* problem ($h(c_4) = \bot$), since (1) the *Pointer Assertions* $\Pi(p_3)$ are havoced and (2) the assertion encoded in F_5 can be fulfilled since variable a would not even be modified by the assignment statement in line 4. Instead, the *CBMC*-generated internal object p_object would be modified which can be seen as an invalid memory address referenced by the uninitialized pointer p. However, a correction of only the RHS of line 4 is not sufficient for repairing the program, because the pointer p can still be invalid. The result has to be interpreted such that a modification of the pointer on the LHS might also be required. On the contrary, if the fault-model from Dec. 2.a is implemented the *Pointer Assertions* $\Pi(p_3)$ would not be havoced by $h(c_4) = \bot$. Then solely line 4 is not a valid solution to the *MBD* problem, since the *Pointer Assertions* would still be violated with p being uninitialized in obs_1 and the conjunction with $\Pi(p_3)$ in F_4 would evaluate to $false$. A valid solution in this case would be the pair of lines 2 and 4, suggesting to modify the condition in line 2 such that p always gets initialized - in this case to the address of variable a. Also modifying the RHS of line 4 is then necessary to ensure that the assertion encoded in F_5 holds.

```
1  int *p, a, b, c;
2  if(c) {
3    p = &a; }
4  *p = 5;
5  assert(a == b);
```

Listing 4.7: *MBD* solution writing invalid pointer

$$\begin{aligned}
&F_2 := (h(c_2) \implies (guard1 = c_1)) \land p_3 = (guard1 \ ?\ p_2 : p_1)\\
&F_3 := h(c_3) \implies (p_2 = address(a_1))\\
&F_4 := h(c_4) \implies [a_2 = ((p_3 = address(a_1)) \ ?\ 5 : a_1)\\
&\land p\_object_2 = (p_3 = address(a_1)) \ ?\ p\_object_1 : 5) \land \Pi(p_3)]\\
&F_5 := a_2 = b_1
\end{aligned} \tag{4.17}$$

Although we have elaborated how results have to be interpreted with care when it comes to lines with LHS pointer de-references, we use Dec. 2.b in our implementation (see Table 6.1). The fact that instances may become unsolvable with Dec. 2.a, as demonstrated in Example 4.2.3, justifies the trade-off concerning the interpretation of results. It should also be noted that solely for the purpose of demonstration, *Pointer Assertions* are introduced redundantly in $TF_M(P)$ in both F_6 and F_7 in Eqs. (4.10) to (4.14) corresponding to the unmodified pointer p in lines 6 and 7 in Listing 4.4. Our implementation avoids this redundancy by generating *Pointer Assertions* only for the first appearance of an unmodified pointer, which is also the default behavior of *CBMC*. Therefore only the first line violating *Pointer Assertions* might be reported by an *MBD* solver.

Furthermore, the user can choose whether *Pointer Assertions* are generated or not in our implementation (see Section 6.2). If it is not the case, then all clauses with *Pointer Assertions* $\Pi()$ in $TF_M(P)$ are omitted and Decs. 1 and 2 obsolete. This might be useful if performance is an issue and thus traded off against possibly invalid pointer usages in the results. We will show a significant difference in runtime caused by more assertions in $TF_M(P)$ in Section 7.3.3. Still, discarding *Pointer Assertions* can strongly affect results or even cause empty results for faulty programs, as shown in Example 4.2.5.

Example 4.2.5. Consider the program in Listing 4.8, where we extended Listing 4.5 by an assertion. The uninitialized pointer c is de-referenced in line 2 and the resulting value assigned to variable a which is then checked to have value 1 by the assertion in line 3. A test case can trivially fail at runtime since the value of a is non-deterministic. We assume there is no input to the program in this case, i.e., the observation OBS is empty. The corresponding fragments of $TF_M(P)$ are described in Eq. (4.18) with *Pointer Assertions* being completely omitted in our fault-model. Then the resulting *MBD* problem is immediately consistent with all components healthy, i.e., with an empty diagnosis $\Delta = \emptyset$, since no address is assigned to pointer c so the de-reference function of *CBMC* assumes it points to some object c_object in the internal memory model of *CBMC*. Since this internal object is also uninitialized any value can be assigned to it which is encoded by the unrestricted variable c_object_1 in F_2. Finally, the asserted value 1 can simply be assigned to variable a_1 in F_2 by a solver. Essentially, no lines are declared faulty despite the obvious error in the program.

```c
int a, *c;
a = *c;
assert(a == 1);
```

Listing 4.8: Reading invalid pointer

$$\begin{aligned} F_2 &:= h(c_2) \implies (a_1 = c\_object_1) \\ F_3 &:= a_1 = 1 \end{aligned} \tag{4.18}$$

Dec. 3: Left hand side De-reference

In Eq. (4.13) from Dec. 3.a a healthy variable is introduced in $TF_M(P)$ after the de-reference is done which simply havocs the whole expression generated by the de-reference function. This approach might be considered the intuitive choice since it does not require to modify the basic instruction modeling $TF(P)$, but only adding healthy variables on top of it. However, when declared unhealthy, all objects which could be referenced up to the point of the de-reference are havoced by construction (Eq. (4.13)). We can demonstrate that this behavior can cause misleading diagnoses as shown in Example 4.2.6. In order to avoid this problem, we suggest the fault-model Eq. (4.14) from Dec. 3.b where the de-reference function is modified and healthy variables are introduced such that only the referenced object is havoced, enabling a more precise fault localization.

Example 4.2.6. Consider again Listing 4.4 where pointer p can have either the address of variable a or b in line 7, therefore the RHS value 5 is assigned either to a or b. Assume we add the assertion $assert(a < 0)$ in line 8 as specification and the observation $OBS_1 = \{a = 1, b = -1, \}$ as input. With OBS_1, the condition in line 2 then evaluates to $true$ and the address of b is assigned to p in line 3. In line 7, p is de-referenced on the LHS and the value 5 is thus assigned to variable b. The introduced assertion does not hold, since variable $a = 1$ is not modified. Now assume we use the fault-model of Dec. 3.a with the corresponding fragment F_7 of $TF_M(P)$ given in Eq. (4.13). Then line 7 is a valid solution to the resulting MBD problem since all possible variables for p - including a - are havoced and a solver could assign $a = -1$ to fulfill the assertion in line 8. However, it is obvious that the error cannot be corrected by changing the RHS of line 7, given that p cannot have the address of a with OBS_1. Hence, this solution might be considered misleading or even incorrect. On the contrary, say we use the fault-model of Dec. 3.b with the corresponding fragment F_7 of $TF_M(P)$ given in Eq. (4.14). Then line 7 is not a solution to the resulting MBD problem since the value of variable a would not be havoced with the modified de-reference function. An example for a valid solution in this case would be the pair of lines 2 and 7, suggesting to modify the condition in line 2 such that the address of a is assigned to pointer p in line 5. Changing the RHS of line 7 to a negative number is then enough to fulfill the assertion in line 8 in the model, as well as in the actual program.

While we have shown that is beneficial to implement Dec. 3.b, note that our implementation currently only utilizes Dec. 3.a in general, because of the significant implementation effort of modifying the de-reference process (see Table 6.1).

4.2.5 Arrays

When it comes to the fault-model of arrays, similar design decisions as for pointers arise and it is intuitive to treat them analogously. As mentioned in Section 3.2.6, large or dynamic arrays are in fact modeled as pointers, the fault-model from Section 4.2.4 therefore applies directly. But also for small arrays we show that the same arguments can be made, even though the modeling in $TF_M(P)$ is not identical. In fact this section is organized similar as for pointers in Section 4.2.4, beginning with listing design decisions, followed by describing the corresponding fault-model based on an example. Then our choices are again supported by similar arguments to those which have been constructed for pointers. The main differences compared to pointers are (1) the index case split instead of the de-reference function and (2) *Bound Assertions* instead of *Pointer Assertions*.

Again we distinguish between arrays appearing on the left or right hand side of an assignment since in the first case an array is modified, while in the latter case only some value of the array is read. Based on the appearance, *Bound Assertions* can be treated differently in the fault-model. Furthermore, the case split of indices has to be considered when inserting healthy variables for LHS appearances. We summarize these possibilities in the following list consisting of three independent **design decisions (Decs. 4 to 6)** and two possibilities (a or b) for the fault-model are considered each. Note that the decisions coincide with Decs. 1 to 3 from pointers.

4. Right hand side of assignment - *Bound Assertions*
    a) Havoc only expression Eq. (4.19)
    b) Havoc expression and *Bound Assertions* Eq. (4.20)
5. Left hand side of assignment - *Bound Assertions*
    a) Havoc only expression Eq. (4.21)
    b) Havoc expression and *Bound Assertions* Eq. (4.22)
6. Left hand side of assignment - index case split
    a) Havoc whole expression after index case split Eq. (4.21)
    b) Adapt case splitting procedure, havoc only the actual index in array Eq. (4.22)

All possibilities have a reference to an equation containing the corresponding fragment in $TF_M(P)$ based on the example Listing 4.9. This example is used to describe the construction of $TF_M(P)$ since a general formal description would be too complex for arrays. We assume the constant size *c_size* is small enough that the array modeling applies (recall that large and dynamic arrays are in fact modeled as pointers). In line 2, a random index j is read from a and in line 3 a random index i is written. The corresponding fragments in $TF_M(P)$ for the whole program are described in Eqs. (4.19) to (4.23), where the fragments F_2 and F_3 are defined twice. These are associated to the design decisions (Decs. 4 to 6) through the equation references. Depending on the actual choice, the corresponding variant of F_2 and F_3 is used in the fault-model. In the following we analyze advantages and disadvantages for each decision.

```
int i, j, a[c_size], v;
v = a[j];
a[i] = exp;
```

Listing 4.9: Array Example - Fault-Model

$$F_2 := (\mathbf{h}(\mathbf{c_2}) \implies (v_2 = a_1[j_1])) \land B(a_1, j_1) \tag{4.19}$$

$$F_2 := (\mathbf{h}(\mathbf{c_2}) \implies (v_2 = a_1[j_1])) \land (\mathbf{h}(\mathbf{c_2}) \implies B(a_1, j_1)) \tag{4.20}$$

$$F_3 := \mathbf{h}(\mathbf{c_3}) \implies (\bigwedge_{0 \le k < c\_size} a_2[k] = (i_1 = k \text{ ? } exp : a_1[k])) \wedge B(a_1, i_1) \tag{4.21}$$

$$\begin{aligned} F_3 := & \bigwedge_{0 \le k < c\_size} (i_1 = k) \text{ ? } (\mathbf{h}(\mathbf{c_3}) \implies a_2[k] = exp) : (a_2[k] = a_1[k]) \\ & \wedge \mathbf{h}(\mathbf{c_3}) \implies B(a_1, i_1) \end{aligned} \tag{4.22}$$

$$B(a_1, j_1) = j_1 \ge 0 \wedge j_1 < c\_size \qquad B(a_1, i_1) = i_1 \ge 0 \wedge i_1 < c\_size \tag{4.23}$$

Dec. 4: Right hand side *Bound Assertions*

If a line with a RHS array appearance, i.e., read operation on an array, is declared faulty, it is intuitive to havoc the whole RHS including *Bound Assertions*, similar as for pointers. In Example 4.2.7 we show that this is indeed also a good choice for arrays.

Example 4.2.7. Consider the program in Listing 4.10 which reads from the array a with invalid index 3 in line 2. Using the fault-model from Dec. 4.a, the *Bound Assertions* $B(a_1, 3)$ for line 2 would stay intact in $TF_M(P)$ even if the line is declared faulty as described in Eq. (4.24). Even if we assume there is no input to the program in this case, i.e., the observation *OBS* is empty, the resulting *MBD* problem is unsolvable, since the *Bound Assertions* are always violated. However, using the fault-model from Dec. 4.b as suggested intuitively, the *MBD* problem is trivially solved by declaring line 2 unhealthy, since then the *Bound Assertions* $B(a_1, 3)$ would also be havoced by $h(c_2) = \bot$ and any value can then be assigned to variable x_1 by the solver.

```
1  int a[2], x;
2  x = a[3];
```

Listing 4.10: Reading invalid index

$$\begin{aligned} & F_2 := (h(c_2) \implies (x_1 = a_1[3])) \wedge B(a_1, 3) \\ & B(a_1, 3) = 3 \ge 0 \wedge 3 < 2 \end{aligned} \tag{4.24}$$

Dec. 5: Left hand side *Bound Assertions*

Even though it might not be as intuitive whether *Bound Assertions* should be havoced or not for LHS array appearances, i.e., writing to an array, we can show advantages for havocing them (Dec. 5.b) with the same arguments as for LHS pointer de-references. In Example 4.2.8 we show the problem which arises if *Bound Assertions* would not be havoced (Dec. 5.a).

Example 4.2.8. Consider the program in Listing 4.11 which writes to the array a with invalid index 3 in line 2. Using the fault-model from Dec. 5.a, the *Bound Assertions* $B(a_1, 3)$ for line 2 would stay intact in $TF_M(P)$ even if the line is declared faulty as described in Eq. (4.25). Even if we assume there is no input to the program in this case, i.e., the observation *OBS* is empty, the resulting *MBD* problem is unsolvable, since the *Bound Assertions* $B(a_1, 3)$ are always violated. However, using the fault-model from Dec. 5.b, the *MBD* problem is trivially solved by declaring line 2 unhealthy, since then the *Bound Assertions* $B(a_1, 3)$ would also be havoced by $h(c_2) = \bot$.

```
int a[2]
a[3] = 4;
```

Listing 4.11: Writing to an invalid index

$$\begin{aligned} &F_2 := h(c_2) \implies (\bigwedge_{0 \le k < 2} a_2[k] = (k = 3\ ?\ 4 : a_1[k])) \land B(a_1, 3) \\ &B(a_1, 3) = 3 \ge 0 \land 3 < 2 \end{aligned} \tag{4.25}$$

Note that - similar as for LHS pointer de-references - reported lines with a LHS array access need to be interpreted such that not only the RHS but also the array access could be the reason for the error. If a solver declares such a line faulty, the array is not modified at all in the model, since the used index does not match any valid index. This can be seen in Eq. (4.25), where the invalid index 3 is used for a write operation to array a. The case split then assigns the array $a_2[k]$ at all indices k to the previous value $a_1[k]$. However, for lines declared faulty which contain valid write operations to an array, a solver can choose any value for the RHS, including the previous array value such that the array would not be modified. Thus, the behavior cannot be seen as a disadvantage. Our implementation uses Dec. 5.b in order to avoid the problem elaborated in Example 4.2.8 (see Table 6.1). Furthermore, note that our implementation generates redundant *Bound Assertions* only for the first appearance, also similar to *Pointer Assertions*. For example, in Listing 4.12 our implementation would create *Bound Assertions* $B(a_1, i_1)$ only for line 2, since the array accesses in lines 2 and 3 are identical with variable i being unchanged.

```
1 int a[2], x, y, z, i;
2 a[i] = x;
3 y = a[i] + z;
```

Listing 4.12: Redundant Bound Checks

Like *Pointer Assertions*, our implementation lets users choose whether *Bound Assertions* are generated at all with similar effects (see Section 6.2). If they are omitted, then all clauses with *Bound Assertions* $B()$ in $TF_M(P)$ are dropped and Decs. 4 and 5 obsolete. Less assertions cause better performance at the cost of result quality due to possible invalid array accesses. In Example 4.2.9 we show how wrong results can occur when discarding *Bound Assertions*.

Example 4.2.9. Consider again Listing 4.10, where the invalid index 3 is read from array a of size 2 and the resulting value assigned to variable x. Assume we add the assertion $assert(x = 2)$ in line 3. Further, consider the array a as input, x as output and the observation $OBS = \{a[0] = 2, a[1] = 2, x = -1\}$, which could be a failed test case at runtime, since the access $a[3]$ in line 2 is non-deterministic. The corresponding fragments of $TF_M(P)$ are described in Eq. (4.26) with *Bound Assertions* being completely omitted in our fault-model. The resulting *MBD* problem is then immediately consistent with all components healthy, i.e., with an empty diagnosis $\Delta = \emptyset$. since an array access with an invalid index - here $a_1[3]$ - is also non-deterministic in the model and therefore the asserted value 2 can simply be assigned to variable x_1 by a solver. Essentially, no lines are declared faulty despite the obvious error in the program.

$$\begin{aligned} F_2 &:= (h(c_2) \implies (x_1 = a_1[3])) \\ F_3 &:= x_1 = 2 \end{aligned} \tag{4.26}$$

Dec. 3: Left hand side index case-split

The index case-split of LHS array appearances (write-access) can be treated analogously as the de-reference function for LHS pointer appearances. In Eq. (4.21) from Dec. 6.a a healthy variable is introduced in $TF_M(P)$ after the array index case-split is done which simply havocs the whole generated expression. This approach might be considered the intuitive choice since it does not require a modification of the basic instruction modeling $TF(P)$, but only adds healthy variables on top of it. However, when declared unhealthy, the whole array content (every index) is havoced by construction (Eq. (4.21)). We can prove that this behavior can cause misleading diagnoses as shown in Example 4.2.10. In order to avoid this problem, we suggest the fault-model Eq. (4.22) from Dec. 6.b where the array index case-split function is modified such that healthy variables are introduced to only havoc the array content at the current index, enabling a more precise fault localization.

Example 4.2.10. Consider program P from Listing 4.13, where the value of variable x is assigned to array a at index i in line 3 and an assertion checks the value of $a[1] = -1$ in line 4. Assume all variables declared in line 1 - except i - are observable and let the observation $OBS = \{a[0] = 1, a[1] = 1, x = 0\}$. The array a is only modified at line 3 and since $i = 0$ from line 2, it is only modified at index 0 in this case. The assertion in line 4 does not hold, since $a[1] = 1$ is never modified. Assuming we use the fault-model of Dec. 6.a the resulting fragments of $TF_M(P)$ are described in Eq. (4.27). Then declaring line 3 faulty ($h(c_3) = \perp$) is a valid solution to the resulting *MBD* problem since the whole array content - at all indices - is havoced in $TF_M(P)$, including $a_2[1]$ and a solver could assign $a_2[1] = -1$ to fulfill the assertion in line 4. However, it is obvious that the error cannot be corrected by changing the RHS of line 3, even though *Bound Assertion*s are not violated, given that with line 2 still healthy only index 0 of the array a is modified. Thus, this solution might be considered misleading or incorrect. Now suppose on the contrary that we use the fault-model of Dec. 6.b. Then solely line 3 would not be a valid solution to the resulting *MBD* problem since $a_2[1]$ would not be havoced by $h(c_3) = \perp$ with the modified index case-split. A valid solution in this case would be the pair of lines 2 and 3 suggesting to assign value 1 to the index variable i_1 in F_2 (line 2) in addition to modifying the RHS of line 3 so that the asserted value -1 is assigned to $a_2[1]$.

```c
int i, a[2], x;
i=0;
a[i] = x;
assert(a[1]==-1);
```

Listing 4.13: Example: Array index case-split

$$
\begin{aligned}
&F_2 := (h(c_2) \implies (i_1 = 0)) \\
&F_3 := [h(c_3) \implies \bigwedge_{0 \le k < 2} a_2[k] = (k = i_1 \; ? \; x_1 : a_1[k])] \wedge [h(c_3) \implies B(a_1, i_1)] \\
&F_4 := a_2[1] = -1
\end{aligned}
\tag{4.27}
$$

While we have shown that it is beneficial to implement Dec. 6.b, note that our implementation currently only utilizes Dec. 6.a in general, because of the significant implementation effort of modifying the array case-splitting process, similarly as for the pointer de-reference process (see also Table 6.1).

CHAPTER 5

Multiple Observations

Testing and verification is crucial in the software industry and the automation of such processes is an ever growing discipline. Various methods and tools like continuous integration and deployment (CI/CD) or regression and unit testing are increasing in popularity [HKKT18]. As a result, a large number of test cases (observations) are typical, leading to possibly many failed test cases. The question arises how multiple observations can be processed efficiently when it comes to *SFL*. As mentioned in Section 2.2, the *MBD* algorithm *Hitting Set Dualization* (*HSD*) from [IMWM19] processes multiple observations and its effectiveness is shown on the domain of hardware circuits, but not on software yet. Our fault-model introduced in Chapter 4 allows us to use *HSD* directly as a black-box in order to perform *SFL* with multiple observations, immediately leveraging all advantages of the algorithm.

In the following, we extend the definition of the diagnosis problem from Eq. (4.3), as well as the definition of a diagnosis from Eq. (4.4) for multiple observations $\{OBS_1, \dots, OBS_M\}$ step by step, similarly as in [IMWM19]. The goal is to also describe a diagnosis for multiple observations in a single equation. Each individual observation OBS_i is inconsistent with the system description SD when all components are declared healthy. In Eq. (5.1) we simply add index i to OBS compared to the original Eq. (4.3).

$$SD \wedge OBS_i \wedge \bigwedge_{c \in COMPS} h(c) \models \bot \tag{5.1}$$

A diagnosis Δ can have different variable assignments in SD when it comes to different observations. Hence, the system description SD (in our case given by $TF_M(P)$) is copied for each observation so that only the healthy variables $h(c)$ are shared, but all other variables are replicated. The replication can be achieved by simply adding an index i to all variables in SD. The resulting equation for a diagnosis Δ with a single observation OBS_i is described in Eq. (5.2) where only index i is added to both SD and OBS compared to the original definition of a diagnosis Eq. (4.4).

$$SD_i \wedge OBS_i \wedge \bigwedge_{c \in \Delta} \neg h(c) \wedge \bigwedge_{c \in COMPS \setminus \Delta} h(c) \not\models \bot \tag{5.2}$$

The replication of *SD* finally allows us to describe a **diagnosis Δ with multiple observations** in a single equation - Eq. (5.3). Informally, a diagnosis Δ has to make the system *SD* consistent with **all** observations $\{OBS_1, \ldots, OBS_m\}$, when all components $c \in \Delta$ are declared unhealthy. The task of *MBD* with multiple observations is to find and list such diagnoses Δ with minimal cardinality [IMWM19]. Applied to our domain of *SFL* we list cardinality-minimal sets of repair-locations where each set makes a faulty program *P* consistent with **all** given observations (e.g. test-cases).

$$\bigwedge_{1 \leq i \leq m} (SD_i \wedge OBS_i) \wedge \bigwedge_{c \in \Delta} \neg h(c) \wedge \bigwedge_{c \in COMPS \setminus \Delta} h(c) \not\models \bot \tag{5.3}$$

5.1 Multiple observations in *SFL*

Existing model-based *SFL* tools process only single observations and then use different methods to combine the results. BugAssist joins each individual result set simply by union in combination with a ranking based on how often a solution appears. The ranking can be problematic since it is strongly depending on the observations. *SNIPER* processes each observation individually followed by the algorithm *DC* which computes pairwise unions of the results. This approach can still lead to diagnoses which are not minimal as shown in [IMWM19]. In [BFP19] - which is based on the Griesmayer method [GSB06, GSB10] - the individual observation results are conjoined by intersection. However, we show in the following that both principle combination methods - union and intersection - are disadvantageous compared to using an *MBD* algorithm like *HSD* which processes multiple observations directly and only computes minimal diagnoses.

Intersecting results in fact becomes futile whenever multiple faults are present on different paths in a program. Thus, implementations which use this method only work with the assumption of a single fault in the input program - which is also the case in [BFP19]. However, this assumption obviously cannot be made in real-world applications. In Example 5.1.1 we show that the intersection approach can even lead to empty solutions, since the result sets might be disjoint. Furthermore, the example shows that multiple faults on different paths cannot even be detected if only a single observation would be used.

Example 5.1.1. Recall the program from Listing 3.1 which has two inputs a and b and calculates $max(a, b) + 1$ in the variable $result$. A modified version is given in Listing 5.1 with two introduced bugs in line 4 and 6, removing the addition of value 1 in the expression on each right-hand-side. The corresponding model in $TF_M(P)$ is given in Eq. (5.4). Consider the two failing observations $OBS_1 = \{a = 1, b = 2\}$ and $OBS_2 = \{a = 2, b = 1\}$. Suppose we use an *MBD* solver which processes only single observations. Then the optimal solution is $\{c_4\}$ with OBS_1 and $\{c_6\}$ with OBS_2. Since these two solutions are disjoint, the intersection would yield an empty set, whereas the union correctly describes both introduced bugs. On the contrary, suppose we use an *MBD* solver which processes multiple observations. Then the optimal solution when processing both OBS_1 and OBS_2 is $\{c_4, c_6\}$ right away.

```
int maxPlusOne(int a, int b) {
  int result;
  if (a < b)
    result = b; // Bug: missing "+ 1"
  else
    result = a; // Bug: missing "+ 1"
  ASSERT(a,b < result);
  return result;
}
```

Listing 5.1: Example: Multiple bugs in maxPlusOne

On the other hand, we can at least ensure that no potential repair-location is missing when combining results with the union approach, since by construction no repair-location from any individual observation is dropped. However, taking the union of individual results leads to non-minimal solutions, i.e., the precision of the results suffers. In Example 5.1.2 we show that calculating the naive union causes that a set of all 3 possible repair-locations is obtained, whereas the minimal diagnosis - which can be obtained by an *MBD* algorithm with multiple observations like *HSD* - consists only of the correct single repair-location. This also shows that the intuitive guess - that more observations can enable more precise results - is correct.

$$
\begin{aligned}
&F_3 = (\mathbf{h}(\mathbf{c_3}) \implies (g_1 = a_1 < b_1)) \wedge (g_1 \implies result_3 = result_1) \wedge (\neg g_1 \implies result_3 = result_2)\\
&F_4 = \mathbf{h}(\mathbf{c_4}) \implies (result_1 = b_1)\\
&F_6 = \mathbf{h}(\mathbf{c_6}) \implies (result_2 = a_1)\\
&F_7 = (a_1 < result_3 \wedge b_1 < result_3)\\
&TF_M(P) = F_3 \wedge F_4 \wedge F_6 \wedge F_7
\end{aligned}
\tag{5.4}
$$

Example 5.1.2. Another modified version of the program from Listing 3.1 is given in Listing 5.2 with an introduced bug in the condition in line 3, exchanging the less-than with a greater-than operator. The corresponding model in $TF_M(P)$ is similar as Eq. (5.4) only with the comparison operator in F_3 exchanged and the addition of value 1 added in F_4 and F_6. Consider the two failing observations $OBS_1 = \{a = 1, b = 2\}$ and $OBS_2 = \{a = 2, b = 1\}$. Suppose we use an *MBD* solver which processes only single observations. Then the optimal solutions are $\{\{c_3\}, \{c_6\}\}$ with OBS_1 and $\{\{c_3\}, \{c_4\}\}$

with OBS_2. Taking the union of all single solutions from both observations would result in all possible repair locations $\{c_3, c_4, c_6\}$. On the contrary, suppose we use an *MBD* solver which processes multiple observations. Then the optimal solution when processing both OBS_1 and OBS_2 is $\{c_3\}$.

```
int maxPlusOne(int a, int b) {
  int result;
  if (a > b) // Bug: wrong operator
    result = b+1;
  else
    result = a+1;
  ASSERT(a,b < result);
  return result;
}
```

Listing 5.2: Example: Precision maxPlusOne

In Example 5.1.2 we used a naive union over all repair-locations. Note that also the pairwise-union approach from [LN14, LN16] computes non-minimal diagnoses and can have exponential runtimes, as shown in [IMWM19]. To sum up, we argued that combining results from individually processed observations through existing methods like (pairwise) union and intersection is disadvantageous compared to an *MBD* algorithm which calculates minimal diagnoses with multiple observations. As already mentioned in Section 2.2, this is achieved in the algorithm *HSD* from [IMWM19]. It uses an "Implicit Hitting Set Dualization" approach to generate minimal diagnoses. The paper also demonstrates significant runtime improvements compared to the combination algorithm from [LN14, LN16] which is demonstrated on hardware circuits with injected faults - the *ISCAS85* benchmark suite.

CHAPTER 6

Implementation details

An overview of our fault localization setup is illustrated in Figure 6.1. As already mentioned we directly deploy the *HSD* algorithm from [IMWM19] as a black-box to perform fault localization together with the fault-model introduced in Chapter 4. A *C++* implementation of *HSD* is available as open-source software [hsd19]. It accepts *MBD* instances as described in Eq. (5.1) in an encoding based on the *WCNF* format for *MaxSAT* [wcn19], which is similar to the well-known *DIMACS CNF* format often used by SAT-Solvers. Such instances consist of **(1)** a list of observations, **(2)** a list of healthy variables (components) and **(3)** the system description with already introduced healthy variable implications.

We implemented a command-line option `--wcnf` for *CBMC* in order to generate such instances instead of the default behavior of *CBMC*. It accepts *ANSI-C* programs as input where observations can be specified in *ANSI-C* syntax with *CBMC* style directives as demonstrated in Listing 6.1. The input program gets transformed into our fault-model $TF_M(P)$ from Chapter 4 which directly serves as the system description with healthy variables **(3)**.

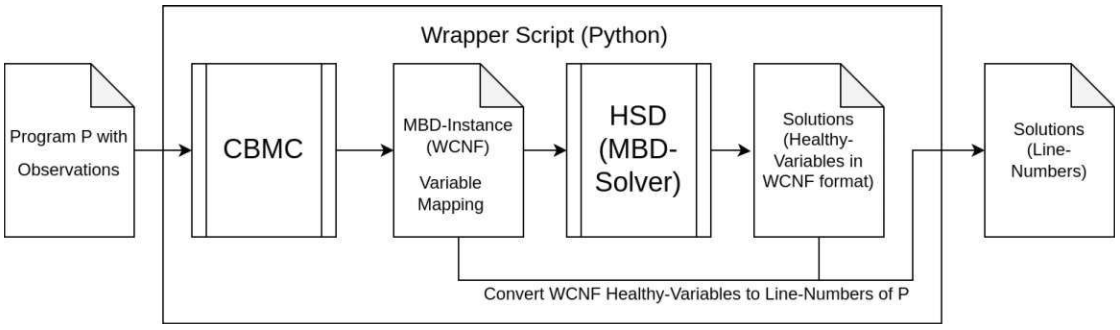
Wrapper Script (Python)
Program P with Observations
CBMC
MBD-Instance (WCNF) Variable Mapping
HSD (MBD-Solver)
Solutions (Healthy-Variables in WCNF format)
Solutions (Line-Numbers)
Convert WCNF Healthy-Variables to Line-Numbers of P

Figure 6.1: Implementation Overview

```
__CPROVER_observation_begin();
a=3; b=4; expectedOut=5; // Observation 1
__CPROVER_observation_end();
__CPROVER_observation_begin();
a=-2; b=0; expectedOut=1; // Observation 2
__CPROVER_observation_end();
__CPROVER_assert(functionInTest(a,b) == expectedOut, "error");
```

Listing 6.1: Observation specification

During the transformation, the list of healthy variables is recorded **(2)** including a mapping to line numbers in the program. Moreover, the observations **(1)** are parsed and mapped to the corresponding variables in the target encoding. The generated *MBD* instance is written to a `.wcnf` file using the standard *CBMC* argument `--outfile` which is in turn used as input for the *MBD* solver.

A wrapper script handles the execution of both *CBMC* and *HSD*. It automatically maps reported solutions - consisting of sets of unhealthy components (healthy variable identifiers) - back to line numbers in the original program. The wrapper script is implemented in *Python* and accepts programs with observations (see Listing 6.1) as input and its output directly contains the line numbers corresponding to fault locations. Since the transformation of a program to an *MBD* instance is completely separated from the actual *MBD* solver, it would also be possible to use other *MBD* algorithms in future works, if the *WCNF* format is accepted as input.

6.1 *MBD* result processing

The *MBD* solver *HSD* may deliver multiple result sets of different cardinality. For example, assume we have a faulty program with 10 lines. Then *HSD* might report multiple solution sets which can fix the bug, for example: $\{\{3,5\},\{5,7\},\{1,2,4\},\{1,4,8\}\}$. Our tool uses only the sets with minimal cardinality since we are interested in the optimal solution and in listing as few line numbers as possible. However, we take the union of all minimal-cardinality solution sets, such that no optimal solution is discarded. In the example, our tool would therefore report the lines $\{3,5,7\}$.

6.2 Fault-Model options (CBMC)

As already mentioned, we implemented the command-line option `--wcnf` in order to get *CBMC* into our fault-modeling mode. Since the *CBMC* code is available on *GitHub* [cbm24b], we created a fork of the repository which is publicly available [cbm24a]. Our code modifications and additions are encapsulated such that (1) the default behavior of *CBMC* is unchanged when the `--wcnf` option is not used and (2) new releases of the base code can be merged effortlessly with our implementation. In Chapter 4 we discussed various design decisions and options for the fault-model. In the following we summarize which choices are fixed in our implementation with *CBMC* and which can be chosen by

Decision	Implemented	Description	Preferred Choice
Dec. 1	Dec. 1.b	Havoc RHS *Pointer Assertions*	Y
Dec. 2	Dec. 2.b	Havoc LHS *Pointer Assertions*	Y
Dec. 3	Dec. 3.a	De-reference function not modified	N
Dec. 4	Dec. 4.b	Havoc RHS *Bound Assertions*	Y
Dec. 5	Dec. 5.b	Havoc LHS *Bound Assertions*	Y
Dec. 6	Dec. 6.a	Index Case-Split not modified	N

Table 6.1: Implemented Design Decisions

the user. Table 6.1 contains the implemented design decisions for *Pointers* and *Arrays*. Due to the implementation effort, we did not implement the preferred choice for Decs. 3 and 6. This is left as an improvement for future works.

Additionally to the design decisions which are fixed in the implementation, there are command-line arguments which affect the fault-model and can be chosen by the user. Generally, all standard command-line arguments of *CBMC* which can be used in combination with the standard `--dimacs` option can also be used along with our `--wcnf` option. Most notably, the effect of using or omitting the following two options has been discussed theoretically in Sections 4.2.4 and 4.2.5 and practically tested and evaluated in Chapter 7.

- `--pointer-check`
- `--bounds-check`

Further command-line arguments available in *CBMC* include - among others - options for generating various types of assertions, e.g., `--memory-leak-check` or `--div-by-zero-check`. Note that we neither use any of these options in our test runs, nor have we theoretically evaluated their effect on the fault-model in detail. This would also be a topic for future works. However, we expect similar effects as for the assertions which we already analyzed. A sketch for arguing about the `--div-by-zero-check` option is given in Example 6.2.1. Similarly to what we showed for *Pointer Assertions* and *Bound Assertions* in Example 4.2.2 and Example 4.2.7, respectively, we expect that for the other assertion types we could also construct examples which would be unsolvable if the assertions are not havoced in the fault-model. Hence, our implementation generally havocs all such *CBMC* assertions if (1) they are generated at all (using the according arguments) and (2) the according line is declared faulty. Nevertheless, we do not address the effect on the fault-model for all types of available assertions in detail.

```c
int a;
for(int i=3;i>=0;i--) {
  a = 5 / i;
}
```

Listing 6.2: Example: Division by zero

Example 6.2.1. Consider the program from Listing 6.2 which causes a division by zero in line 3 at the last iteration of the for-loop. The behavior of a division by zero is undefined in *ANSI-C* and a Floating Point Exception can occur when running the program. If the `div-by-zero-check` argument is omitted when using *CBMC* the error is not detected, since the according assertions are not generated. When performing fault localization, the resulting *MBD* instance would then also be immediately consistent (since there are no assertions) and no fault location reported.

6.3 Library function handling

In general, standard library functions (e.g., functions processing Strings) are not modeled in *CBMC*. Instead, possible return values, as well as possible call-by-reference variables (which are written to) simply become non-deterministic in the model. Obviously, this behavior can strongly limit fault localization, whenever the results of such library functions are used in a programs logic. In theory it would be possible to implement precise models for such functions. However, besides the significant implementation effort, the size of generated model instances can potentially become too large to be handled by a solver. Limitations are discussed in detail in Section 7.4. Still, we can implement models of library functions with low complexity for specific cases, namely when their usage is constrained in a program, as we will show for a specific example in the following.

In the benchmark program *schedule* - which is part of the Siemens set of benchmarks for fault localization [DER05] - the inputs for the program are provided via command line arguments (`argv`), as well as via the default input `stdin` during runtime, whereas outputs are written to the default output `stdout`. The benchmark suite contains 2650 unit tests, each consisting of (1) the command line arguments (`argv`) and (2) a file which contains the input string to `stdin`. Additionally, the correct result strings (`stdout`) can be obtained by running each test case on the original program where no faults are introduced.

The program uses the *ANSI-C* standard library (`<stdio.h>`) functions `fscanf` and `fprintf` to interact with `stdin` and `stdout`, respectively. As mentioned before, the actual behavior of such library functions is not modeled in *CBMC*. Instead, the return value, as well as the call-by-reference variables (which are written to) become non-deterministic in the model. This makes fault localization for the *schedule* program impossible, since the observations are parsed via these functions and therefore any observation would immediately be replaced by a non-deterministic value. Hence, we implemented replacement functions (essentially defining simple models) for `fscanf` and `fprintf`, as used in the *schedule* program, in order to enable specifying the given values of `stdin` and `stdout` for each test case with our introduced observation syntax (see Listing 6.1).

Since all calls of `fprintf` in the *schedule* program only write integers - except for early returns of the program (see lines 3 and 4 in Listing 6.3) - these can simply be replaced by a check (`assert`) against an array (`expectResult`) of integers containing the expected results (see lines 3 and 4 in Listing 6.4). This array can then be specified for each observation (see lines 4 and 5 in Listing 6.5). Note that we define a special integer `ERROR_CODE` which can be assigned to the array when specifying an observation to also check against the mentioned early returns in a simple way in our model.

```
1  fscanf(stdin, "%d", &prio); // fscanf Variant 1 used in schedule
2  fscanf(stdin, "%f", &ratio); // fscanf Var. 2 used in schedule
3  fprintf(stdout, "%d ", cur_proc->val); // fprintf Var. 1 in schedule
4  fprintf(stdout, "ERROR MESSAGE\n"); return; // fprintf Var. 2 in schedule
```

Listing 6.3: Schedule Benchmark: fscanf/fprintf calls

```
1  wcnf_fscanf_d(&prio); // fscanf Variant 1 Replacement
2  wcnf_fscanf_f(&ratio); // fscanf Var. 2 Replacement
3  ASSERT(cur_proc->val == expectResult[resultIterator++]); //fprintf Var. 1
4  ASSERT(ERROR_CODE == expectResult[resultIterator++]); return;
```

Listing 6.4: Schedule Benchmark: fscanf/fprintf Replacements

```
1  __CPROVER_observation_begin(); // Unittest: stdin="4.2 5"
2  // Expected stdout="2 ERROR MESSAGE\n"
3  expectResult[0] = 2;
4  expectResult[1] = ERROR_CODE;
5  stdin_array[0] = 4.2;
6  stdin_state[0] = WCNF_STDIN_STATE_MID_DOT;
7  stdin_array[1] = 5;
8  stdin_state[1] = WCNF_STDIN_STATE_INT;
9  __CPROVER_observation_end();
```

Listing 6.5: Schedule Benchmark: Observation specification

The `fscanf` calls in the *schedule* program either read an integer (`%d`) or a float value (`%f`) (see lines 1 and 2 in Listing 6.3). However, the calls can occur in different sequences depending on the inputs. Still, the behavior can be simplified such that the inputs can be specified as an array of float values for each observation (see lines 5-8 in Listing 6.5). Our replacement function for an `fscanf` reading an integer is given in Listing 6.6. Introducing a state (variable `stdin_state`) enables the simplification to the float array (variable `stdin_array`) instead of implementing a full handling of a given input string (`stdin`). This state variable basically encodes different number formats in the input string corresponding to each array position. The replacement function for an `fscanf` reading a float consists of a similar case split. This means that the `stdin` strings for all unit tests (observations) must be converted beforehand to initialize both the value and state array when specifying the observations (as in Listing 6.5). However, this can be done in an automated fashion. Due to the constraint that only numbers are read in the program we only need to convert `stdin` strings until the first text character appears. In this context, a text character is anything except (1) numbers (2) dot (decimal separator) (3) number separators (blanks, new lines).

Note that we performed the necessary replacement of the functions, as well as initializing the needed helper variable (e.g. `resultIterator`) in the programs source code via a script that also inserts all observations in our observation syntax. It would also be possible to implement our replacement function for `fscanf` - as well as replacements for any library function in general - in *CBMC* to be directly used as a model for the according function. Then it would not be necessary to pre-process the source code beforehand. However, since our model of `fscanf` only works for the constrained case where only the patterns `%d` and `%f` are used, we could not include it in *CBMC* to be used as general model for `fscanf`. An interesting future work with *CBMC* could be to check the input program for constraints and conditionally apply such simple models if possible.

```
int wcnf_fscanf_d(int *intValue) {
  switch(stdin_state[stdin_index]) {
    case STDIN_STATE_INT:
      *intValue = (int)stdin_array[stdin_index];
      stdin_index++;
      return 1;
    case STDIN_STATE_RIGHT_DOT:
      *intValue = (int)stdin_array[stdin_index];
      stdin_state[stdin_index] = STDIN_STATE_DOT_ONLY;
      return 1;
    case STDIN_STATE_MID_DOT:
      *intValue = (int)stdin_array[stdin_index];
      // get decimal part only:
      stdin_array[stdin_index] = stdin_array[stdin_index] - *intValue;
      stdin_state[stdin_index] = STDIN_STATE_LEFT_DOT;
      return 1;
    case STDIN_STATE_LEFT_DOT:
    case STDIN_STATE_DOT_ONLY:
    case STDIN_STATE_TEXT:
      return 0;
    case STDIN_STATE_EOF:
      return EOF;
  }
}
```

Listing 6.6: Schedule Benchmark: fscanf replacement function

To summarize, we have shown that simplified models of library functions like `fscanf` are possible, if the usage is limited to certain patterns (here integer and float numbers). This enables handling observations consisting of input streams to a program which is necessary for the *schedule* benchmark program. A complete model of functions such as `fscanf` which processes input strings with all possible patterns directly would require significantly more implementation effort and would also result in a more complex model. Therefore, we implemented the simplified version instead. Note that we also discovered our model-based fault localization approach is not able to cope with the general complexity of the *schedule* program as described in Section 7.4.1, even with this simplified handling of the `fscanf` function. Thus, we can hypothesize that the complexity of a full modeling of `fscanf` would exceed the limitations of model-based fault localization anyway. These general limitations are discussed in detail in Section 7.4.

CHAPTER 7

Experiments and Results

This chapter presents our experimental results as well as theoretical findings. First of all, common evaluation metrics are elaborated in Section 7.1. Then, the experimental results from our tool with hand-crafted examples and popular benchmarks are analyzed in detail in Sections 7.2 and 7.3. The limitations of our tool and model-based *SFL* in general are addressed in Section 7.4. Finally, we present a theoretical finding concerning successful observations in Section 7.5.

7.1 Metrics

The following two aspects are primarily considered when evaluating *SFL* implementations.

- Runtime
- Result quality
  - Is the actual fault location contained in the result set? (= Hit?)
  - Code Size Reduction (or similar scores as described in [WGL+16])

7.1.1 Runtime

When it comes to model-based fault localization, we can consider the runtimes of (1) the model creation and (2) the diagnosis algorithm separately. When only single observations are processed, then depending on the implementation, (1) might be required for every run (every observation), or it might be enough to create the model once and then only run (2) for each observation. Since most existing approaches process single observations, they usually give average runtimes for individual runs.

A comparison of runtime between our approach and related work is difficult since (1) we process multiple observations and (2) we have a more complex model (e.g. containing more assertions). Both (1) and (2) can improve result quality as we demonstrated both theoretically and practically on our hand-crafted examples summarized in Section 7.2, but come at the cost of a higher runtime. Also, differences in hardware and operating system lead to differences in runtimes. Therefore, we evaluate and compare runtimes for different options within our implementation (e.g.: Multiple vs. Single observation runs, with/without *Bound Assertions*, etc.) rather than against other approaches.

7.1.2 Result Quality

Fault localization tools report sets of program locations which are candidates for containing the actual bug(s). The most important and trivial evaluation is of course whether the actual bug location(s) are contained in the result set. If it is the case, we denote it as a "*Hit*". It might even be considered as a flaw in the fault-model if there are instances where an actual bug location is not found.

Assuming that the result sets are correct, i.e., contain the actual bug locations, the survey [WGL+16] states: *"...the effectiveness of a software fault localization technique is defined as the percentage of code that needs to be examined before the first faulty location for a given bug is identified."* While there are multiple types of scores described in the study, most model-based fault localization techniques stick to Code Size Reduction (CSR). It is defined as the ratio of reported fault locations to the total number of lines in the program [LN14] as described in Eq. (7.1).

$$CSR = \frac{|ResultSet|}{\#TotalProgramLines} \tag{7.1}$$

As mentioned in [WGL+16], only lines containing executable statements are counted in some fault localization works, which of course causes much higher CSR values, especially if a program contains many blank or comment lines. However, in the model-based software fault localization literature which is most related to our approach, all lines are counted to get the total number of lines, regardless of the lines content [JM10, LN16, LN14]. Therefore, we also stick to this type of evaluation.

7.2 Hand-crafted Examples

In Chapter 4 we already argued about advantages of our fault-model with hand-crafted examples. We ran our tool on all the introduced examples to verify the qualitative results which we derived theoretically - based on the formulas $TF_M(P)$ - also experimentally. All example programs are summarized in Table 7.1. The column "Program" contains the references to the listings in the previous chapters and the column "Experiment result" describes the main outcome of the runs with our tool. Note that "Dec." refers to the design decisions for pointers and arrays introduced in Sections 4.2.4 and 4.2.5.

Program	Experiment result
Listing 4.4*	(*added assertion as in Example 4.2.6) Line 7 is a solution with Dec. 3.a
Listing 4.5	Unsolvable with Dec. 1.a, correct location with Dec. 1.b
Listing 4.6	Unsolvable with Dec. 2.a, correct location with Dec. 2.b
Listing 4.7	Verified behavior described in Example 4.2.4
Listing 4.8	Consistent without *Pointer Assertions* - no solution reported
Listing 4.9	Verified generated $TF_M(P)$
Listing 4.10	Unsolvable with Dec. 4.a, correct location with Dec. 4.b
Listing 4.11	Unsolvable with Dec. 5.a, correct location with Dec. 5.b
Listing 4.12	*Bound Assertions* only generated for line 2
Listing 4.10*	Verified behavior described in Example 4.2.9 (*added assertion)
Listing 4.13	Line 3 is a solution with Dec. 6.a
Listing 5.1	Verified solutions with $\{OBS_1\}, \{OBS_2\}, \{OBS_1, OBS_2\}$ from Example 5.1.1
Listing 5.2	Verified solutions with $\{OBS_1\}, \{OBS_2\}, \{OBS_1, OBS_2\}$ from Example 5.1.2

Table 7.1: Experiments on hand-crafted examples

7.3 *TCAS*

The most popular benchmark program for *SFL* is *TCAS* from the Siemens set of benchmarks [HFGO94, DER05] with 98 papers referencing it up to 2016 [WGL$^+$16, Table 6]. It consists of 173 lines of code in total and does not contain any loops or pointers, but a small array, multiple *If-Then-Else* statements and various assignments with logical expressions. Note that with our fault-model, there are only 34 lines in *TCAS* that can be declared faulty. The remaining 139 lines are blank lines, comments, lines with only opening/closing brackets, variable declarations, function definitions, defines or includes. The purpose of the program is to calculate altitude separation for aircrafts based on multiple input values. The benchmark suite contains the original correct version of the program, as well as 41 modified versions where different bugs are introduced in each version. Additionally, it contains 1608 unit tests each consisting of a value assignment for the 12 input arguments. We ran all unit tests on the original version in order to generate the expected output for each test, which serves as a specification and completes the observations. Note that the last observations in the benchmark do not specify all 12 arguments, resulting in an immediate exit of the program in all version. Thus, we skipped such observations and only used the first 1578 unit tests which have the correct number of arguments. In [LN14] only 1578 tests are mentioned in the first place and we assume other approaches also skipped the incomplete unit tests.

We conducted experimental runs on *TCAS* using various combinations of observations and options of our tool. All experiments were carried out on an Intel Core i5-11600K 3.90GHz processor, 16GB DDR4 RAM and the operating system Ubuntu 22.04. First, we analyze a special feature of *TCAS* concerning array-bound violations in Section 7.3.1 which is required to explain certain results afterwards. Then we present tables containing the detailed results of each experiment and individual *TCAS* version in Section 7.3.2. Finally, aggregated values for precision and runtime are summarized in Section 7.3.3 and compared against relevant literature.

7.3.1 *TCAS* Array-Bound Violations

The benchmark does not have any pointers, but it does contain a small constant array of size 4. We analyze the array-associated code parts in detail, since it is already enough to significantly influence results depending on the array fault-model. Listing 7.1 contains the relevant lines from the original *TCAS* program where no bug is introduced. In the following, we refer to the line numbers of the listing, which do not correspond to the real line numbers of the whole program. Line 1 shows the declaration with a constant size of 4, making indices $\{0, 1, 2, 3\}$ valid. It is then completely initialized with constant values in lines 3 - 6. The array is only accessed by a read in the `ALIM` function - here at line 11 - at the index given by the variable `Alt_Layer_Value`. There are 4 calls of the `ALIM` function in *TCAS* which may be avoided by control flow. However, the value of `Alt_Layer_Value` is set directly to a program input beforehand (see line 8), without any checks or modification in between. Thus, the given test cases (observations) can directly cause array-bound violations.

In fact, multiple of the benchmarks test cases have invalid indices. Moreover, two versions of the program (v33 and v38) have introduced bugs exactly on lines containing array operations. In v33, the four lines initializing the array (lines 3-6 in Listing 7.1) have indices $1-4$ instead of $0-3$, respectively. Therefore, the last index is invalid, index 0 stays uninitialized and index 1-3 get wrong values assigned compared to the correct version. In v38, the array is declared with size 3 instead of 4 (see line 1 in Listing 7.1) which essentially makes the initialization at index 3 in line 6, as well as the read access in line 11 faulty for all observations which have `Alt_Layer_Value` > 2. Since the declaration itself cannot be a bug location in our fault-model, we treated the two lines 6 and 11 in Listing 7.1 as the correct bug locations for v38 when evaluating our results. These insights are needed to explain certain results in the following Section 7.3.2.

```
1   int Positive_RA_Alt_Thresh[4]; // Array declaration
2   ....
3   Positive_RA_Alt_Thresh[0] = 400; // Array initialization
4   Positive_RA_Alt_Thresh[1] = 500; // Writing values
5   Positive_RA_Alt_Thresh[2] = 640;
6   Positive_RA_Alt_Thresh[3] = 740;
7   ....
8   Alt_Layer_Value = atoi(argv[7]); // index variable assignment from input
9   .....
10  int ALIM () { // Reading from array:
11    return Positive_RA_Alt_Thresh[Alt_Layer_Value]; }
```

Listing 7.1: *TCAS* Benchmark: Array Access

7.3.2 Detailed *TCAS* results

The following tables, Tables 7.3 to 7.5, list our experimental results for each *TCAS* version. They share the same column descriptions, where "Ver." identifies the *TCAS* program version. "#Err" is the count of bugs in the program. "#ObsF" gives the number of failing observations, which was gathered by running each version with each observation. An observation is failing for a version, if its output differs from the output of the original version. In all tables we compare runs with multiple observations (top column "All fail. obs.") against single observation runs (top column "Single obs. runs"). "#Loc" specifies the number of reported lines, i.e., the size of the solution set. "HIT" (Y/N) describes whether the solution set contains the actual bug location, or if there are multiple bugs, if the set of actual bugs overlaps with the reported solution set. "#LocU" is the number of reported lines, i.e., the size of the solution set when combining the results of single observation runs by union. "#Miss" is the number of single observation runs where the actual bug location is not contained in the reported solution set. "#Miss (Filt.)" is the same as "#Miss", except that runs where the solver did not find any solution are not counted (filtered). "IsubSet" (Intersection Sub Set) describes whether the solution set of the run with multiple observations (set A) is a subset of the intersection of the result sets of the single observation runs (set B). Since an *MBD* solver should deliver optimal solutions, this should be the case for all versions which only have a single faulty line and serves as a sanity check. The meaning of the "IsubSet" column entries are described in Table 7.2, they were chosen this way so that different entries are clearly visible when looking at the table.

Note that some single observations are failing not because of the actual introduced bug, but due to an array-bound violation of the input as described in detail in Section 7.3.1. As already elaborated theoretically in Example 4.2.9, such cases can cause the solver to report no solutions, since the model is immediately consistent when *Bound Assertion*s are omitted. When computing the intersection for the column "IsubSet", we discarded such single observations which have no solution since this would make the overall intersection empty. This is basically also an example that shows that intersecting results is not safe as soon as there are multiple bugs in the program.

Also note that when it comes to multiple observation runs, we performed experiments both with only the failing observations, as well as with all observations (failed and successful). However, the experiments showed that the results are the same, but come at the cost of longer runtime when adding also successful observations. This result led us to argue that adding successful observations in fact cannot improve result quality in Section 7.5. Therefore, the detailed tables only contain results from failing observations, while we also present the aggregated values of other experiments afterwards in Section 7.3.3.

Without *Bound Assertions* - Table 7.3

In Table 7.3 we compare the results from runs with the set of all failing observations against the union of the result sets when running each failing observation individually, both without *Bound Assertion*s. Since the handling of arrays in the fault-model is not

Table entry	empty	y, $<$	y, eq.	N, $\supset$	N
Meaning	empty set	$A \subset B$	$A = B$	$A \supset B$	no subset relation

Table 7.2: "IsubSet" column explanation

TCAS			All fail. obs.		Single obs. runs				#Miss (liter.)	
Ver.	#Err	#ObsF	#Loc	HIT	#LocU	#Miss	#Miss (Filt.)	IsubSet	BugAss.	SNIPER
v1	1	131	16	Y	22	0	0	y, eq.	0	0
v2	1	67	4	Y	27	0	0	y, eq.	0	0
v3	1	23	10	Y	27	0	0	y, eq.	10	0
v4	1	23	16	Y	22	3	0	y, eq.	0	1
v5	1	10	8	Y	27	0	0	y, eq.	0	0
v6	1	12	15	Y	21	0	0	y, eq.	0	0
v7	1	36	4	Y	23	0	0	y, eq.	0	0
v8	1	1	18	Y	18	0	0	y, eq.	0	0
v9	1	7	10	Y	10	0	0	y, eq.	0	0
v10	2	14	13	Y	24	0	0	y, eq.	0	0
v11	2	14	5	Y	21	0	0	y, eq.	0	0
v12	1	70	8	Y	28	0	0	y, eq.	22	0
v13	1	4	9	Y	27	0	0	y, eq.	0	0
v14	1	50	4	Y	4	0	0	y, eq.	0	0
v15	3	10	8	Y	27	0	0	y, eq.	0	0
v16	1	70	16	Y	18	0	0	y, eq.	0	0
v17	1	35	4	Y	23	0	0	y, eq.	0	0
v18	1	29	4	Y	23	0	0	y, eq.	0	0
v19	1	19	4	Y	23	0	0	y, eq.	0	0
v20	1	18	16	Y	21	0	0	y, eq.	0	0
v21	1	16	15	Y	21	0	0	y, eq.	0	0
v22	1	11	8	Y	8	0	0	y, eq.	0	0
v23	1	42	9	Y	9	0	0	y, eq.	1	0
v24	1	7	15	Y	19	0	0	y, eq.	0	0
v25	1	4	8	Y	10	1	0	y, eq.	0	0
v26	1	11	9	Y	28	0	0	y, eq.	0	0
v27	1	10	8	Y	27	0	0	y, eq.	0	0
v28	1	76	2	Y	31	0	0	y, eq.	18	0
v29	1	18	3	Y	26	0	0	y, eq.	4	0
v30	1	58	4	Y	27	0	0	y, eq.	0	0
v31	3	14	15	Y	17	0	0	y, eq.	0	0
v32	3	2	15	Y	16	0	0	y, eq.	0	0
v33	4	89	4	N	25	46	46	y, eq.	-	-
v34	1	77	8	Y	26	0	0	y, eq.	0	0
v35	1	76	2	Y	31	0	0	y, eq.	18	0
v36	1	123	2	Y	3	0	0	y, eq.	0	0
v37	1	95	5	Y	23	0	0	y, eq.	0	0
v38	2	76	0	N	0	76	0	empty	-	-
v39	1	4	8	Y	10	1	0	y, eq.	0	0
v40	2	123	10	Y	15	0	0	y, eq.	0	0
v41	1	23	16	Y	22	3	0	y, eq.	0	1

Table 7.3: TCAS experiments without BA

addressed in any related work, we assume that other approaches do not use *Bound Assertions* in their fault-model. Therefore, these results are the most comparable, even though we already argued about the problems when discarding *Bound Assertions* in Example 4.2.9. The column "#Miss (liter.)" displays the number of misses from the most related works, BugAssist [JM10] and *SNIPER* [LN14]. Still, a comparison between these values is difficult, since the number of failing observations ("#ObsF") is already slightly inconsistent for some versions between our results, [JM10] and [LN14]. Test cases which are left out in one paper but are used in another might potentially be a miss. We have no explanation for this discrepancy as the setup for retrieving these values is the same in all works, namely running each program version with each test case and comparing the result to the original version.

What is clearly visible in the table is that the number of locations from the union of single observation runs "#LocU" is significantly larger than the number of locations from the single run with multiple observations, "#Loc", for almost all version. This confirms what we have already shown in Example 5.1.2 - namely that processing multiple observations with an *MBD* algorithm delivers more precise results compared to processing single observations only.

The multiple observation runs all contain the exact bug location (column "HIT": "Y"), except for versions 33 and 38. In v38 the array `Positive_RA_Alt_Thresh` is incorrectly declared, as described in Section 7.3.1, namely by size 3 instead of 4. Since we omitted *Bound Assertions* in this experiment, the *MBD* solver can assign any value to array accesses with invalid indices, which makes v38 consistent with all observations and the result set empty. This again shows the problems that potentially arise when discarding *Bound Assertions*. In v33 the actual bug locations are the 4 lines initializing the same array, see also Section 7.3.1. The *MBD* solver calculates 4 optimal solution sets in this case, each containing only a single location, therefore the column "#Loc" shows 4 in Table 7.3 which comes from the union of the optimal result sets that our tool reports, as elaborated in Section 6.1. One is the only line reading from the array (see Listing 7.1) and the other three are locations which appear after the array access in the control flow.

In related work [LN14, JM10] the versions 33 and 38 are simply omitted from their results without any explanation. Also in [BFP19] - which generally works only for single-fault programs - version 38 is skipped without explanation even though it has only a single fault. We assume this is the case, since all other approaches do not consider the fault-model of arrays.

Another interesting result is visible in the column "#Miss". With a correct fault-model we expected every optimal solution to contain the actual bug location, i.e., "#Miss" should always be zero, at least for all versions that have only a single bug (#Err = 1) and except for the versions v33 and v38 as just described. However, this is not the case for some versions, e.g., v4 or v25. Analyzing these cases reveals that all observations leading to a miss contain a direct array-bound violation as described in Section 7.3.1. In all these cases the solver reported that the system is consistent and no results, since again any value can be assigned when an invalid index is read from an array when *Bound*

*Assertion*s are discarded. In order to confirm this we added the column "#Miss (Filt.)" where we filtered out the single observations where the solver reported "consistent" and it is indeed zero for all versions with "#Err" = 1 as can be seen in the table.

Also in related work misses are reported, which is visible in the last two columns of Table 7.3 for [JM10] and [LN14]. However, this issue is not addresses in any of these papers. Since array handling is generally not addressed in any related work, we can guess that their misses might also be caused by test cases with invalid array indices. In fact, the line reading from the array (line 11, Listing 7.1) is an additional fault location, as the used index is unchecked. The bug becomes active as soon as an observation with an invalid array index is encountered. This again underlines the importance of the array fault-model, as well as that the single fault-assumption is insufficient.

The column "IsubSet" shows that the evaluation with multiple observations is at least as good as the intersection of the single observation run results for all version. This experimental result confirms that the *MBD* solver indeed delivers optimal results considering all observations. While we already showed that intersection is only safe for a single-fault assumption, in this case it was possible to build the intersection for all versions, including those with multiple errors. However, as already mentioned, we had to discard empty result sets when building the intersection, which is again an argument supporting that the intersection approach is not safe in the general case.

To summarize the key takeaways from Table 7.3, we have shown that results are more precise with a multiple observation processing *MBD* solver compared to the union of single observation runs. Moreover, results with a multiple observations processing *MBD* solver are even at least as precise as the intersection of single observation runs (when the intersection is possible). Further, we explained the results of *TCAS* versions v33 and v38, as well as the encountered misses, which are not addressed in any literature, and confirmed zero misses with filtered observations.

With *Bound Assertions* - Table 7.4

The results for all *TCAS* versions with the intended configuration of our tool - namely with *Bound Assertion*s enabled - are listed in Table 7.4. We compare these results only against our own results with other configurations (Tables 7.3 and 7.5) because the array fault-model is not addressed in related word.

Again, the number of locations from the union of single observation runs "#LocU" is significantly larger than the number of locations from the single run with multiple observations "#Loc" for almost all version. This confirms that processing multiple observations with an *MBD* algorithm delivers more precise results compared to processing single observations only, also with *Bound Assertion*s enabled.

The column "HIT" has some entries with "N" (e.g.: v25) that have been "Y" without *Bound Assertions* (see Table 7.3), i.e., some result sets from the runs with all failing observations do not contain the actual fault location. This is explainable by the array-bound violations caused by several observations for almost all version. Since we use

TCAS			All fail. obs.		Single obs. runs		
Ver.	#Err	#ObsF	#Loc	HIT	#LocU	#Miss	IsubSet
v1	1	131	16	Y	22	0	y, eq.
v2	1	67	4	Y	27	0	y, eq.
v3	1	23	10	Y	27	0	y, eq.
v4	1	23	5	Y	22	0	y, eq.
v5	1	10	8	Y	27	0	y, eq.
v6	1	12	15	Y	21	0	y, eq.
v7	1	36	4	Y	23	0	y, eq.
v8	1	1	18	Y	18	0	y, eq.
v9	1	7	10	Y	10	0	y, eq.
v10	2	14	13	Y	24	0	y, eq.
v11	2	14	5	Y	21	0	y, eq.
v12	1	70	8	Y	28	0	y, eq.
v13	1	4	9	Y	27	0	y, eq.
v14	1	50	4	Y	4	0	y, eq.
v15	3	10	8	Y	27	0	y, eq.
v16	1	70	16	Y	18	0	y, eq.
v17	1	35	4	Y	23	0	y, eq.
v18	1	29	4	Y	23	0	y, eq.
v19	1	19	4	Y	23	0	y, eq.
v20	1	18	16	Y	21	0	y, eq.
v21	1	16	15	Y	21	0	y, eq.
v22	1	11	8	Y	8	0	y, eq.
v23	1	42	18	Y	18	0	N, $\supset$
v24	1	7	15	Y	19	0	y, eq.
v25	1	4	2	N	10	1	y, eq.
v26	1	11	9	Y	28	0	y, eq.
v27	1	10	8	Y	27	0	y, eq.
v28	1	76	1	N	31	1	y, eq.
v29	1	18	3	Y	26	0	y, eq.
v30	1	58	2	N	27	1	y, eq.
v31	3	14	15	Y	17	0	y, eq.
v32	3	2	15	Y	16	0	y, eq.
v33	4	89	5	Y	26	0	y, eq.
v34	1	77	8	Y	26	0	y, eq.
v35	1	76	1	N	31	1	y, eq.
v36	1	123	9	Y	11	0	N, $\supset$
v37	1	95	5	Y	23	0	y, eq.
v38	2	76	2	Y	9	0	y, eq.
v39	1	4	2	N	10	1	y, eq.
v40	2	123	2	N	10	3	y, eq.
v41	1	23	4	Y	22	0	y, eq.

Table 7.4: TCAS experiments with BA

Bound Assertions, line 11 in Listing 7.1 is a new viable fault location in addition to the actual locations in every version. Thus, each version technically has at least 2 fault locations now. When it comes to multiple bugs, it can happen that the solver finds optimal solutions where all bugs can be prevented by havocing a single line, analogous to what happened previously for v33 in Table 7.3. However, if we count the invalid array access (line 11) also as a valid fault location, then "HIT" would in fact always be "Y" in Table 7.4, i.e., the actual fault location is always found. An improvement compared to Table 7.3 is that the correct fault location is now found also for the two array-specific versions v33 and v38 ("HIT"="Y"), which is a clear advantage that comes from our array fault-model. For v38 the *MBD* solver even returned just a single optimal result set of size 2 which exactly contains the actual fault locations (lines 6 and 11 in Listing 7.1).

Also here, some of the single observation runs did not contain the actual fault location, see column "#Miss", e.g., v25. Analyzing these cases again reveals that all observations leading to a miss contain a direct array-bound violation as described in Section 7.3.1 and the line with the array-bound violation is reported instead of the intended location from *TCAS*. The column "#Miss" would therefore always be zero if we counted the invalid array access as a valid fault location.

The column "IsubSet" shows that the evaluation with multiple observations is at least as good as the intersection of the single observation run results for all version, except for versions v23 and v36. Analyzing these two versions reveals that the intersection discards the location with the invalid array access (line 11 in Listing 7.1), while the runs processing all failing observations include it. Therefore, it is another example that intersecting results is not safe when it comes to multiple bugs in a program, since the invalid array access is technically a correct bug location.

With *Bound Assertions* and filtered observations - Table 7.5

Our explanation for the previously encountered issues concerning the "#Miss" and "HIT" columns were all related to the observations which directly contain array-bound violations (`Alt_Layer_Value` > 3) since they essentially trigger a bug which is not the actual bug introduced by each *TCAS* version. In order to verify that this indeed explains all such issues, we ran our tool again with the same configuration as for Table 7.4, except that we filtered out all observations that contain a direct array-bound violation beforehand for both the runs with all failing observations, as well as the single observation runs. In other words, we got rid of the additional array-bound violating fault locations by keeping only observations that do not trigger this additional bug. The results with only the filtered observations are displayed in Table 7.5.

The "HIT" column shows that the actual bug location is found for all *TCAS* versions with the multiple observation runs. The "#Miss" column is always zero which means that the actual bug locations are contained in the result sets of all the single observation runs. Also the "IsubSet" column shows that the results of the multiple observation runs are now always a subset of the intersection of single observation run result sets, which underlines our explanation why this was not always the case in Table 7.4. As expected,

TCAS			All fail. obs.		Single obs. runs		
Ver.	#Err	#ObsF	#Loc	HIT	#LocU	#Miss	IsubSet
v1	1	131	16	Y	22	0	y, eq.
v2	1	67	4	Y	27	0	y, eq.
v3	1	23	10	Y	27	0	y, eq.
v4	1	23	16	Y	22	0	y, eq.
v5	1	10	8	Y	27	0	y, eq.
v6	1	12	15	Y	21	0	y, eq.
v7	1	36	4	Y	23	0	y, eq.
v8	1	1	18	Y	18	0	y, eq.
v9	1	7	10	Y	10	0	y, eq.
v10	2	14	13	Y	24	0	y, eq.
v11	2	14	5	Y	21	0	y, eq.
v12	1	70	8	Y	28	0	y, eq.
v13	1	4	9	Y	27	0	y, eq.
v14	1	50	4	Y	4	0	y, eq.
v15	3	10	8	Y	27	0	y, eq.
v16	1	70	16	Y	18	0	y, eq.
v17	1	35	4	Y	23	0	y, eq.
v18	1	29	4	Y	23	0	y, eq.
v19	1	19	4	Y	23	0	y, eq.
v20	1	18	16	Y	21	0	y, eq.
v21	1	16	15	Y	21	0	y, eq.
v22	1	11	8	Y	8	0	y, eq.
v23	1	42	9	Y	18	0	y, eq.
v24	1	7	15	Y	19	0	y, eq.
v25	1	4	8	Y	10	0	y, eq.
v26	1	11	9	Y	28	0	y, eq.
v27	1	10	8	Y	27	0	y, eq.
v28	1	76	2	Y	31	0	y, eq.
v29	1	18	3	Y	26	0	y, eq.
v30	1	58	4	Y	27	0	y, eq.
v31	3	14	15	Y	17	0	y, eq.
v32	3	2	15	Y	16	0	y, eq.
v33	4	89	5	Y	26	0	y, eq.
v34	1	77	8	Y	26	0	y, eq.
v35	1	76	2	Y	31	0	y, eq.
v36	1	123	2	Y	11	0	y, eq.
v37	1	95	5	Y	23	0	y, eq.
v38	2	76	2	Y	9	0	y, eq.
v39	1	4	8	Y	10	0	y, eq.
v40	2	123	10	Y	10	0	y, eq.
v41	1	23	16	Y	22	0	y, eq.

Table 7.5: TCAS experiments with BA, only valid array-index obs.

the multiple observation runs are still much more precise than the union of the single observation runs for almost all versions ("#Loc" < "#LocU"). In conclusion, we get perfect results by eliminating the additional array-bound violation bugs which are not intended by the *TCAS* benchmark.

7.3.3 Summarized *TCAS* results

Runtime

In Table 7.6 we present aggregated runtime values over all versions. The column "Evaluation" displays the used configuration, where "All obs." means that simply all observations (test cases) available in the *TCAS* benchmark were used, i.e., failing and successful. "Failing obs." means that only failing observations are used and "BA" is an abbreviation for *Bound Assertions*. Note that Nr. 2 ("Failing obs.") corresponds to the detailed results in Table 7.3, Nr. 4 ("with BA") to Table 7.4 and Nr. 6 ("only valid-bound-inputs") to Table 7.5. The columns *CBMC* and *HSD* describe the times consumed by the respective tool, i.e., the time for building the model (trace formula) and the time of the *MBD* solver (*HSD*), respectively, whereas the column "SUM" simply contains the sum of both - the total runtime. The values are given in seconds.

The displayed values are calculated by the arithmetic mean over the individual *TCAS* version results. For the single observation runs we calculated the sum of each individual run to get the total runtime for each version, since each run is executed sequentially. For aggregating over the *TCAS* versions for the single observation runs the arithmetic mean is used as well (with the sum of single observation runs for each version as basis).

While runtime results are not available for *SNIPER* [LN14], BugAssist [JM10] gives the average runtime of a single observation run within each *TCAS* version. We calculated the aggregated time for BugAssist by multiplying each average runtime by the number of failed observations (see "#ObsF" in Table 7.5) and then taking the arithmetic mean over these totals of each version. Since BugAssist processes only single observations and

Nr.	Evaluation	CBMC [s]	HSD [s]	SUM [s]
1	All obs.	0.896	1.895	2.791
2	Failing obs.	0.053	1.349	1.401
3	All obs. with BA	0.897	101.082	101.979
4	Failing obs. with BA	0.053	10.937	10.990
5	All obs. with BA, only valid-bound-inputs	0.892	8.732	9.624
6	Failing obs. with BA, only valid-bound-inputs	0.053	1.614	1.667
7	Sum of single observation runs	1.442	0.442	1.884
8	Sum of single observation runs with BA	1.574	0.501	2.076
9	BugAssist (from paper)	-	-	2.747

Table 7.6: TCAS Average Runtimes over all versions

does not implement *Bound Assertions*, the value is probably most comparably to our result Nr. 7, "Sum of single observation runs", where our value is lower by $0,907$ seconds or faster by $\sim 33\%$. However, the significance of this value is questionable, since we use different hardware for our experiments, as well as another fault-model.

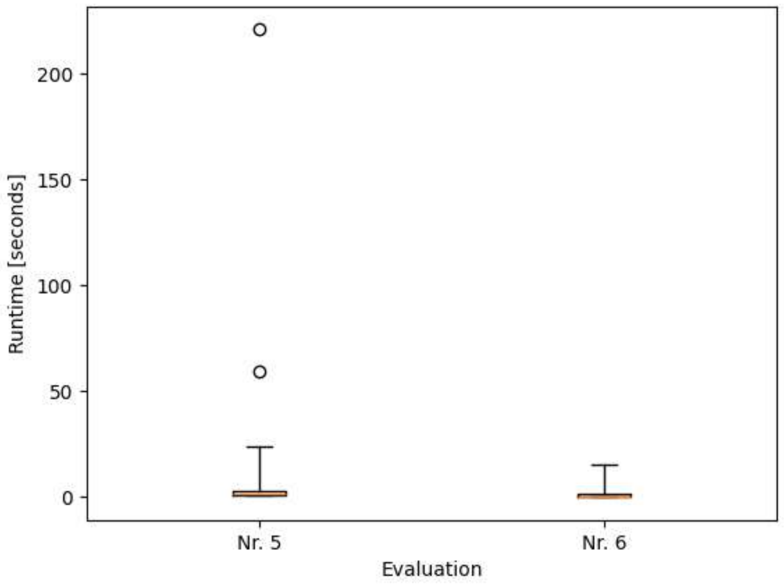

Figure 7.1: Boxplot: Version results

What is notable is the comparison between runs where all observations are used against runs where only failing observations are used, i.e., Nrs. (1 vs. 2), (3 vs. 4) and (5 vs. 6). Adding the successful observations always causes a significantly larger runtime, although it does not improve result quality, as we will show in Section 7.5. The factor of runtime increase by adding successful observations amounts to ~ 2.0 for the runs without *Bound Assertions* (Nr. 1 and 2), ~ 9.2 with *Bound Assertions* (Nr. 3 and 4) and ~ 5.4 with *Bound Assertions* and filtered observations (Nr. 5 and 6). This shows that adding assertions can significantly increase complexity of the *MBD* problem. Especially without filtered observations (Nr. 3 and 4), the factor is notably large since some of the added observations actually introduce conflicts with the *Bound Assertions*, i.e., they are additional failing observations with our fault-model.

Intuitively this should not be the case with filtered observations (only valid-bound-inputs, Nr. 5 and 6). However, the difference is still quite high with the factor ~ 5.4. Analyzing the results in detail shows that this is solely caused by the runtimes of v33 (~ 60 sec) and v38 (~ 221 sec) for evaluation Nr. 5 which corresponds to the two visible outliers in the boxplot Figure 7.1. These two outliers are caused by the array-bound violations with all observations having `Alt_Layer_Value` > 2 of v33 and v38 as explained in

Nr.	Evaluation	CSR
1	All obs.	5.2%
2	Failing obs.	5.2%
3	All obs. with BA	2.6%
4	Failing obs. with BA	4.7%
5	All obs. with BA, only valid-bound-inputs	5.1%
6	Failing obs. with BA, only valid-bound-inputs	5.1%
7	Union of single observation runs	12.0%
8	Union of single observation runs with BA	12.0%
9	SNIPER (from paper)	11.0%
10	BugAssist (from paper)	8.0%

Table 7.7: TCAS Average CSR over all versions

Section 7.3.1. Therefore, we again have additional failing observations. Calculating the factor of runtime increase when discarding these two versions yields a similar value as for without *Bound Assertions* (Nrs. 1 and 2), namely $Nr5/Nr6 = 2.92/1.52 =\sim 1.9$.

Result Quality - CSR

In Table 7.7 we present the aggregated Code Size Reduction (*CSR*) over all *TCAS* versions. The column "Evaluation" again displays the used configuration and the line Nrs. 1 to 8 have the same meaning as previously for the runtime evaluation in Table 7.6. Additionally, the results from *SNIPER* [LN14] and BugAssist [JM10] are shown in line Nrs. 9 and 10 which are copied directly from the respective papers. The *CSR* is calculated as described in Section 7.1.2 for each run, i.e., the number of reported code lines divided by the total number of code lines which amount to 173 in *TCAS*. The individual result sets of single observation runs (Nrs. 7 and 8) within each *TCAS* version are combined via union and the size of the resulting sets is used to calculate the *CSR* in this case. For aggregating over all *TCAS* versions, the arithmetic mean over each versions *CSR* is used. Note that with our fault-model there are 34 valid fault locations in *TCAS*. Thus, the maximum possible *CSR* (all possible locations declared faulty) yields $\sim 19.7\%$ and the minimum (one fault location) $\sim 0.6\%$.

As expected, the results from multiple observation evaluations (Nrs. 1-6) are significantly better than the union of single observation runs (Nrs. 7 and 8). The *CSR* from the union of single observation runs without *Bound Assertions* (12%) is the highest in the table and closest to *SNIPER* [LN14] with 11% which makes sense since they combine results with a pairwise union approach. The second closest is BugAssist [JM10] with 8%. Note that the comparison of *CSR* values with related work has to be interpreted carefully since both BugAssist and *SNIPER* do not give the number of viable fault locations in *TCAS* within their fault-model. Furthermore, the method of combining the results from individual runs within a version is unclear for BugAssist, but we assume a union approach is used.

The results from our multiple observation runs (Nrs. 1-6) are significantly stronger compared to related work (Nrs. 9 and 10), even Nrs. 1 and 2, which have the largest CSR among our multiple observation evaluations but have the most comparable fault-model to related work since no *Bound Assertions* are used. This highlights also experimentally the benefit of using an *MBD* solver that processes multiple observations.

As we will show in Section 7.5, adding successful observations cannot improve result quality. This is experimentally confirmed as the CSR values are equal in line Nrs. 1 and 2, as well as 5 and 6. However, for the evaluation with *Bound Assertions* and unfiltered observations (Nrs. 3 and 4) the CSR is significantly smaller (2.6%) when adding all observations compared to just failing observations (4.7%). This is again explained by the fact that failing observations are added in this case, since a lot of *TCAS* test cases contain array-bound violations. Analyzing the results in detail shows that solely the line where the array is read (see line 11 in Listing 7.1) is a valid optimal solution for many *TCAS* versions. When adding all observations this line then becomes the only optimal solution since it can fix both the actual error as well as the array-bound violation. This explains the surprisingly low CSR of 2.6% in this case. For example, evaluating *TCAS* v18 with only failing observations (Nr. 4) yields the result sets $\{\{52\}, \{58\}, \{127\}, \{141\}, \{91, 104\}, \ldots\}$. Since our tool reports the union of optimal result sets, the final solution size is 4. Note that line 58 in *TCAS* corresponds to line 11 in Listing 7.1. Adding all observations in this case yields the result sets $\{\{58\}, \{118, 141\}, \ldots\}$, thus the final solution size is 1.

7.4 Limitations

The scalability of model-based *SFL* is strongly limited since the size of the model can grow exponentially based on the complexity of the input program, e.g., when it comes to multiple levels of nested loops. We describe the limitations of our approach based on the example *schedule* in the following Section 7.4.1. Note that all existing model-based *SFL* methods for *ANSI-C* are only evaluated on *TCAS* or similarly small programs [WGL$^+$16]. If results for more complex programs are given, they are only achieved by using other fault localization methods additionally.

For example, BugAssist [JM10] also presents results from larger programs of the Siemens benchmarks [DER05] including *schedule*. However, these programs are too complex for their model-based approach and they only state the following concerning how they obtained their results: "..we combine our technique with existing trace reduction techniques like program slicing, concolic execution, and isolating failure-inducing input using delta debugging.". Without any details on these additional techniques the significance of such results concerning model-based *SFL* is highly questionable.

In general, we conclude that the strength of model-based *SFL* lies in obtaining correct and precise results for rather small programs or parts of programs, while it is not suitable for processing large instances. For example, [BFP19] compare the results from their model-based fault localization tool against spectrum-based techniques and confirm a much higher precision on small instances.

7.4.1 Schedule

The second most popular program (according to [WGL$^+$16]) among the Siemens benchmark programs [DER05] is called *schedule*. A major difficulty for this program is that the observations consist not only as a list of command line arguments, but also of a string which is piped to `stdin` during runtime. In the program various calls of the function `fscanf` read and parse this input. We already described our observation handling for *schedule* (`stdin`) in Section 6.3. However, even with our simplified model of the `fscanf` function, the instances become too large to be handled by our technique. In fact, the creation of the model is already infeasible, even with a low loop-unwind depth of 4. This applies also to the standard *CBMC* behavior - without our modifications for fault localization - when disabling propagation (option `"--no-propagation"`).

Propagation is usually used by *CBMC* to significantly decrease the size of the trace formula $TF(P)$. For example, RHS expressions containing variable reads are replaced by the actual value, if the value is unique at the current point in the program. Expressions which do not contain variables anymore can then be immediately resolved which can even lead to discarding paths in the control flow when the value of *Guard Conditions* can be derived through the propagation. For fault localization, propagation cannot be used, since we must be able to havoc any variable assignment. Havocing a variable would invalidate any previously propagated value and therefore invalidate the trace formula.

Schedule has multiple levels of nested loops leading to exponential growth of the trace formula when unrolling every iteration. Also, the *Guard Conditions* become larger for each step with every additional iteration that is unrolled. The instances become so big that even with the mentioned unwinding depth of 4, *CBMC* got stuck when the 16GB memory of our machine was fully consumed.

7.5 Successful Observations

When a program contains a bug, it is usually only triggered by a subset of test cases (observations). Thus, we can split the set of all observations OBS_A into a set of failed observations OBS_F and a set of successful observations OBS_S where Eqs. (7.2) and (7.3) hold.

$$OBS_S \cap OBS_F = \emptyset \tag{7.2}$$

$$OBS_S \cup OBS_F = OBS_A \tag{7.3}$$

The definition of the diagnosis problem for multiple observations Eq. (5.1) only considers observations which make the system inconsistent, i.e., only OBS_F. However, the question arises if adding successful observations OBS_S can improve result quality, i.e., lead to a smaller result set.

Proposition 7.5.1. Adding successful observations to a diagnosis problem cannot influence the diagnosis Δ (=result set), i.e., cannot lead to a different diagnosis.

In the following, we show informally that Proposition 7.5.1 holds. Our argument is based on the trivial fact that an observation can only influence a diagnosis Δ if it causes an inconsistency. For the sake of contradiction, suppose a solution Δ_1 (set of unhealthy components) that restores consistency for a set OBS_F causes an inconsistency with some successful observation $OBS_i \in OBS_S$. This would mean that Eq. (7.4) holds.

$$\bigwedge_{c \in COMPS} (h(c) \implies f_c) \wedge OBS_i \wedge \bigwedge_{c \in \Delta_1} \neg h(c) \wedge \bigwedge_{c \in COMPS \setminus \Delta_1} h(c) \models \bot \tag{7.4}$$

However, since OBS_i is a successful observation it must be consistent with all components declared healthy Eq. (7.5).

$$\bigwedge_{c \in COMPS} (h(c) \implies f_c) \wedge OBS_i \wedge \bigwedge_{c \in COMPS} h(c) \not\models \bot \tag{7.5}$$

If Eq. (7.5) holds then Eq. (7.4) trivially cannot hold, a contradiction.

In other words, we only havoc components of SD, i.e., SD becomes only weaker, thus observations which are already consistent with all components are also consistent if any component is havoced. However, note that in the recently published *MBD* algorithm [ZOTZ23] successful observations are used in order to increase efficiency (runtime) of the algorithm, according to the paper. Since no implementation was available we did not confirm or measure the improvement.

CHAPTER 8

Conclusion

Existing *SFL* methods for *ANSI-C* have limitations when it comes to the fault-model as well as processing multiple observations efficiently. We evaluated the effects of different design decisions concerning the model for each *ANSI-C* instruction type. With these insights a novel fault-model that also considers complex instructions like pointers and arrays was defined. We implemented our fault-model as a feature of the open-source model-checker *CBMC*. Creating a standard instance of the *MBD* problem allows us to deploy the recent *MBD* algorithm *HSD* which processes multiply observations effectively as opposed to other currently available implementations of model-based *SFL* tools.

We demonstrated the advantages in terms of result sets due to the new fault-model on hand-crafted examples in theory and confirmed them with our implementation also experimentally. The enhanced fault-model - especially for arrays - allowed us to derive meaningful results on all versions of the *TCAS* benchmarks while other existing tools skip the versions containing array faults.

Furthermore, we argued about the benefits and necessity of processing multiple observations on hand-crafted examples, especially concerning the size of result sets. The evaluation of our full implementation on the *TCAS* benchmark showed significant improvements of the *CSR*, compared against other approaches, as well as against combination of result sets of individual observations within our fault-model.

We proposed that adding successful observations cannot cause smaller result sets and confirmed it informally. Based on the larger benchmark program *schedule* we elaborated on the limitations of model-based *SFL* algorithms.

Future work could be devoted to implementing and evaluating further improvements in the fault-model, e.g., implementing Decs. 3.b and 6.b (see Sections 4.2.4 and 4.2.5) or conditionally simplified models for functions similar to `fscanf` as explained in Section 6.3. In addition, newer *MBD* algorithms which claim improvements to *HSD* (e.g.: [ZOTZ23]) could be deployed and evaluated.

Bibliography

[ACG$^+$05] Liliana Ardissono, Luca Console, Anna Goy, Giovanna Petrone, Claudia Picardi, Marino Segnan, and Daniele Theseider Dupré. Cooperative model-based diagnosis of web services. 2005.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In Rance Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[BFP19] Geoff Birch, Bernd Fischer, and Michael Poppleton. Fast test suite-driven model-based fault localisation with application to pinpointing defects in student programs. *Softw. Syst. Model.*, 18(1):445–471, 2019.

[cbm24a] Repository: CBMC Fork with WCNF feature. `https://github.com/LukasGraussam/cbmc`, 2024. Accessed: 2024-04-05.

[cbm24b] Repository: CBMC. `https://github.com/diffblue/cbmc`, 2024. Accessed: 2024-04-05.

[CFD93] Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In Ruzena Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28 - September 3, 1993*, pages 1494–1501. Morgan Kaufmann, 1993.

[CK20] Dean Cazes and Meir Kalech. Model-based diagnosis with uncertain observations. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 2766–2773. AAAI Press, 2020.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[CKY00] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. *Technical Report CMU-CS-03-126, Carnegie Mellon University, School of Computer Science*, 1 2000.

[CKY03] Edmund M. Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and verilog programs using bounded model checking. In *Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003*, pages 368–371. ACM, 2003.

[DER05] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empir. Softw. Eng.*, 10(4):405–435, 2005.

[dKW87] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artif. Intell.*, 32(1):97–130, 1987.

[Gra24] Lukas Graussam. Consistency-based Software Fault Localization with Multiple Observations. `https://doi.org/10.5281/zenodo.13895727`, October 2024.

[Gro04] Alex Groce. Error explanation with distance metrics. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2004.

[GSB06] Andreas Griesmayer, Stefan Staber, and Roderick Bloem. Automated fault localization for C programs. In Roderick Bloem, Marco Roveri, and Fabio Somenzi, editors, *Proceedings of the Workshop on Verification and Debugging, V&D@FLoC 2006, Seattle, WA, USA, August 21, 2006*, volume 174 of *Electronic Notes in Theoretical Computer Science*, pages 95–111. Elsevier, 2006.

[GSB10] Andreas Griesmayer, Stefan Staber, and Roderick Bloem. Fault localization using a model checker. *Softw. Test. Verification Reliab.*, 20(2):149–173, 2010.

[GZ17] Weiwei Gong and Xu Zhou. A survey of SAT solver. *AIP Conference Proceedings*, 1836(1):020059, 06 2017.

[HFGO94] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *Proceedings of 16th International Conference on Software Engineering*, pages 191–200, 1994.

[HKKT18] Timo Hynninen, Jussi Kasurinen, Antti Knutas, and Ossi Taipale. Software testing: Survey of the industry practices. In Karolj Skala, Marko Koricic, Tihana Galinac Grbac, Marina Cicin-Sain, Vlado Sruk, Slobodan Ribaric, Stjepan Gros, Boris Vrdoljak, Mladen Mauher, Edvard Tijan, Predrag Pale, and Matej Janjic, editors, *41st International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2018, Opatija, Croatia, May 21-25, 2018*, pages 1449–1454. IEEE, 2018.

[hsd19] Repository: Model-Based Diagnosis with Multiple Observations (HSD Implementation). `https://github.com/alexeyignatiev/mbd-mobs`, 2019. Accessed: 2024-03-30.

[IMM17] Alexey Ignatiev, António Morgado, and João Marques-Silva. Model based diagnosis of multiple observations with implicit hitting sets. *CoRR*, abs/1707.01972, 2017.

[IMWM19] Alexey Ignatiev, António Morgado, Georg Weissenbacher, and João Marques-Silva. Model-based diagnosis with multiple observations. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 1108–1115. ijcai.org, 2019.

[JM10] Manu Jose and Rupak Majumdar. Cause clue clauses: Error localization using maximum satisfiability. *CoRR*, abs/1011.1589, 2010.

[JS16] Dietmar Jannach and Thomas Schmitz. Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach. *Autom. Softw. Eng.*, 23(1):105–144, 2016.

[KSL21] Meir Kalech, Roni Stern, and Ester Lazebnik. Minimal cardinality diagnosis in problems with multiple observations. *Diagnostics (Basel)*, 11(5), April 2021.

[LN14] Si-Mohamed Lamraoui and Shin Nakajima. A formula-based approach for automatic fault localization of imperative programs. In Stephan Merz and Jun Pang, editors, *Formal Methods and Software Engineering - 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings*, volume 8829 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2014.

[LN16] Si-Mohamed Lamraoui and Shin Nakajima. A formula-based approach for automatic fault localization of multi-fault programs. *J. Inf. Process.*, 24(1):88–98, 2016.

[MSKC14] Amit Metodi, Roni Stern, Meir Kalech, and Michael Codish. A novel sat-based approach to model based diagnosis. *J. Artif. Intell. Res.*, 51:377–411, 2014.

[NPQW13] Iulia Nica, Ingo Pill, Thomas Quaritsch, and Franz Wotawa. The route to success - A performance comparison of diagnosis algorithms. In Francesca Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 1039–1045. IJCAI/AAAI, 2013.

[OJM24] Pedro Orvalho, Mikolás Janota, and Vasco Manquinho. Cfaults: Model-based diagnosis for fault localization in C programs with multiple test cases. *CoRR*, abs/2407.09337, 2024.

[Rei87] Raymond Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987.

[SHCvH07] Stefan Schlobach, Zhisheng Huang, Ronald Cornet, and Frank van Harmelen. Debugging incoherent terminologies. *J. Autom. Reason.*, 39(3):317–349, 2007.

[SKFP12] Roni Tzvi Stern, Meir Kalech, Alexander Feldman, and Gregory M. Provan. Exploring the duality in conflict-directed model-based diagnosis. In Jörg Hoffmann and Bart Selman, editors, *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*, pages 828–834. AAAI Press, 2012.

[SMV$^+$07] Sean Safarpour, Hratch Mangassarian, Andreas G. Veneris, Mark H. Liffiton, and Karem A. Sakallah. Improved design debugging using maximum satisfiability. In *Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007, Proceedings*, pages 13–19. IEEE Computer Society, 2007.

[SSW03] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Interactive type debugging in haskell. In Johan Jeuring, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2003, Uppsala, Sweden, August 28, 2003*, pages 72–83. ACM, 2003.

[TCJ08] Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson. Finding minimal unsatisfiable cores of declarative specifications. In Jorge Cuéllar, T. S. E. Maibaum, and Kaisa Sere, editors, *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30,* *2008, Proceedings*, volume 5014 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2008.

[wcn19] WCNF Format - from the MaxSAT Evaluation 2019 website. `https://maxsat-evaluations.github.io/2019/rules.html#input`, 2019. Accessed: 2024-03-03.

[WGL$^+$16] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Trans. Software Eng.*, 42(8):707–740, 2016.

[WSM02] Franz Wotawa, Markus Stumptner, and Wolfgang Mayer. Model-based debugging or how to diagnose programs automatically. In Tim Hendtlass and Moonis Ali, editors, *Developments in Applied Artificial Intelligence, 15th International Conference on Industrial and Engineering, Applications of Artificial Intelligence and Expert Systems, IEA/AIE 2002, Cairns, Australia, June 17-20, 2002, Proceedings*, volume 2358 of *Lecture Notes in Computer Science*, pages 746–757. Springer, 2002.

[ZLAA19] Abubakar Zakari, Sai Peck Lee, Khubaib Amjad Alam, and Rodina Ahmad. Software fault localisation: a systematic mapping study. *IET Softw.*, 13(1):60–74, 2019.

[ZOTZ23] Huisi Zhou, Dantong Ouyang, Xinliang Tian, and Liming Zhang. Diagdo: an efficient model based diagnosis approach with multiple observations. *Frontiers Comput. Sci.*, 17(6):176407, 2023.

[ZOZT22] Huisi Zhou, Dantong Ouyang, Liming Zhang, and Naiyu Tian. Model-based diagnosis with improved implicit hitting set dualization. *Appl. Intell.*, 52(2):2111–2118, 2022.

[ZOZZ22] Huisi Zhou, Dantong Ouyang, Xiangfu Zhao, and Liming Zhang. Two compacted models for efficient model-based diagnosis. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*, pages 3885–3893. AAAI Press, 2022.