UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1998

On Solving String Constraints

PHI DIEP BUI

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2021

ISSN 1651-6214
ISBN 978-91-513-1098-5
urn:nbn:se:uu:diva-428900

Dissertation presented at Uppsala University to be publicly examined in 2446, ITC, Polacksbacken (Lägerhyddsvägen 2), Uppsala, Monday, 1 February 2021 at 09:15 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor Matthew Hague (Department of Computer Science, Royal Holloway-University of London).

Abstract

Bui, P. D. 2021. On Solving String Constraints. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 1998. 54 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-513-1098-5.

Software systems are deeply involved in diverse human activities as everyone uses a variety of software systems on a daily basis. It is essential to guarantee that software systems all work correctly. Two popular methods for finding failures of software systems are testing and model checking. Various efficient testing and model checking approaches are satisfiability-based, where the core of the approaches is Satisfiability Modulo Theories (SMT) solvers for solving the path feasibility and/or reachability problems. The significant growth of string manipulating programs in modern programming languages, including Python and JavaScript, demands SMT solvers being capable of analysing string constraints. This thesis proposes two frameworks for checking the satisfiability of extensive classes of string constraints, discovers a new decidable fragment of string constraints, and introduces efficient solvers for solving string constraints.

The first framework for checking the satisfiability of string constraints is based on Counter-Example Guided Abstract Refinement (Cegar) procedure, and applicable to diverse classes of string constraints. It is worth mentioning that the framework is the first one ever that can support both context-free membership and transducer constraints. The framework has two components: under-approximation and over-approximation. The under-approximation uses flat automata to restrict the search for a solution to only strings generated by a flat automaton. The over-approximation abstracts the input constraints and produces a counter-example of the abstraction. In the second framework for checking the satisfiability string constraints, the under-approximation uses parametric flat automata to restrict the domain of variables, thus allows better performance. Furthermore, the second framework is capable of solving string-number conversion constraints. It is a crucial characteristic since string-number conversion is a part of the definition of core semantics in numerous program languages such as Python and JavaScript.

The thesis introduces a new decidable fragment of string constraints, called weakly-chaining. This fragment pushes the borders of decidability of string constraints by generalising the existing straight-line as well as the acyclic fragment of the string logic. The new decidable fragment is empirically useful as it helps string solvers guarantee termination in many more cases since the solvers do not provide any guarantee of termination to handle string constraints in general.

The thesis also presents three efficient solvers for solving string constraints, called Trau, Trau +, and Z3-Trau. Trau uses the first framework presented above and is capable of solving a large class of constraints including transducer and context-free grammar. Trau+ is a later version of Trau and implemented the decision procedure of the weakly-chaining fragment in the over-approximation. Z3-Trau follows the second framework above and uses parametric flat automata for under-approximating the domain of variables. These three string solvers are evaluated on not only existing but also newly generated benchmarks. Evaluation results show that the solvers significantly outperform other state-of-the-art string solvers.

Keywords: String constraint solving, SMT, Verification

Phi Diep Bui, Department of Information Technology, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-513-1098-5
urn:nbn:se:uu:diva-428900 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-428900)

Dedicated to my parents and my wife

List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I **Flatten and Conquer: A Framework for Efficient Analysis of String Constraints.** Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukáš Holík, Ahmed Rezine, and Philipp Rümmer. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), 2017. [3]

II TRAU**: SMT solver for string constraints**. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukáš Holík, Ahmed Rezine, and Philipp Rümmer. In *Formal Methods in Computer-Aided Design* (FMCAD), 2018. [4]

III **Chain Free String Constraints**. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bui Phi Diep, Petr Janků, and Lukáš Holík. In *International Symposium on Automated Technology for Verification and Analysis* (ATVA), 2019. [8]
Best paper award.

IV **Efficient Handling of String-Number Conversion**. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Julian Dolby, Hsin-Hung Lin, Petr Janků, Lukáš Holík, and Wei-Cheng Wu. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), 2020. [2]

I am the main author of papers I, II, and IV, and the co-main author of paper III. The ideas originated and were developed in discussions with other authors. I am the sole implementor of I, II, and IV, and the co-main implementor of paper III. I wrote the papers together with other authors. Reprints were made with permission from the publishers.

Other publications

The following papers were not included in this thesis.

V **Counter-Example Guided Program Verification.** Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Bui Phi Diep. In *Formal Methods* (FM), 2016. [7]

VI **Distributed Bounded Model Checking**. Prantik Chatterjee, Subhajit Roy, Bui Phi Diep, and Akash Lal. In *Formal Methods in Computer-Aided Design* (FMCAD), 2020. [26]

Acknowledgement

This will be a long list of people I would like to say “Thank you".

First and foremost, I want to send my special thanks to my family.

Con cảm ơn bố mẹ đã sinh ra, nuôi dưỡng, và dạy dỗ con trưởng thành. Mỗi khi con có thành công dù nhỏ hay lớn, người đầu tiên con nghĩ về luôn là bố mẹ. Mỗi khi con gặp trở ngại, hay ngay cả lúc con mất niềm tin vào chính mình, niềm tin bố mẹ dành cho con giúp con đứng lên và bước tiếp. Tình cảm, sự động viên của bố mẹ là một hành trang quan trọng con luôn mang theo trong suốt quá trình học tập. Con yêu bố mẹ rất nhiều.

Cảm ơn Dịu, em đã và đang đồng hành cùng anh trong cuộc sống. Anh tự hào vì có em để cùng chia sẻ những niềm vui, cùng nhau vượt qua những khó khăn. Sự lạc quan của em làm cho cuộc sống của anh từ khô cằn trở nên màu sắc. Vì anh, em đã từ bỏ nước Nga, bỏ tương lai ở Việt Nam để sang Thụy Điển sống cuộc sống vất vả cùng anh, và để giúp anh chuyên tâm học tập và nghiên cứu. Anh sẽ mãi không quên sự hi sinh này của em.

Cảm ơn chị Cẩm, anh Đức, Ken và Hana. Gia đình mình lúc nào cũng tuyệt vời.

I would like to express thanks of gratitude to my supervisors, Dr. Mohamed Faouzi Atig and Prof. Parosh Aziz Abdulla. Faouzi, you are the best supervisor I have had. You taught me various things about research, from forming ideas, problem-solving, to writing papers. You are always the first person I seek advice from when I struggle in doing my PhD. I could not complete the PhD study without your generous support. Parosh, I will not forget your encouragement and trust throughout the years. I still remember our lunch in Skarholmen in May 2014 when you asked me to be your PhD student. It is such a memorable, life-changing moment to me. And of course, thank you for sharing your tea interest with me. I enjoyed every moment we discussed the tea in the office. The black tea that I got from your trip to France last year tastes really good. I still keep its box and many other tea boxes as they remind me of your continuous mental support you have given me.

I would like to thank Prof. Bengt Jonsson, my third advisor, for his advice, support and also for all the valuable graduate courses he gave and which I attended with great pleasure.

I am very thankful to my co-authors Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, Petr Janků, Julian Dolby, Hsin-Hung Lin, Wei-Cheng Wu, and Akash Lal. Thank you for your encouraging comments and for the pleasant discussions we had during the work.

My thanks also go to the previous and current members of the Verification group: Aiswarya Cyriac, Lisa Kaati, Carl Leonardsson, Othmane Rezine, Amendra Shrestha, Jari Stenman, Cong Quy Trinh, Tuan Phong Ngo, Yunyun Zhu, Magnus Lång, Sarbojit Das.

I want to thank Elisabeth Lindqvist, the administrator of PhD studies in Department of Information Technology, for helping me handle multiple issues in multiples occasions in 5 years at the department.

Besides, thank Nikolaj Bjørner. We just met once in Marktoberdorf and have not had a real conversation. However, I admired your great work with Z3, and also your hard work in so many years. You are a role model for me to follow. Thank Akash Lal, Zing Wang, and Björn Terelius for mentoring me during my internships at Microsoft and Google. I gained so much working experience from the internships.

Of course, big thanks to my beloved Vietnamese friends. Swedish weather is cold, thus our warm friendship keeps me alive through the years. Cảm ơn anh Quý, em Hà, bé Khỉ, anh Minh, chị Thủy, anh Phong, chị Quỳnh Anh, bé Chi và Bông, anh Phúc, chị Trâm Anh, bé Nam Phương, anh Lâm, chị Thùy, bé Bống và Bông, anh Long, chị Hương, anh Tuấn, chị Oanh, bé Cam, em Lệ Anh, anh Thành, chị Diễm, bé Anna và Lisa, anh Long, em Thái. Cảm ơn tập thể Bách Khoa với rất nhiều thành viên, Tuấn Anh, Nam và các em intern/exchange. Chơi cùng mấy đứa lúc nào cũng vui. Anh em đừng quên smash nhau nhé. Không thể không nhắc tới tập thể Happy men, có Tuyến, Tố, Trọng, Tú. Thật tuyệt khi dù tao ở thật xa và không nhiều khi về nhà, bọn mình vẫn nói chuyện hàng ngày. Amazing, gút chóp.

Finally, a special thanks to Justin and Amanda as you help me to translate the thesis abstract to Swedish. You helped me complete the most important and difficult part of the thesis, and I have really appreciated it.

Sammanfattning på Svenska

Programvarusystem är djupt involverade i olika mänskliga aktiviteter eftersom alla använder en mängd olika programvarusystem dagligen. Ingen av mycket grundläggande enheter som en digital klocka eller praktiska innovationer inom mobiltelefoner skulle fungera utan programvara. Det är viktigt att garantera att alla programvarusystem fungerar korrekt. Två populära metoder för att hitta fel i programvarusystem är testning och modellkontroll. Olika effektiva test och modellkontrollmetoder är tillfredsställelsebaserade, där kärnan i tillvägagångssätten är Satisfiability Modul Theories (SMT) solvers för att lösa genomförbarhetsvägen och/eller nåbarhetsproblem. Den betydande tillväxten av strängmanipulerande program på moderna programmeringsspråk, inklusive Python och JavaScript, kräver att SMT-solvers kan analysera sträng begränsningar.

Doktorsavhandlingen föreslår ramar för att kontrollera tillfredsställelsen hos omfattande klasser av strängbegränsningar, upptäcker ett nytt avgörbart fragment av strängbegränsningar och introducerar effektiva strängsolvers som stöder klasserna av strängbegränsningar.

Det första ramverket för kontroll av tillfredsställande sträng begränsningar är baserat på Counter-Example Guided Abstract Refinement-proceduren (hädanefter kallad CEGAR) och kan tillämpas på olika klasser av sträng begränsningar, inklusive sammanfogning, längd, sammanhangsfritt medlemskap och givare. Vidare är det värt att nämna att ramverket är den första någonsin som kan stödja både sammanhangsfritt medlemskap och transducerbegränsningar. Ramverket har två komponenter: under-approximation och over-approximation. Under-approximationen använder platt automat för att begränsa sökningen efter en lösning till endast strängar som genereras av ett platt automat. Tillfredsställelsen av de underskattade begränsningarna löses genom att beräkna Parikhsbilden av strängbegränsningarna för att sedan använda en SMT solver för att lösa Parikhs bildbegränsningarna. Överapproximationen abstraktar inmatningsbegränsningarna och ger ett motexempel på abstraktionen. Motexemplet används sedan för att generera en parameter som indikerar former av platta automater i underskattningen. På samma sätt följer det andra ramverket för kontroll av tillfredsställande sträng begränsningar CEGAR-metoden. Den stora förändringen i det andra ramverket är att under-approximation använder parametrisk platt automat för att begränsa domänen för variabler, och således möjliggör bättre prestanda. I under-approximationens komponenter för båda de två ramarna introducerar vi algoritmer som översätter tillfredsställelseproblemet med sträng

begränsningar till satisfieringsproblemet med en linjär formel i polynomtid. Dessutom är vårt andra ramverk kapabel till att lösa strängnummerens omvandlingsbegränsningar. Det är en avgörande egenskap eftersom (i) konvertering av strängnummer är en del av definitionen av kärnsemantik på många programspråk som Python och JavaScript och (ii) de flesta av de senaste strängbegränsningslösarna ger begränsat stöd till strängnummerens omvandlingsbegränsningar.

Vidare introducerar avhandlingen ett nytt avgörbart fragment av sträng begränsningar, kallat weakly-chaining. Sträng begränsningar som ska beaktas i det weakly-chaining fragmentet inkluderar omvandlare, ordekvationer och stränglängdsbegränsningar. Detta fragment driver gränserna för decidability strängbegränsningar genom att generalisera den befintliga rätlinjen såväl som det acykliska fragmentet av stränglogiken. Detta fragment driver gränserna för avgörbarhet för strängbegränsningar genom att generalisera den befintliga rätlinjen liksom det acykliska fragmentet av stränglogiken. Det nya avgörbara fragmentet är användbart i empiriskt arbete eftersom det hjälper stränglösare att garantera avslutning i mycket fler fall, eftersom lösarna inte ger någon garanti för avslutning för att hantera sträng begränsningar i allmänhet.

Avhandlingen presenterar även tre effektiva lösare för att lösa strängbegränsningar, kallade TRAU, TRAU+ och Z3-TRAU. TRAU använder det första ramverket som presenteras ovan och är kapabel till att lösa en stor klass av begränsningar inklusive givare och kontextfri grammatik. TRAU har två viktiga optimeringar för att öka prestanda. Den första optimeringen är för hantering av givare begränsningar genom att konvertera dem till kontextfria medlemsbegränsningar med tvåvägs push down automata. Den andra optimeringen är att kombinera den flat-automatabaserad metoden med den delade-baserad metoden vid implementeringen av under-approximationen. TRAU+ är en senare version av TRAU och implementerade beslutsproceduren för det weakly-chaining fragmentet i över approximationen. Både TRAU och TRAU+ använder Z3 externt för att lösa aritmetiska begränsningar. Z3-TRAU är den senaste versionen, som är helt integrerad i Z3 SMT-lösaren. Z3-TRAU följer det andra ovannämnda ramverket och använder parametrisk flatt automata för att underskattar domänen för variabler. En annan skillnad på Z3-TRAU jämfört med TRAU och TRAU+ är stödet för strängnummer konverterings begränsningar. Alla dessa tre stränglösare utvärderas inte bara på existerande men också nyligen genererade måttstock. Alla måttstockar hämtas från praktiska exempel. Utvärderingsresultat visar att lösarna signifikant överträffar andra toppmoderna stränglösare.

Contents

1. Introduction

In this chapter, we present the background of the thesis and the motivation behind it. First, we explain the importance of controlling the quality of software systems. In order to do this, developers often use two techniques: *testing* and *model checking*. An overview of these two techniques is mentioned before we give a summary of the *satisfiability-based approach*, which is in the center of various efficient testing and model checking tools. Afterward, we focus on the satisfiability problem of *string constraints* as the analysis of string constraints is a necessary component of any Satisfiability Modulo Theories (SMT) solvers. Finally, we present our results and challenges in solving the satisfiability problem for the analysis of string constraints.

1.1 Background

Software systems are deeply involved in diverse human activities as everyone uses a variety of software systems on a daily basis. From very basic devices such as a digital watch to handy innovations in cell phones, none of these machines could function without software. Some tasks controlled by software systems are even too complex and risky for humans. For instance, manually controlling nuclear power plants, and rail-way systems are too dangerous and complicated, thus using software for automatic control is safer and more convenient. It is essential to guarantee that software systems work correctly. Any failure costs us time, labor, money, or even somebody's life.

Guaranteeing software systems working correctly is not an easy task since it is nearly impossible to find and keep track of all scenarios and executions in the systems. First, software systems usually perform a large number of diverse functions which can interact with each other in complex and subtle ways. Therefore, the number of scenarios in each software system is huge, or infinite in many cases. Second, because most software systems manipulate unbounded data structures, e.g. array or string structures, when the systems interact with users, or databases, the number of states in the systems can be also extremely large. As a consequence, there is a large demand for *efficient* methods for finding failures of software systems.

Two popular methods for finding failures of software systems, which are testing and model checking, are explored in Section 1.1.1. In Section 1.1.2 we present the satisfiability-based approach as the common ground of testing and model checking methods in finding failures.

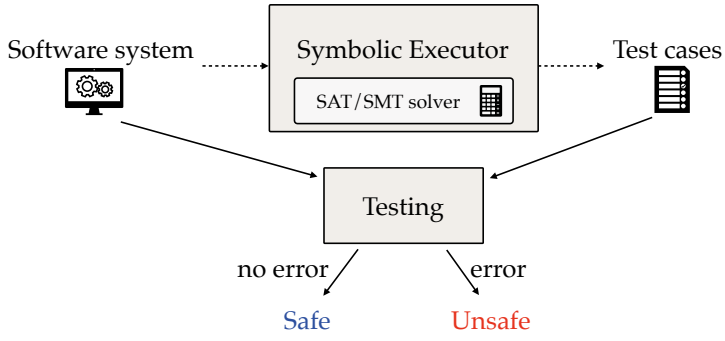

Figure 1.1. A basic form of testing

1.1.1 Testing and Model Checking

Testing

Testing [17] is a widely-used technique to detect bugs in software systems. The technique is used in many areas such as detecting errors in software front-end, back-end, network communication protocols, and hardware circuits [34]. Each phase of software development requires a different type of tests. For example, in early stages, unit testing is for small and particular functions. When software systems are large, integration and end-to-end testing are used to guarantee that sub-components of the systems can work together. Testing typically consumes at least 40%-50% of development effort of the software [74, 60]. The time dedicated to testing is more significant for systems that require higher levels of reliability. Therefore, testing is a significant part of the software development process.

Figure 1.1 presents a basic form of testing. Inputs of testing consist of a software system to be tested, and *test cases*. A test case specifies the conditions under which the software under test will run. It specifies also the expected results. Test cases can be constructed either manually or automatically by techniques such as *symbolic execution* [51]. The number of test cases used in testing varies with systems but is often large in practice due to the system complexity. Given a system and test cases, a testing phase includes two steps: executing the system with all test cases, then checking output results. If the result of a test case does not match its expected output, the software system is declared to be buggy. Otherwise, if all output results match the expected outputs, the software is declared to be safe.

The main disadvantage of testing is its incompleteness due to the difficulty of covering all behaviors/states of software systems. On one hand, the number of test cases can be significantly large or infinite. Hence, generating all the test cases that are needed to cover all the behaviors of the software is a challenging task, or even impossible. On the other hand, when a system requires user interactions, testing needs to take into account the sequence of user actions.

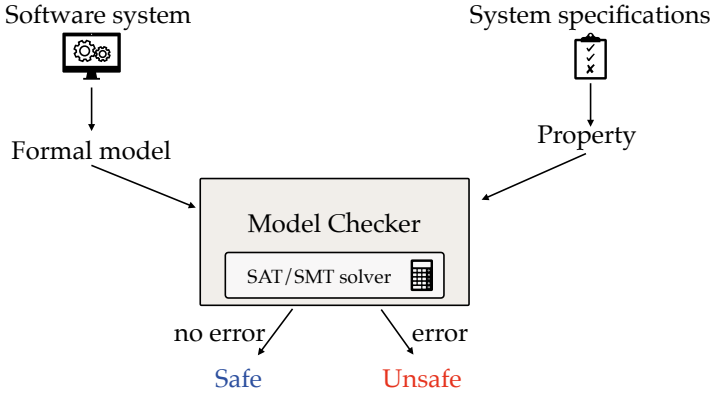

Figure 1.2. A basic form of model checking

As a result, testing can hardly explore all possible executions of a software (and this sometimes leads to "*code coverage*" issue)

Code coverage is the degree to which the source code of a program is executed when running test cases. One effective approach to have high code coverage in testing is symbolic execution [51]. Specifically, symbolic execution allows programmers to check the feasibility of a path in a program, i.e. determining the value of inputs under which the given path can be executed. For example, symbolic execution for automated test generation involves concolically running a program to collect path constraints on the inputs. Another example is symbolic execution for vulnerability detection. It combines derived path constraints with the specifications for attacks, often given by security experts. In both examples, the satisfiability of path constraints, which is solved by SAT/SMT solvers [19, 61, 68, 16, 39], indicates whether the path is feasible or not. If the path constraints are satisfiable, then the path is feasible. Likewise, if the path constraints are unsatisfiable, then the path is infeasible.

Model Checking

Unlike testing, *model checking* is exhaustive. This means that model checking can show the absence of failures while testing can only show their presence [34]. Model checking is a fully automatic method, also known as the push-button method. Compared to testing, model checking has two advantages. First, model checking is capable of detecting difficult bugs and reproducing them. In addition, model checking guarantees the correctness of a software system when no errors are found during its exploration.

Figure 1.2 presents a one way of performing model-checking [36]. The center of model checking is a *model checker*, which takes a *formal model* and a *property* as inputs. While the formal model represents behaviors of the software system, the property describes a correctness property of the for-

mal model. The correctness property of a software system generally is constructed from the *system specification*. Two main classes of properties are safety properties, which guarantee nothing bad happens in the system, and liveness properties, which guarantee something good eventually happens in the system. Given a formal model and a property, the model checker determines whether the formal model satisfies the property. If it does, then the system satisfies the specification. Otherwise, a counter-example is generated to show that the system is not safe.

Designing a scalable model checker is difficult since it requires to deal with the state-space explosion problem. Precisely, a model checker should explore the full state space of a system. In fact, the number of states can be very large, or even infinite, and this makes the exploration of the full state space to be infeasible in practice. To address the state-space explosion problem, under/over-approximation techniques have been developed. Examples of under/over-approximation techniques are bounded model checking [21] and counter-example guided abstract refinement (CEGAR) [30, 7].

Bounded model checking combines model checking with satisfiability solving. Given a software program and a finite bound, e.g. the maximum number of the loop iterations or recursion calls in the program, bounded model checking unfolds the program according to the bound, then encodes the feasibility of all the resulting finite paths in a propositional formula. By doing so, checking correctness properties of the program can be reduced to checking the satisfiability of a propositional formula. Later, that formula is given to a propositional decision procedure, e.g. SAT-solvers [19, 61, 68], SMT-solvers [16, 39, 61], to either obtain a satisfying assignment or to prove there is none. In the former case, the program is not safe. In the latter case, the program is safe under the finite bound.

CEGAR finds an abstraction of the software system, before encoding the abstraction to a formula. If the formula is proved to be unsatisfiable by a satisfiability solver, then the system is safe. If the formula is satisfiable, then a counter-example is generated and checked whether it is spurious. In case the counter-example is spurious, the system is not safe. Otherwise, CEGAR refines its abstraction in order to remove the counter-example.

1.1.2 Satisfiability-based Approach for Testing and Model Checking

To have high code coverage in testing, we need to address the path feasibility problem. The problem is usually solved by a reduction to the satisfiability problem of formulas. More precisely, a program path, which is a sequence of program statements, is translated to equivalent constraints in static single assignment (SSA) form [11, 13]. One way for solving the satisfiability of formulas is to use SMT solvers [16, 39, 61]. Similarly, SMT solvers are used

in model checking methods, including bounded model checking [21] and CEGAR [30, 7], for solving the satisfiability problems of the obtained formulas.

There are several advantages of the satisfiability-based approach. The first advantage is re-usability. One can develop an off-the-shelf SMT solver, which can be used for different purposes such as extended static checking, predicate abstraction, test case generation, and bounded model checking. The second advantage is integrity. An SMT solver is a combination of multiple theories, e.g. string, arithmetic, or bit vector theories. Theories for a specific domain can be developed independently and integrated with other theories that have already been available in SMT solvers [16, 39, 61]. The desired theories depend on the types of program expressions to be analyzed. For example, arithmetic and string theories are often parts of SMT solvers since numbers and strings are two of the most common data types in programming languages. In fact, the more theories SMT solvers support, the more useful and applicable they are.

Text is such a common form of data that we use in everyday life, so the *string data type* is omnipresent in programming languages. For instance, JavaScript automatically converts primitives to string objects. Furthermore, various security vulnerabilities such as SQL injection and cross-site scripting attacks are caused by malicious string manipulation. According to Open Web Application Security Project [1], SQL injection (#1) and cross-site scripting (#3) are the most common and serious web attacks. Therefore, the satisfiability problem of string constraints has received considerable attention in the constraint solving community.

1.2 Satisfiability Problem for Analysis of String Constraints

In this section, we first introduce some motivating examples to show the importance of analysing of string constraints. We then review the preliminary background of the satisfiability problem for the analysis of string constraints, which is crucial for numerous application areas such as verification of string-manipulating programs [6] and analysis of security vulnerabilities of scripting languages (e.g., [82, 83, 96, 64]).

1.2.1 Motivating Examples

To show the undeniable impact of analysis of string constraints, we present three examples in the security area. The first two examples, which are Cross-site Scripting vulnerability and SQL injection, demonstrate the need of string theories for identifying security vulnerabilities. A detailed explanation of how to apply string theories to handle issues raised in the examples is also provided. The third example shows the practical limitation of current decidable

fragments of string constraints in application of checking a security vulnerabilities

Cross-site Scripting vulnerability
The first example demonstrates a cross-site scripting vulnerability. According to the Open Web Application Security Project [1], Cross-site Scripting is the third most serious web application vulnerability. Consider an HTML code as below.

```
<h1>
    User
    <p id="user" onMouseOver="display(‘{{info}}’)"> {{name}}
    </p>

    <script>
    function display(info) {
        document.getElementById(‘user’).
                setAttribute(‘title’, info);
    }
    </script>
</h1>
```

The HTML code above displays an user's public information on a webpage. For each user to be displayed - with their username and information stored in variables *name* and *info* respectively - the string `{{name}}` will be replaced by the value of *name*. Similarly, the string `{{info}}` will be replaced by the value of *info*. For example, an user *Alice* with her information `Student at Uppsala University` would result in the HTML below.

```
<p id="user" onMouseOver=
    "display(‘Student at Uppsala University’)"> Alice
</p>
```

The HTML then displays the text `User Alice` in the normal mode. When the mouse moves over the text, an additional text `Student at Uppsala University` will be shown next to the mouse position.

However, the code above can be attacked if the data is not validated, filtered, or escaped. Specifically, if *info* value contains some attacking pattern, e.g. `’); alert(‘Being attacked’); (‘`, the HTML code becomes the following:

```
<p id="user" onMouseOver=
    "display(‘’); alert(‘Being attacked’); (‘’)"> Alice
</p>
```

The malicious JavaScript `alert(‘Being attacked’)` will be executed when the mouse moves over the text. Notice that attackers can also replace `alert(‘Being attacked’)` by any other JavaScript code such as redirecting the page, or stealing authentication cookies.

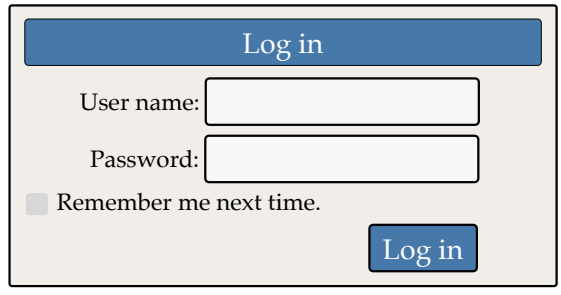

Figure 1.3. A basic user login form

SQL Injection

Injection flaw attacks are the most common web application vulnerabilities [1], and SQL injection is a typical example of these kinds of attacks. In the following, we present an SQL injection attack with a solution which uses string theories to avoid it.

To illustrate an SQL injection, we use an example of a login form, which requires users to log in by entering username and password before processing further given in Figure 1.3. Once the user's login information is filled, the user processes the login by tapping the `Log in` button. Afterward, the browser sends an authentication request to the application server. The implementation of the authentication is simplified as follows:

```
void LoginAuth(object sender, AuthenticateEventArgs e){
    SqlConnection con = new SqlConnection();
    SqlCommand query = new SqlCommand(
        "select * from users where user='" + Login.username +
        "' and passwd = '" + Login.password + "'");
    SqlDataAdapter adapter = new SqlDataAdapter(query);
    DataTable dt = new DataTable();
    adapter.Fill(dt);
    if (dt.Rows.Count >= 1){
        Response.Redirect("index.aspx");
    }
}
```

The authentication implementation above receives the user login information and checks whether the user can move forward. Initially, a SQL query is created with the given username and password. The query then searches in the application database to find if there exists a row that matches the login information. If it exists, then the authentication successfully completes and the user is redirected to the `index.aspx` page. Otherwise, nothing happens and the user is not allowed to log in.

Even though the implementation looks logical, it is vulnerable to an SQL injection attack. In particular, if an attacker provides malicious login information such as `Alice` to be the `Username`, and `' or '1' = '1` to be the `Password`, the attacker is always able to log in. The reason is that given the login information, the SQL query becomes `"select * from users where user='Alice' and passwd = '' or '1' = '1'"`. Since the SQL condition contains `'1' = '1'`, which is always true, the attacker can successfully log in. That simple trick allows him to steal `Alice` personal information without knowing her actual password.

Fortunately, SMT solvers with the support of string theories can help to avoid the above SQL injection attack and the similar ones. Specifically, we model an SQL injection attack as follows:

```
(query = "select * from users where user='" . username .
        "' and passwd = '" + password . "'") and
(query in L(Query_Grammar)) and
(query.contains("or " . x . "=" . x))
```

where `Query_Grammar` is defined as follows.

```
Query_Grammar : "select " TERM " from " TERM " where " EXPR
EXPR : COMP | EXPR or EXPR
COMP : TERM (=|>|<) TERM
TERM : [a-z]+[0-9]*| [0-9]+ | 'TERM'
```

The key to the solution is to model the SQL injection using string constraints and ask a string solver to check the constraint satisfiability. The first equality constraint represents the SQL *Select* query to the database. The equality constraint is simply obtained from the C# query command by replacing the plus string operators by concatenations. `username` and `password` are two string variables capturing the inputs username and password respectively. The second constraint is a membership constraint, which states that the query is valid according to SQL query grammar. The grammar defines the language of SQL *Select* query, which guarantees that `query` defined in the first constraint is a valid SQL query. The last constraint is a `contains` constraint, which checks if `query` can contain any SQL injection pattern. To prevent the above SQL injection attack, we can check if the `query` has any pattern of the form `or x = x`, where `x` is a string variable. Indeed, `query` captures the attack `or '1' = '1'`. After all necessary constraints are defined, a string solver supporting all types of constraints above, including equality, context-free membership, contain, and concatenation, is put into work. If the string solver answers unsatisfiable, then the web application is safe under the SQL injection attack. Otherwise, it is vulnerable to an SQL injection because there exists an SQL query such that it is used in the application and matches the SQL injection pattern.

Limitation of current decidable fragments of string constraints

The following pseudo-PHP script (a variation of a code at [90]) that prompts a user to change his password. The script is an example of a program that generates string constraints that fall out from all state-of-the-art decidable fragments of string constraints [64, 6].

```
$old=$database->real_escape_string($oldIn);
$new=$database->real_escape_string($newIn);
$pass=$database->query(
    "SELECT password FROM users WHERE userID=".$user);
if($pass == $old)
   if($new != $old)
      $query = "UPDATE users SET password=".$new.
                " WHERE userID=".$user;
      $database->query($query);
```

The user enters the old password `oldIn` and the new password `newIn`, which are sanitized and assigned to `old` and `new`, respectively. The old sanitized password is compared with the value `pass` from the database to authenticate the user. Moreover, the old sanitized password is also compared with the new sanitized password to ensure that a different password was chosen. If both comparisons hold, then the new password is finally saved in the database. The sanitization is present to prevent SQL injection attacks. To ensure that the sanitization works, we wish to verify that the SQL query `query` is safe, that is, it does not belong to a regular language *Bad* of dangerous inputs. This safety condition is expressed by the following constraints:

$$\begin{aligned}
&\texttt{new} = \mathscr{T}(\texttt{newIn}) \wedge \texttt{old} = \mathscr{T}(\texttt{oldIn}) \wedge \\
&\texttt{pass} = \texttt{old} \wedge \texttt{new} \neq \texttt{old} \wedge \\
&\texttt{query} = \texttt{"UPDATE users SET password="}.\texttt{new}.\texttt{" WHERE userID="}.\texttt{user} \wedge \\
&\texttt{query} \in \textit{Bad}
\end{aligned}$$

The sanitization on lines 1 and 2 of the PHP script is modeled by the transducer $\mathscr{T}$. Two comparisons over the old sanitized password are presented by the third and forth constraints. The fifth constraint models the SQL query to save in the database. Even though the above constraints look standard, they fall out of all existing decidable fragments of string constraints such as straight-line fragment [64] and acyclic fragment [5]. Specifically, the straight-line fragment does not allow more than one constraint over each string variable, thus the above constraints are not straight-line as there exists two constraints over `new`, namely $\texttt{new} = \mathscr{T}(\texttt{newIn})$ and $\texttt{new} \neq \texttt{old}$. Besides, the above constraints do not fall in the acyclic fragment because the fragment allows each string variable to occur only once while the string variable `new` occurs multiple times in $\texttt{new} = \mathscr{T}(\texttt{newIn})$, $\texttt{new} \neq \texttt{old}$, and $\texttt{query} = \texttt{"UPDATE users SET password="}.\texttt{new}.\texttt{" WHERE userID="}.\texttt{user}$.

In summary, strings are ubiquitous across computer science, and arguably more in web programming. Web applications often ask for user inputs in the form of strings before manipulating them to do other internal queries. If the input strings are not sanitized, the web applications will be vulnerable to attacks. Therefore, it is necessary to analyze string constraints to protect the applications.

1.2.2 What Kind of String Constraints Do We Need?

In this section, we present classes of string constraints that string solvers need to support. The classes should be expressive enough to satisfy the need of their applications.

Perhaps the most common string constraint is concatenation. Together with length constraints, concatenation constraints occupy 78% fragment of used string constraints in practice [82]. Indeed, various string operations, including `contain`, `index-of`, `substring`, etc, can be rewritten using the string concatenation. For instance, the constraint `contain(x, y)`, which checks whether string x contains string y, can be rewritten to `x = u · y · v`, where `u` and `v` are two extra string variables, '$\cdot$' is the concatenation operation. If the equality `x = u · y · v` holds, then `contain(x, y)` also holds. Otherwise, `contain(x, y)` does not. Existing string solvers, e.g. CVC4 [63], Norn [6], S3P [88] and Z3-str3 [98], support well concatenation constraints.

However, for application areas as verification and security, concatenation alone is insufficient. Finite state transducer and membership constraints should be supported by string solvers as well. First, finite state transducer constraints are used in detecting a variety of security vulnerabilities in web applications of a realistic browser model [95]. For example, in order to avoid cross-site scripting and SQL injection attacks, as shown in motivating examples in Section 1.2.1 and Section 1.2.1, web applications usually sanitize input strings before processing them. Standard sanitization functions include JavaScript-Escape and HTML-Escape implemented in The Google Closure Library. HTML-Escape converts special characters in HTML such as `&`, `>`, `'`, to their corresponding entities `&`, `>`, and `&#39;`. JavaScript-Escape converts special characters by adding backslash before the characters, e.g. `'` and `"` become `\'` and `\"`, respectively. Such sanitization functions can be encoded as finite-state transducers. Second, membership constraints are naturally used for describing languages of string variables and string patterns. On one hand, various inbuilt functions in JavaScript, e.g. `match` and `split`, are based on regular expressions as their parameters can be regular expressions. On the other hand, context-free membership constraints should be fully examined as they are used in the verification of the absence of SQL injection in the security area [87, 94], symbolic testing [59], and reasoning about the ambi-

guities or correctness of parsers [66]. Therefore, string solvers should support transducers and membership constraints.

Moreover, the above kinds of string constraints are also not sufficient in most applications such as verification and model checking. For example, string length constraints are one of the most popular constraints [82]. String solvers can be applied for verification of programs that often have non-string variables interacting with string variables via length and string-number conversion operations. It is worth mentioning that string-number conversion constraints should be considered by string solvers because of its popularity in practice as shown in Example 1.1.

Example 1.1 (The popularity of string-number conversion)**.** In fact, a script receiving string input tends to need to convert at least some of that input into numbers. The program fragment below is a variant of the Luhn test algorithm that is used in credit card and ID validation.

```
function checkLuhn(value) {
   var sum = 0;
   for (var i = value.length - 1; i >= 0; i-=2) {
       var d = parseInt(value.charAt(i));
       sum += d;
   }
   for (var i = value.length - 2; i >= 0; i-=2) {
       var d = parseInt(value.charAt(i));
       if ((d *= 2) > 9) d -= 9;
       sum += d;
   }
   var last= sum.toString().charAt(sum.length-1);
   return last == '0';
}
```

The input `value` of the Luhn test algorithm is a sequence of digits in a form of string. The algorithm processes the digits in the reversed order. The value of every odd digit (e.g., 1st, 3rd, etc.) is added to `sum` directly. For the value of every even digit, the algorithm (1) doubles its value, (2) subtracts its value by 9 if the doubled-value is larger than 9, and (3) adds the final result to `sum`. In the end, the input is validated if the last digit of `sum` is 0 (i.e., `sum mod 10=0`).

To check whether the program path that traverses both loops exactly once and finally passes this test has a valid input, we create the following (string) constraint:

$$
\begin{array}{ll}
1 & value_0 \in [1,9]^+ \wedge sum_0 = 0 \wedge \\
2 & i_0 = |value_0| - 1 \wedge \\
3 & d_0 = \mathsf{toNum}(\mathsf{charAt}(value_0, i_0)) \wedge \\
4 & sum_1 = sum_0 + d_0 \wedge \\
5 & i_1 = |value_0| - 2 \wedge \\
6 & d_1 = \mathsf{toNum}(\mathsf{charAt}(value_0, i_1)) \wedge \\
7 & sum_2 = sum_1 + \mathsf{ite}(d_1 * 2 > 9, d_1 * 2 - 9, d_1 * 2) \wedge \\
8 & i_2 = 0 \\
9 & last_0 = \mathsf{charAt}(\mathsf{toStr}(sum_2), |\mathsf{toStr}(sum_2)| - 1) \wedge \\
10 & last_0 = \text{“0”}
\end{array}
$$

Here $value_0$ and $last_0$ are string variables and the others are integer variables. The method $\mathsf{charAt}(x, i)$ returns the character at index i in the string x while $n = \mathsf{ite}(b, e, e')$ assigns to n the value of the expression e if b is true and the value of the expression e' otherwise. Line 1 describes the initial condition: value should be a sequence of digits and sum is initially zero. Lines 2-4 and lines 5-7 describe one execution of the first and second loops, respectively. Line 8 describes the condition on i_2 before leaving the loop. Finally, Lines 9-10 describe the condition that the last digit of *sum* is zero. Note that we use the following types of constraints to describe the program path:

- *regular* constraints (e.g., $value_0 \in [1,9]^+$, which says $value_0$ is in the regular language $[0,9]^+$),
- *integer* constraints (e.g., $i_0 = |value_0| - 1$, which says i_0 equals the length of $value_0$ minus one),
- *equality* constraints (often $y = \mathsf{charAt}(x, i)$ is encoded as $x = x_1.x_2.x_3 \wedge |x_1| = i \wedge |x_2| = 1 \wedge y = x_2$, which uses equality of string terms x and $x_1.x_2.x_3$), and
- *string-number conversion* (e.g., $\mathsf{toStr}(sum_2)$, which is the string value of the number sum_2).

□

Besides, the string-number conversion is also implicitly used in many programming languages. For instance, consider the following JavaScript example:

```
for(var i = 0; i < 10; i++) {
    arr[i] = 0;
}
```

A casual glance at the above code reveals no use of strings at all, but the semantics of field access is somewhat unusual in JavaScript: the arrays are indexed by strings, and numeric indices are converted to strings. This conversion is mandated explicitly by the JavaScript semantics: the 2019 edition of ECMAScript [41] requires that *ToPropertyKey* be called on the element expression (§12.3.2.1), and *ToPropertyKey* calls `ToString` on that value in all but special cases (§7.1.14). Therefore, any faithful string solvers supporting

symbolic execution of JavaScript must handle such conversions for even basic array operations to work correctly.

To summarize, the wide range of the commonly used primitives for manipulating strings requires string solvers to handle an expressive class of string constraints. The most important features that string solvers need to model are finite-state transducers, concatenation, membership, string length, and string-number conversion constraints.

1.2.3 String Constraints

In this section, we formally define string constraints. To begin with, we fix a finite alphabet $\Sigma \subseteq \mathbb{N}$. Note that here we assume that the alphabet is a finite subset of natural numbers. Essentially, we try to capture the numerical encoding of the corresponding symbols in computers (e.g., in ASCII, 'A' is encoded as 65). Hence, we can assume w.l.o.g. that there is a one-to-one mapping between numbers in Σ and the character it encodes. For the simplicity of the presentation, we assume that the character '0' is mapped to the number 0, '1' to 1,..., and '9' to 9. For other character c, we use $[\![c]\!]$ to denote the number that it maps to. Notice that this approach is general enough to support any finite set of characters.

A minor technical difficulty is that sometimes we may need to treat ε as a number. Therefore, we encode ε as some fixed number $[\![\varepsilon]\!] \in \mathbb{N} \setminus \Sigma$.

Assume that X is a set of *string variables* ranging over Σ^* and Z a set of *integer variables* ranging over $\mathbb{Z}$. An *interpretation over X and Z* is a mapping $I : X \cup Z \to \Sigma^* \cup \mathbb{Z}$. A *word term* is an element in X^*. We lift the interpretation I to word terms and linear constraints in the standard manner.

We use four types of *atomic string constraint*:

- An *equality constraint* ϕ_e is of the form $t_1 = t_2$ where t_1, t_2 are word terms. The *model* of ϕ_e is the set of interpretations $[\![\phi_e]\!] = \{I \mid I(t_1) = I(t_2)\}$. A *disequality constraint* ϕ_d is of the form $t_1 \neq t_2$ and is interpreted analogously.
- An *integer constraint* ϕ_i is a linear constraint over the integer variables in Z and values of $|x|$ for all $x \in X$, where $|\cdot| : X \to \mathbb{N}$ is the string length function defined in the standard way. We define $[\![\phi_i]\!] = \{I \mid I(\phi_i) = \mathsf{true}\}$.
- A *transducer constraint* ϕ_t is of the form $y \in \mathscr{T}(x)$ where x, y are string variables, and $\mathscr{T}$ is a transducer. We define $[\![\phi_t]\!] = \{I \mid I(y) \in \mathscr{T}(I(x))\}$.
- A *grammar constraint* ϕ_g is of the form $x \in L(G)$ where x is a string variable and G is a context-free grammar. The *model* of ϕ_g is the set of interpretations $[\![\phi_g]\!] = \{I \mid I(x) \in L(G)\}$. The special case of *regular constraint*, of the form $x \in L(R)$, where R is a regular expression over Σ is defined in a similar manner.
- A *string-number conversion constraint* ϕ_s is of the form $n = \mathsf{toNum}(x)$, where the function $\mathsf{toNum}(x)$ is defined as follows. For $a \in [0,9]$,

we have $\mathsf{toNum}(a) = a$ and for $w \cdot a \in [0,9]^+$, $\mathsf{toNum}(w \cdot a) = 10 \times \mathsf{toNum}(w) + a$. For $w \notin [0,9]^+$, $\mathsf{toNum}(w) = -1$. We define $[\![\phi_s]\!] = \{I \mid I(n) = \mathsf{toNum}(I(x))\}$. The *number-string* conversion constraint $x = \mathsf{toStr}(n)$ is treated as a syntactic sugar for $n = \mathsf{toNum}(x)$. We assume decimal encoding of numbers.

A *string constraint* is then a conjunction of atomic string constraints, with the semantics defined in the standard manner. It is *satisfiable* if there is an interpretation which evaluates the constraint to true.

Notice that only positive integer is supported in the string-number conversion function. This is the semantics used by most of the SMT solvers, and hence we follow it in this dissertation. The encoding has a benefit that it can also handle the case where x is "not a number", using the condition $\mathsf{toNum}(x) = -1$. Supporting only positive integer is not a strong restriction, since negative integers can still be encoded using only the positive version.

1.2.4 The Satisfiability Problem for Theories over Strings

The satisfiability problem for theories over strings has been the focus of various studies for a long time. This section reviews satisfiability results for different sub-classes over strings.

Already in 1946, Quine [79] showed that the first order theory of string equations is undecidable. A fundamental line of work has been to identify sub-classes for which decidability can be achieved. In 1977, Makanin [67] proved that the satisfiability problem for quantifier-free word equations, i.e., Boolean combinations of equalities and disequalities, where variables may denote words of arbitrary lengths, is decidable.

The decidability and complexity of different sub-classes have been considered by numerous works, e.g. [77, 78, 69, 80, 84, 47, 46]. In particular, Plandowski showed that the satisfiability of word equations with constants is in PSPACE in 1999 [77]. In 1999, various results on the satisfiability problem of sub-classes of word equations were found by Robson and Diekert [80]. In particular, the satisfiability problem of word equations where each variable occurs at most twice is NP-hard even for a single equation. If the lengths of variables in a possible solution are fixed, then the time complexity of the problem is linear. Robson and Diekert also addressed the problem with regular constraints: the uniform version is PSPACE-complete, and the non-uniform version is NP-hard. The result of Makanin on the satisfiability problem for quantifier-free word equations was generalized by Schulz in 1992. Schulz demonstrated that the solvability of word equations with variables remains decidable when variables are specified by regular languages.

More recently, Jez applied a new technique called re-compression to word equations, and proposed more efficient algorithms for many sub-classes of word equations [56, 57]. One important result is that the satisfiability of word

equations can be decided in linear space. However, there is a mismatch between this upper bound and the lower bound: solving word equations is NP-hard, but whether the problem is NP-complete remains open.

Since more functionalities than just word equations are required in practice, many research works extended the theory of word equations with certain functions e.g. linear arithmetic over the length, replace-all, and extract. Other works also extended the theory of word equations with predicates, e.g. numeric-string conversion predicate, regular-expression membership, etc. First, the satisfiability of word equations combined with length constraints of the form $|x| = |y|$ is open [24]. Also, the satisfiability problem for the `replace-all` function is undecidable [27]. More precisely, if pattern parameters of `replace-all` functions are variables, then the satisfiability problem is undecidable. If the pattern parameters are either constants or regular expressions, then the satisfiability problem is PSPACE-complete [27]. The undecidability holds when a numeric-string conversion predicate is combined with word equations [46].

Regular membership and transducer constraints were identified as a desirable feature of string languages, especially in the context of software analysis with an emphasis on security. Adding these kinds of constraints to the mix immediately leads to undecidability [72]. Hence numerous decidable fragments of sub-classes containing regular and transducer constraints were proposed [6, 15, 64, 27, 28]. Among these fragments, the straight line fragment of [64] is the most general decidable combination of concatenation, regular membership, and transducer constraints. Specifically, the satisfiability problem of the straight line fragment is EXPSPACE-complete, and under some certain reasonable assumptions, the complexity reduces to PSPACE. Another important decidable fragment is the acyclic fragment [6], which is incomparable with the straight line fragment. The acyclic fragment does not include transducer constraints but could be extended with them in a straightforward manner. Different syntactic features were considered by several works, including [27, 28]. However, the limit of decidable combinations of the core string features, including transducer, regular, length, and concatenation syntactic stays at [64] and [6]. Later in our paper III, we introduce a new decidable fragment, called *weakly-chaining*. The weakly-chaining fragment not only strictly subsumes the acyclic fragment of Norn but also the straight-line fragment. Thus it extends the known border of decidability for string constraints.

1.2.5 String Solvers

The past decade has witnessed a significant amount of progress in string constraint solving technologies thanks to the emergence of efficient SAT-solvers [19, 61, 68] and SMT-solvers [16, 39, 61]. Some notable robust string solvers are HAMPI [59], CVC4 [63], Stranger [96], Norn [6], S3P [88], Z3-str2 [98],

and Z3-str3 [18]. Based on the treatment of string lengths, string solvers are divided into three categories: fixed length solvers [59, 76, 71], bounded length solvers [82, 85], and unbounded length solvers [98, 63, 88, 6].

Fixed-Length String Solvers

Fixed-length string solvers were developed in the early 2000s and targeting at handling membership constraints. For example, HAMPI [59], which is one of fixed-length string solvers, encodes a string as a fixed-length bit-vector. HAMPI is able to handle both regular and context-free membership constraints. For a set of such constraints, given a fixed-length string variable, HAMPI can either return satisfiable with a concrete model or unsatisfiable. REGULAR$_L$ [76] is also another fixed-length string solver. It focuses on solving regular membership constraints, which are modeled in the solver by finite-state automata. Fixed-length bit-vector variables have been also explored in CP [71].

Bounded-Length String Solvers

Bounded-length string solvers [82, 85] are more flexible and expressive than fixed-length string solvers. Two of most well-known solvers in this category are KALUZA [82] and GECODE [85]. KALUZA solves constraints in two main steps. The first step is over-approximating of the constraints by creating the conjunction of explicit length constraints, length constraints implied by string constraints, and other integer constraints. The idea is to find a possible length of string variables using an SMT solver. If the SMT solver is unable to find the satisfiability assignments for lengths of variables, then KALUZA terminates and reports unsatisfiable. Otherwise, the second step is to under-approximate the constraints by bounding lengths of string variables based on satisfying assignments from the over-approximation. The under-approximated constraints are solved in a similar manner to HAMPI [59]. KALUZA repeats interaction between the over-approximation and the under-approximation until it finds the solution or proves unsatisfiable. The main limitation of KALUZA is that it cannot produce the satisfying model in the case that constraints are satisfiable, thus the satisfying answer is not reproducible. Besides, GECODE supports multiple string constraint types, including length, concatenation, and regular membership. GECODE models string variables in different ways such as bounded arrays, and block lists. The bounded length applied in GECODE often is fixed to a small number in the beginning. If no set of satisfying strings can be found, the solver increases the bound and then restarts the search. One mutual limitation of KALUZA and GECODE is the lack of transducer and context-free membership constraints support, thus their applicability is practically limited.

Unbounded-Length String Solvers

While both fixed-length and bounded-length string constraint problems are decidable as string variables have a finite domain, the decidability is no longer guaranteed when considering the unbounded-length. However, the significant development of SMT solvers, e.g. Z3, CVC4, allows unbounded-length string solvers to work efficiently. This generation of solvers includes Z3-str2 [98], Z3-str3 [18], CVC4 [63], S3 [88], Norn [6], and Ostrict [28]. In addition to having high performance, another advantage of these unbounded-length solvers is that they handle a variety of string constraints, including word equations, regular expression memberships, length constraints, and (more rarely) regular/rational relations to satisfy demands for verification and security areas. Since the decidability of the considered string constraints is no longer a guarantee, the solvers are not complete for the full combination of those constraints. They often only decide a (more or less well-defined) fragment of the individual constraints.

In particular, there are several approaches [88, 63, 97] on solving the satisfiability problem over string equations. These approaches are often based on the same technique that is string arrangement. Generally, string arrangements aim to recursively convert each equation to the disjunction of conjunctions of simpler equations. In order to do so, string arrangements align the concatenation of the left-hand side and the right-hand side of the equation. That leads to some splits in the string variables in the equation.

However, the string arrangement technique has a number of limitations. First, the search space explosion problem arises when (1) handling a large set of equations, and (2) expected results are long strings. The reason is, with a long string, the number of possible arrangements grows exponentially. Second, the recursion of the splitting strategy does not terminate if the problem is cyclic, for example, a variable exists on both sides of the equation. Last, the technique is unable to handle the context-free grammar or even the regular language in a complete manner. Therefore, to handle the regular language, it requires some under-approximation methods such as unfold-and-consume [88], which lazily unfolds the membership language until it finds a matching between constant string segments in the two sides of an equation.

Another technique for solving string constraints is *automata-based* [96, 93]. Stranger [96] soundly over-approximates string constraints using regular languages, and outperforms other string solvers when checking single execution traces according to [58]. It has recently been observed [93] that automata-based algorithms can be combined with model checking algorithms, in particular IC3/PDR, for more efficient checking of the emptiness for automata. However, many kinds of constraints such as length constraints, word equations, and context-free grammars, cannot be handled by automata-based solvers in a complete manner. Flat automata (or equivalently bounded languages [49, 50]) have also been used in the context of verification of concurrent recursive programs (e.g., [43, 44, 48, 65, 42, 12]). Later in Paper I and IV, we introduce our

framework using flat automata to define both over- and under-approximations of constraints. Our framework is general and allows the handling of all classes of constraints known to us from applications, including length constraints, word equations, and context-free grammars.

In summary, in our Paper II, III, and IV, we introduce our sequence of string solvers, named TRAU, TRAU+, and Z3-TRAU respectively. Our solvers are also unbounded-length solvers, and able to handle a diverse class of string constraints, such as concatenation, length, membership, and transducer constraints. The framework for the solvers is presented in Paper I and IV. In particular, we propose our framework using flat automata to define both over- and under-approximations of constraints, and introduce an algorithm to translate the satisfiability of constraints to the satisfiability of quantifier-free Presburger formulas. Therefore, the limitations of previous automata-based approaches are eliminated. As a result, a wider range of constraints can be handled, and satisfying assignments can be computed by our solvers.

1.3 Challenges in Satisfiability Problem for Analysis of String Constraints

Even though recent years have seen many works dedicated to the development of an efficient analysis of string manipulating programs, various challenging questions still remain. The major difficulty is that any reasonable comprehensive theory over strings is either undecidable or difficult to the degree that the decidability problem has been open for many years. Therefore, most existing string theories and string solvers are either not expressive enough, unsound, or unable to provide counter-examples. For example, some theories [95, 98] have tight restrictions on the expressiveness and the input language because they cannot handle word equations, length constraints, or transducer constraints. Other theories allow word equations but only accept if each variable appears at most once. Some string solvers [59, 95, 83] are not complete because they under-approximate constraints by applying upper bounds on the possible lengths of strings. The solver in [95], which performs an over-approximation, however, is incapable of generating a counter-example if the verification fails. Thus, challenges in the analysis of string constraints come from both theoretical and practical perspectives.

From a theoretical perspective, the satisfiability problem for a full class of string constraint with transducer, concatenation, and length constraints is undecidable in general [72, 28]. Even without transducer constraints, the decidability of analyzing a combination of concatenation and length constraints is still an open problem. Recent studies proposed that some sub-classes of string logics for which the satisfiability problem is decidable, e.g. Norn [6], the solved form fragment [47], and also the straight-line fragment [64, 52, 27]. However, those fragments are not expressive enough to cover all popular types

of complicated constraints, which derive from diverse applications such as analysis of security vulnerabilities of scripting languages, and verification of string manipulated programs. The issue raises the following questions:

(**Q1**) *How to find a general method for checking the satisfiability of string constraints that allows all popular constraints?*

(**Q2**) *What is a meaningful and expressive decidable fragment to extend the known border of decidability for string constraints?*

Paper I and Paper II respectively address the first and the second questions. To the best of our knowledge, our method proposed in Paper I has been the first method ever capable of handling unbounded context-free membership constraints. Being able to handle context-free membership constraints is such an important characteristic because the constraints have a wide application, e.g. security analysis in order to verify the absence of SQL injections [87, 94], reason about the ambiguities or correctness of parsers [66], and enable deeper symbolic testing [59]. Moreover, our decidable fragment *Weakly Chaining* in paper II can overcome the limitation of existing fragments. The weakly chaining fragment is strictly larger than the Norn fragment. It can also handle cases where the straight line fragment cannot handle them. More precisely, a straight-line constraint models a path through a string program in the single static assignment form, but as soon as the program compares two string variables, the string constraint falls out of the fragment. The weakly chaining fragment is liberal enough to accommodate constraints that are forbidden in the above fragments.

From the application perspective, the development of numerous string solvers, e.g. Z3-str3 [18], Z3-str2 [98], CVC4 [63], S3P [88, 89], have limited support of string constraint types and performance issues in some cases. For instance, Z3-str2 gives up and returns UNKNOWN in many cases, including simple cases where a variable appears in both sides of the equality, e.g. $\texttt{x} \cdot \texttt{y} = \texttt{y} \cdot \texttt{x}$. S3P does not support such important types of constraints as transducer and context-free membership constraints. Therefore, the open questions are the followings:

(**Q3**) *Does there exist an algorithm to translate the satisfiability problem of string constraint to the satisfiability problem of quantifier-free Presburger formulas in polynomial time?*

(**Q4**) *How to develop an efficient string solver to support all string constraints types?*

The third question is addressed in Paper I and Paper IV. First, we propose a fundamentally new method for checking the satisfiability of string constraints based on the concept of flattening. Specifically, domains of string variables are bounded by languages of flat automata, whose size can be adjusted if no satisfying assignments are found within the bounded domain. By doing so, the satisfiability of string constraints is translated to the satisfiability of quantifier-free Presburger formulas in polynomial time.

The fourth question is answered in Paper II, III, and IV. The center of these works is our string solver, named TRAU, and its later version Z3-TRAU. TRAU supports both standard string constraints such as concatenation, length, as well as more complicated and widely used ones as transduction and context-free membership. Z3-TRAU is a more recent version of TRAU as it is fully integrated to the Z3 SMT solver. In order to verify the efficiency of these solvers, we evaluate them in many existing and practical benchmarks with hundreds of thousands of test cases.

Furthermore, one interesting constraint operation is string-number conversion, which converts a string to a positive integer, or vice versa. String-number conversion is a fundamental operation and mentioned in core semantics of a majority of popular programming languages such as Python and JavaScript. The operation is practically used in many occasions, especially in JavaScript. Regardless of its popularity, there is a lack of support for this operation due to a number of difficulties. First, from the theoretical point of view, the problem is already proven to be undecidable [38]. Second, even if we apply some bounded techniques such as flattening, the generated constraints will contain both polynomials and exponentials because lengths of the involved strings in the operation are unknown. To our best of knowledge, the satisfiability problem of integer constraints with a mix of polynomials and exponentials is still open. Thus, the open question for string solvers is the following:

(**Q5**) *What is the procedure to handle string-number conversion constraints in an efficient, and a practical manner?*

Paper IV addresses the fifth question by introducing a class of parametric flat automata and its special subclass called numeric parametric flat automata. The numeric parametric flat automata help to translate the satisfiability problem of string-number conversion constraints to the satisfiability problem of a linear formula in polynomial-time. We show that our approach is effective based on the experiment results in various benchmarks. Noting that, we also propose new benchmarks targeting the string-number conversion operation with approximate 20000 test cases which are drawn from PyEx symbolic executors on practical programs.

2. Summary of Contributions

This chapter contains short summaries of our four peer-reviewed papers that are included in this dissertation. For each paper, we will explain the problem addressed and the paper main contributions.

2.1 Paper I: Flatten and Conquer: A Framework for Efficient Analysis of String Constraints

Recent years have seen a significant development of string solvers. The main reason for this interest is that string solvers are widely applied in various areas such as model checking, web programming, and security. Even though various string solvers exist, they have their own limitations. Specifically, despite the fact that string constraints come in various different forms, just few of them are supported by the majority of the existing string solvers. For instance, while regular expression membership and equality constraints are supported in Z3-str2 [98], Z3-str3 [18], CVC4 [63], S3 [88], and Norn [6], many other forms of string constraints such as membership in context-free grammars and transducers are not. Some solvers [98, 18] have performance issues as they are slow in practical benchmarks. Therefore, it is necessary to develop a string solver that can handle all types of constraints, and have a high performance. Designing such a solver is difficult because of the following reasons:

- Existing string solvers either bound lengths of string variables or do not support certain forms of constraints, e.g. transducer constraints, whose satisfiability is proven to be undecidable. Unfortunately, those constraints are used in many string operations, e.g. `replace-all` and `replace`, or handling security issues. Bounding string lengths is not good enough in practice if string solvers are used by testing and verification of software applications. For example, attackers can easily exploit a bug of an application by creating an attack involving a string whose length exceeds the bound. As a result, the full support of those types of constraints is needed.
- Combining solutions for all forms of string constraints in a single framework is a challenging problem as each constraint has its own representation and solution approach.
- Performance is another important factor to consider. In particular, to apply string solvers for practical problems, namely solving the reachability problem of programs, symbolic test generation, string solvers need to be capable of efficiently handling thousands of constraints at once.

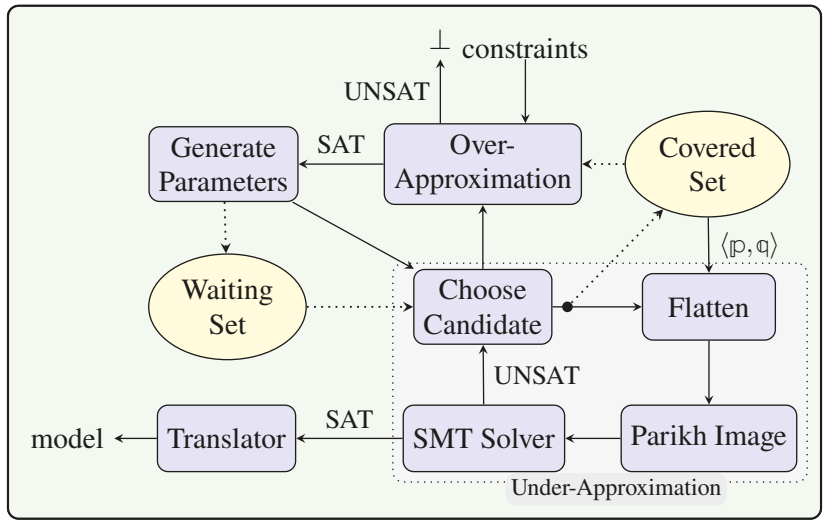

Figure 2.1. Overview of framework

In this paper, we propose a Counter-Example Guided Refinement (CEGAR) procedure to solve the satisfiability problem for string constraints. The framework is applicable to any class of constraints satisfying a sufficient condition, including context-free grammars and transducers. For more details, we make the following contributions:

1. *Algorithm*: We introduce an algorithm to translate the satisfiability of constraints to the satisfiability of quantifier-free Presburger formulas, thus allowing the use of powerful SMT solvers. The algorithm uses a classical concept from language theory, namely the Parikh image. The Parikh image of a word over a given alphabet counts the number of occurrences of each symbol in the word. The Parikh image of a language is the set of Parikh images of the words in the language. Then we say that a language is Parikh-definable if its Parikh image can be characterized by a quantifier-free Presburger formula. Since the SMT-solvers can check the satisfiability of such formulas, we can feed the generated formulas to an SMT-solver. The language is empty if and only if the SMT-solver concludes that the formula is unsatisfiable. It is worth mentioning that our algorithm can be applied to any class of constraints if its flattening is Parikh-definable.
2. *Framework*: We propose a new method for checking the satisfiability of string constraints as shown in Figure 2.1. The framework follows the CEGAR framework, which allows the flow of information between the under-approximation and the over-approximation. The under-approximation consists of three components including *Flatten, Parikh Image*, and *SMT*. The *Flatten* component makes use of a novel technique, called *flattening*. For purpose of defining flattening technique, we first

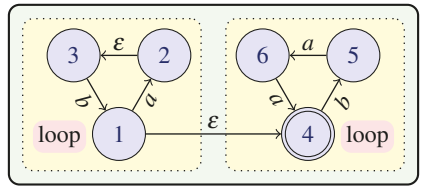

Figure 2.2. A $\langle 3,2\rangle$-flat automaton of $(ab)^* \bullet (baa)^*$.

define a flat automaton using an abstraction parameter consisting of a pair $\langle \mathbb{p}, \mathbb{q} \rangle$ of nature numbers. A run of a flat automaton iterates a sequence of $\mathbb{q}$ loops each corresponding to a fixed word of length at most $\mathbb{p}$. An example of a flat automaton is shown in Figure 2.2. Flattening of a constraint means that we perform an under-approximation in which we restrict the search for a solution to only those strings that are generated by a $\langle \mathbb{p}, \mathbb{q} \rangle$-flat automaton. The values for $\mathbb{p}$ and $\mathbb{q}$ are available in the *Waiting Set*. We solve the satisfiability of under-approximated constraints by computing *Parikh Image* of constraints and then feeding the Parikh constraints to a *SMT solver*. In the case that the under-approximation is able to find a solution, the input constraint is satisfiable. A satisfiable model is generated by the *Translator* component. In another case when the under-approximation fails to find a solution for an abstraction parameter $\langle \mathbb{p}, \mathbb{q} \rangle$, then $\langle \mathbb{p}, \mathbb{q} \rangle$ is added to the *Coverd Set*. The covered set is used for refining the *Over-Approximation* by excluding an infinite set of solutions captured by flat automata which their shape are in the covered set. The over-approximation creates an abstract of the input constraints. If the over-approximation produces a counter-example of the abstraction, then we use the counter-example to get new abstraction parameters and add the new abstraction parameters to the *Waiting Set*. Otherwise, if the over-approximated constraint is unsatisfiable, means the input constraint is unsatisfiable.

3. *Tool*: Based on our framework we implementd a tool, called TRAU. We experiment TRAU in a large set of benchmarks having thousands of test cases with diverse forms of constraints. The experiments not only demonstrate the efficiency of our tool compared to state-of-the-art solvers for string constraints but also show the generality and efficiency of our method. Specifically, we first carry an experiment on the Kaluza benchmark, which contains approximately 50000 test cases with mostly with length, concatenation, and regular constraints. The experimental results show that our string solver is more efficient than other solvers as TRAU is able to solve more constraints with the same time limit. We also carry another experiment on another set of benchmarks consisting of context-free membership constraints to verify the absence of SQL injections. The experimental results also show that our method is capable

of solving the constraints better than other solvers. It is worth mentioning that TRAU is the only string solver that can solve unbounded length context-free membership constraints.

2.2 Paper II: TRAU: SMT solver for string constraints

In this paper, we address the need for an efficient solver for wide ranges of string constraint types. Recall that string solvers can be used in a variety of verification approaches. The need for string solver, which can handle a combination of string operations: *concatenation* in word equations, to model assignments in programs; *context-free grammar,* to model properties or attack patterns; *string length,* to express string manipulation in programs; and *transduction,* to express sanitization, escape operations, and replacement operations in strings (see Section 2.1) has been grown in recent years.

We introduce our tool, called TRAU , for solving string constraints. TRAU is capable of handling not only standard constraints, e.g. lengths, concatenations, but also complicated ones such as transductions and context-free memberships in an efficient manner. To this end, we make the following contributions:

1. *Optimization of handling transducer constraints*: We optimize our method for handling transducer constraints by transforming transducer constraints to context-free membership constraints. To do so, we first construct a pushdown automaton $\mathscr{P}$ such that a word is accepted by $\mathscr{P}$ iff there are two words u and v such that $u \in \mathscr{T}(v)$ and $w = v \cdot \sharp \cdot u^R$ where $\sharp$ is a fresh symbol (not in Σ). Then let $\mathscr{G}$ be a context-free grammar that accepts the same language as the pushdown automaton $\mathscr{P}$ (i.e., $\mathscr{L}(\mathscr{G}) = \mathscr{L}(\mathscr{P})$). Let $\mathscr{G}_1$ (resp. $\mathscr{G}_2$) be the context-free grammar that accepts exactly the following set of words $\{w \cdot \sharp \cdot w^R \mid w \in \Sigma^*\}$ (resp. Σ^*). Now, we can replace the transducer constraint $t' \in \mathscr{T}(t)$ by the conjunction of the following context-free membership constraints: $t \cdot \sharp \cdot y \in \mathscr{G}$, $y \cdot \sharp \cdot t' \in \mathscr{G}_1$ and $t \cdot y \cdot t' \in \mathscr{G}_2$ where y is a fresh variable.
2. *Optimization of the under-approximation*: We optimize our method for handling equality constraints by combining the flattening technique proposed in [3] with the DPLL(T)-style proof procedure and the length-guided splitting of equalities procedure used in [5]. In particular, given a set of constraints ψ, a finite set of variables X, and an abstract parameter $\alpha = \langle \mathbb{p}, \mathbb{q} \rangle$, we proceed as follows: First, we construct the string constraint ψ' by replacing any occurrence of a variable x in ψ, that belongs to an $(\mathbb{p}, \mathbb{q})$-flat language, by $x_1 \cdot x_2 \cdots x_{\mathbb{q}}$ where $x_1, x_2, \ldots, x_{\mathbb{q}}$ are fresh variables that belong to $(\mathbb{p}, 1)$-flat languages. Assume w.l.o.g that ψ' contains an equality constraint ϕ_s of the form $x_1 \cdots x_m = y_1 \cdot y_2 \cdots y_n$. Observe that $x_1, \ldots, x_m, y_1, \ldots, y_n$ belong to $(\mathbb{p}, 1)$-flat languages. Then, for every $j : 1 \leq j \leq m$ (resp. $i : 1 \leq i \leq n$), we construct a string constraint φ (resp. φ') from ψ' by: (1) deleting the equality constraint ϕ_s

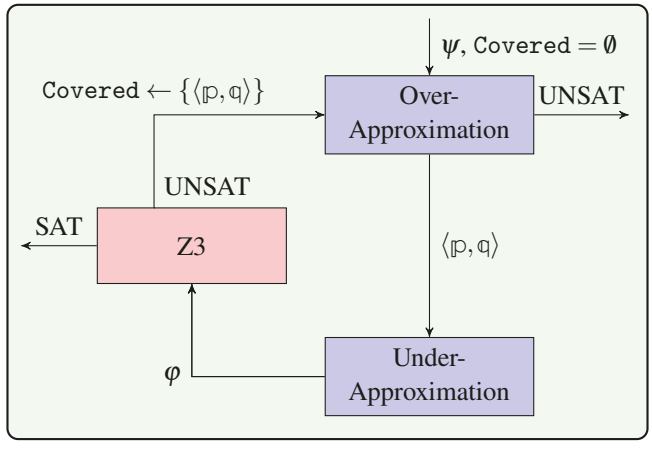

Figure 2.3. Architecture of TRAU

from ψ', (2) replacing any occurrence of the variable y_1 (resp. x_1) by $x_1 \cdot x_2 \cdots x_j$ (resp. $y_1 \cdot y_2 \cdots y_i$), and (3) adding the equality constraint $x_{j+1} \cdots x_m = y_2 \cdots y_n$ (resp. $x_2 \cdots x_m = y_{i+1} \cdots y_n$). For each string constraint φ (resp. ψ'), we repeat the procedure of splitting of the equality constraints until the obtained string constraint does not contain equality constraints. Finally, we declare the string constraint ψ to be satisfiable if one of the constructed string constraints is satisfiable; otherwise we add the abstract parameter $\alpha = \langle \mathbb{p}, \mathbb{q} \rangle$ to the set `Covered`.

3. *Tool*: TRAU implements the framework of Counter-Example Guided Abstraction Refinement (CEGAR) proposed in [7]. The architecture of TRAU is shown in Figure 2.3. TRAU consists of two main modules, namely the *Over-Approximation* module and the *Under-Approximation* module which have been introduced in Paper I. It uses the SMT solver Z3 to handle arithmetic constraints.
   We compare TRAU against four other state-of-the-art string solvers, namely CVC4 [63], S3P [88] and Z3-str3 [98], and TRAU-PRE [3], on a number of benchmarks. Each benchmark draws from the real world applications with diverse characteristics. The evaluation results show that TRAU performs better than other string solvers.

2.3 Paper III: Chain Free String Constraints

A number of string solvers [18], Z3-str3 [98, 18], CVC4 [63], S3P [88, 89], and TRAU [3, 4] have been developed in recent years thanks to the large demand in formal verification and security areas. Even though those string solvers can handle different kinds of string constraints, they have a common limitation relating to their capability of guaranteeing the termination. For example,

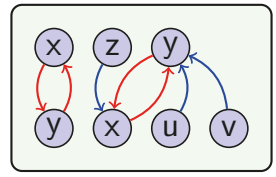

Figure 2.4. The splitting graph of $x = z \cdot y \wedge y = x \cdot u \cdot v$.

Z3-str3 does not terminate if the constraint is cyclic, e.g. a variable appears on both sides of equality. TRAU also does not guarantee termination as it is cannot prove unsatisfiability in general. The major reason is that the satisfiability problem for the full class of string constraints with concatenation, transduction, and length constraints is proven to be undecidable in general [72, 28]. The undecidability of the satisfiability problem even holds for a simple formula of the form $x \in \mathscr{T}(x)$ where $\mathscr{T}$ is a rational transducer and x is a string variable. Therefore, it is necessary to find meaningful and expressive sub-classes of string logics for which the satisfiability problem is decidable. Notable research works trying to expand the decidable class of string fragment are Norn [6], the solved form fragment [47], and also the straight-line fragment [64, 52, 27].

In this paper, we aim to not only find a new decidable fragment of string constraints but also show that the fragment is widely applicable in practice. To this end, we make the following contributions:

1. *Weakly-Chaining Fragment*: We first introduce the *chain-free* fragment. The main idea behind the chain-free fragment is to associate the set of relational constraints to a *splitting graph* where each node corresponds to an occurrence of a variable in the relational constraints of the formula as shown in Figure 2.4. An edge from an occurrence of a variable x to an occurrence of a variable y means that the source occurrence of x appears in a relational constraint which has on the opposite side an occurrence of y different from the target occurrence of y. The chain-free fragment prohibits loops in the graph, that we call *chains*, such as those red lines shown in red in Figure 2.4.
   Then, the *weakly chaining* fragment extends the chain-free fragment by allowing *benign* chains. Benign chains relate relational constraints where each left side contains only one variable, the constraints are all *length preserving*, and all the nodes of the cycles appear exclusively on the left or exclusively on the right sides of the involved relational constraints (as it is the case in Figure 2.4). Weakly chaining constraints may in practice arise from the checking that an encoding followed a decoding function is indeed the identity, i.e., of the form $\mathscr{T}_{\text{enc}}(\mathscr{T}_{\text{dec}}(x)) = x$. For instance, in situations similar to the above constraint, one might like to verify that the sanitization of a password followed by the application of a function supposed to invert the sanitization gives the original password.

2. *Decision Procedure*: The decision procedure for the weakly chaining formulas proceeds in several steps. The formula is transformed to an equisatisfiable chain-free formula, and then to an equisatisfiable concatenation free formula in which the relational constraints are of the form $\mathscr{T}(x,y)$ where x and y are two string variables and $\mathscr{T}$ is a transducer/relational constraint. Furthermore, we provide a decision procedure of a chain and concatenation-free formulae. The decision procedure is based on two techniques. First, we show that the chain-free conjunction over relational constraints can be turned into a single equivalent transducer constraint (in a similar manner as in [15]). Second, the consistency of the resulting transducer constraint with the input length constraints is checked via the computation of the Parikh image of the transducer.
3. *Tool*: The decision procedure of the weakly chaining fragment is implemented in our open-source tool, called TRAU+. TRAU+ is the later version of TRAU. TRAU+ follows the Counter-Example Guided Abstract Refinement (CEGAR) framework, thus it has two main components: over-approximation and under-approximation. The decision procedure specifically lies in the over-approximation component. The two components work together in order to automatically make the approximation more precise. Specifically, the over-approximation first takes the input and checks if it belongs to the weakly-chain fragment. If it is the case, then we use the decision procedure in (2). Otherwise, we choose a minimal set of occurrences of variables X that needs to be replaced by fresh variables such that the resulting constraints fall in our decidable fragment.

   TRAU+ is evaluated in multiple benchmarks, which contains approximately 30000 test cases. Evaluation results show that TRAU+ improves TRAU as TRAU+ is capable of handling transducer constraints in an efficient manner. Besides, TRAU+ performs either better, or at least as good as other string solvers [63, 18]. Interestingly, many test cases in our benchmarks belong to the weakly chain fragment but do not belong to other existing decidable fragments such as straight line or acyclic. TRAU+ shows its practicality when it can handle some cases while other string solvers cannot.

2.4 Paper IV: Efficient Handling of String-Number Conversion

String-number conversion is an undeniably important class of string constraints. Solving string-number conversion is necessary for the symbolic execution of string manipulating programs and the analysis of popular scripting languages such as JavaScript and Python, where string-number is a part of

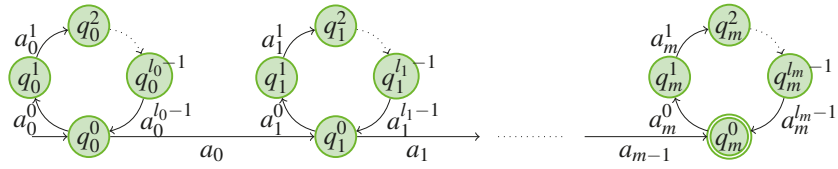

Figure 2.5. An example of a parametric flat automaton

the definition of the core semantics of these languages. In JavaScript, the array access operator, which is one of the most popular operators, relies on the string-number conversion. Specifically, JavaScript semantics states that a property P, which is in the form of string value, is an array index if and only if $\mathsf{toStr}(\mathsf{toNum}(P))$ is equal to P, where toNum is the function that converts a string to a number, and toStr is the function that converts from number to string.

However, solving this type of constraint is very challenging for the state-of-the-art string solvers. While other string operations, including string length, concatenation as well as other transducer-based operations, are well supported, string-number operations suffer from limited support in terms of the scale of formulas. One of the main reasons for solving string constraints with string-number conversion to be challenging is that it is proven to be undecidable [38].

In this paper, we present a new framework that efficiently handles string constraints containing string number conversion. In order to deal with the undecidability of the string-number conversion problem, our framework follows the CEGAR procedure to check the constraints satisfiability. Specifically, our framework under-approximates the constraints to make the constraints fall in a decidable fragment. If the under-approximated constraints is unsatisfiable, our framework over-approximates the constraints to check the unsatisfiability and to guide a new under-approximation. For more details, we make the following contributions.

1. *Parametric Flat Automata*: A class of *parametric flat automata* (PFA) is the key for efficient handling of string constraints. More precisely, a parametric automaton is defined as a tuple $P = \langle A, \psi \rangle$ where A is an automaton operating over an alphabet V of character variables, and ψ is an interpretation constraint over V. Given the definition of a parametric automaton, a PFA is a parametric automaton whose automaton is flat. An example of a PFA having m loops, each having a different size, is shown in Figure 2.5. PFA inherits properties of flat automata we have already introduced in Paper I. We use PFA in our under-approximation to bound domains of our string variables.
2. *Algorithm*: We propose an algorithm that translates the satisfiability problem of string constraints to the satisfiability problem of linear formulas in polynomial-time when the search space restricted by PFAs. The algorithm has two steps. The first step is to over-approximate the input

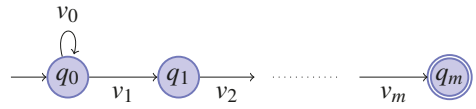

Figure 2.6. PFA for string-number conversion constraints

constraints into a set that falls in the chain-free fragment, which has been described in Paper III. If the satisfiability of the new set is unsatisfiable, then the satisfiability of the original input is also unsatisfiable. The second step is to under-approximate the string constraint by bounding the search space of each variable to the language defined by a PFA.

3. *Procedure for solving string-number conversion*: We present a decisive procedure for checking the satisfiability of string constraints with string-number conversion. First, we define a special PFA, called *numeric PFA*, for string variables that are involved in string-number conversion operators as shown in Figure 2.6. The numeric PFA covers all strings that have converted values up to 10^m with m is the length bound of the numeric PFA. Second, for each string variable restricted by a numeric PFA, its numeric value is represented by a linear formula.
4. *Tool*: We introduce an open-source tool Z3-TRAU implementing our algorithm at (2) and procedure at (3). We demonstrate the efficiency of Z3-TRAU on both existing and new benchmarks. In particular, we carry two sets of experiments. In the first set of experiments, we compare Z3-TRAU with other tools on existing benchmarks over basic string constraints. Those benchmarks do not involve string-number conversion operations. In the second set of experiments, we compare Z3-TRAU with the other tools on new suites focusing on string-number conversion. Our results of the experiments are the following:
   - Z3-TRAU performs as good as or better than the other tools in solving the satisfiability problems of basic string constraints. It ranks either 1st or 2nd in all benchmarks when comparing solved cases, and has the least number of timed out cases.
   - Z3-TRAU performs significantly better than the other tools in solving the satisfiability problems on string-number conversion benchmarks. It always ranks at the 1st on the number of solved cases. Indeed, the result shows the efficiency of PFA in general and *numeric* PFA in particular.

   All benchmarks in the two sets of experiments are drawn from practical examples. The existing benchmarks are collected by running symbolic executor PyEx over Python packages `httplib2`, `pip`, `pymongo`, and `requests`. Furthermore, we collect approximately 20000 new examples containing string-number conversion operators by running the symbolic executor Py-Conbyte on examples in Python core libraries. Those

examples involve diverse usages of string-number conversion operators, e.g. parsing date-time, verifying, and restoring IP addresses from strings.

3. Conclusions and Directions for Future Work

This thesis considers the satisfiability problem of string constraints. The thesis contributions include a novel framework for the analysis of string constraints, algorithms for translate the satisfiability problem of string constraints to the satisfiability problem of linear formula in polynomial-time, an expressive decidable fragment for string constraints, and efficient string solvers.

First, we introduced the background knowledge relevant to the understanding of our four peer-reviewed papers. We showed the importance and the difficulty of finding software failures in light of the evolution of the use of software systems over recent decades. Two techniques for finding software failures, e.g. testing and model checking, were reviewed. We then looked at the satisfiability-based approach for testing and model checking, where the core of the approach is an SMT solver.

Second, we discussed the paramount need for string theories as the extension of SMT solvers. In particular, string theories are extremely useful in the verification of string-manipulating programs and analysis of security vulnerabilities of scripting languages. Several examples to motivate the need were highlighted. We also reviewed related works from both theoretical and practical perspectives.

Last, we discussed various challenges of the analysis of string constraints.

In the summary of four peer-reviewed papers, we highlighted a number of new methods for checking the satisfiability of wide ranges of string constraints and propose our new decidable fragment for string constraints. Moreover, we introduced our powerful string solvers: TRAU, TRAU+, and Z3-TRAU, which are capable of solving various classes of constraints in an effective manner.

In Paper I, we presented a new method for checking the satisfiability of string constraints. The framework is applicable to diverse classes of string constraints, including not only standard constraints such as concatenation, length, but also more complicated ones such as context-free membership and transducer. It should be mentioned that our framework is the first framework ever that can support both membership and transducer constraints. Our framework is based on the CEGAR approach and has two components: under-approximation and over-approximation. The under-approximation uses flat automata to restrict the search for a solution to only strings generated by a flat automaton. The over-approximation abstracts the input constraints and produces a counter-example of the abstraction. Using the counter-example, our algorithm then generates a parameter indicating shapes of flat automata that will be used in the under-approximation. We implemented our string solver TRAU and run it successfully on various benchmarks.

In Paper II, we proposed a number of optimizations for our string solver TRAU. One of the notable optimizations is for the handling of transducer constraints by converting them to context-free membership constraints using pushdown automata. Another optimization is combining the flat-automata based method in Paper I with the split based method [5] in the implementation of the under-approximation. The experimental results of TRAU show that it is remarkably better than its predecessor, and also better than other string solvers, e.g. Z3-str3 [18], CVC4 [63], and S3 [88].

In Paper III, we addressed the issue of previous decidable fragments of string constraints, e.g. acyclic [6], and straight-line [64], and introduced new decidable fragments for string constraints, call *Chain Free* and *Weakly Chaining*. The chain free fragment prohibits loops in the splitting graph where each node corresponds to an occurrence of a variable in the relational constraints of the formula. The weakly chain fragment extends the chain free fragment as it allows loops of variables involving in the length preserving constraints. The weakly chaining fragment is strictly larger than the acyclic fragment and the straight-line fragment, so it covers important cases which fall out other decidable fragments, e.g comparisons in static single assignments [29]. We then proposed a decision procedure of the weakly chaining fragment. We implemented the decision procedure in the over-approximation of our string solver TRAU+. The evaluation shows that with a better over-approximation, TRAU+ works better than its predecessor TRAU as TRAU+ is capable of handling more test cases than TRAU in various benchmarks. TRAU+ also outperforms other string solvers such as Ostrict [28], Z3-str3 [18], and CVC4 [63], especially in test cases falling in the weakly chaining fragment.

In Paper IV, we focused on solving string-number conversion constraints. Specifically, we introduced a class of parametric flat automata (PFA) for efficient handling of string constraints, and a special class of PFA for string-number conversion constraints. We also presented an algorithm to translate the satisfiability of string constraints to the satisfiability problem of linear formulas in polynomial-time when the search space restricted by PFAs. We implemented the algorithm in our solver Z3-TRAU and carry out numerous evaluations on standard benchmarks as well as string-number conversion benchmarks for the purpose of verifying the efficiency of our algorithm. The standard benchmarks use such types of constraints as concatenation, regular membership, and length. The string-number conversion benchmarks, which are collected by running symbolic execution on various practical examples, involve the use of string-number conversion constraints in combination with other constraints. Evaluation results show that Z3-TRAU works as good as or better than other string solvers, e.g Z3-str3 [98], and CVC4 [63] on standard benchmarks. Z3-TRAU also works considerably better than other string solvers in string-number conversion benchmarks.

Future work.
In Paper I, there are several open and difficult questions to consider in the future. First and foremost, we believe that our framework can be extended by applying the idea of symbolic automata and transducers [37] to our flat automata, thus enhancing our under-approximation. Second, even though our method supports context-free membership constraints, the syntax of the constraints is not standardized as a part of the SMTLIB2 language. Therefore, formalizing the syntax for the context-free membership constraint is an essential task to increase its popularity. Lastly, our string solver can also be applied to other string-related problems such as program inversion [53] thanks to the richness of supported languages.

In Paper II, the future work consists of supporting general transducer constraints and extending classes of supported string constraints. First, even though our solver TRAU can support transducer-based string constraints, e.g. `replace`, `to-upper`, and `to-lower`, TRAU does not support general transducer constraints. Consequently, it is unable to run some benchmarks such as SLOG. Second, TRAU does not support essential string constraints such as string-number conversion, `replace-all` where the replacement pattern is variable. Besides, it just supports `not-contain` constraints in an incomplete manner, while `not-contain` constraints apparently occur in `index-of`, `replace`, `replace-all`, and many other types of constraints. All the above future developments are crucial from both theoretical and practical points of view.

There are several directions to further develop the work in Paper III. The first direction is to extend existing decidable fragments of string constraints [5, 64, 52, 27]. Despite the fact that many research works target at solving the decidability problem of string constraints, there remain long-standing open problems, e.g. the decidability of word equations in combination with length constraints. String solvers [98, 63, 3, 52] can greatly benefit from the decidability result as most of them cannot guarantee termination due to the theoretical barrier. The second direction is to fully integrate the decision procedure of the weakly chaining fragment into our string solver TRAU. More precisely, the current implementation of the decision procedure has been done in SLOTH, then we externally use SLOTH as the over-approximation component of TRAU. If the integration can be done internally, then it undoubtedly improves TRAU performance. Simultaneously, SLOTH's limitations such as lack of support for length constraints can be covered by TRAU.

In Paper IV, as we only consider the string-number conversion for positive integers, one arising question is whether our framework can be generalized with negative and real numbers. In practice, both negative and real numbers are popularly used, e.g. `parseInt` and `parseFloat` functions in JavaScript parse a string and return an integer and a floating point number respectively. Since current semantics of string-number conversion constraints used in most string theories allow only positive integers, we need to update the semantics before extending the framework to support negative integers. That can be done

by extending the *numeric PFA* to accept a '−' character at the beginning of automata. Handling real numbers is more challenging. Another direction is to optimize our over-approximation for string-number conversion constraints. In particular, the current over-approximation of string-number conversion constraints does not take into account integer domains covered by numeric PFAs in under-approximation iterations. By strengthening the over-approximation, we expect not only improving the solver performance but also proving unsatisfiable in many more cases. The last direction to consider is migrating our string solver to Z3. Despite the fact that there are several string theories in Z3 such as Z3-str3 and Z3-seq, they do not support the string-number conversion constraints as effectively as our string theory Z3-TRAU. Therefore, migration will definitely improve Z3 in solving string constraints, and also show the practicality of Z3-TRAU.

References

[1] OWASP. `https://owasp.org/`.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Julian Dolby, Petr Janků, Hsin-Hung Lin, Lukáš Holík, and Wei-Cheng Wu. Efficient handling of string-number conversion. In *PLDI 2020*, page 943–957. ACM, 2020.

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukás Holík, Ahmed Rezine, and Philipp Rümmer. Flatten and conquer: a framework for efficient analysis of string constraints. In *PLDI 2017*, pages 602–617. ACM, 2017.

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukás Holík, Ahmed Rezine, and Philipp Rümmer. Trau: SMT solver for string constraints. In *FMCAD 2018*, pages 1–5. IEEE, 2018.

[5] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. String constraints for verification. In *CAV 2014*, pages 150–166, Cham, 2014. Springer.

[6] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. Norn: An SMT solver for string constraints. In *CAV 2015*, volume 9206 of *LNCS*, pages 462–469. Springer, 2015.

[7] Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Bui Phi Diep. Counter-example guided program verification. In *FM 2016*, pages 25–42. Springer, 2016.

[8] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bui Phi Diep, Lukás Holík, and Petr Janku. Chain-free string constraints. In *ATVA 2019*, volume 11781 of *LNCS*, pages 277–293. Springer, 2019.

[9] Parosh Aziz Abdulla, Karlis Čerāns, Bengt Jonsson, and Tsay Yih-Kuen. General decidability theorems for infinite-state systems. In *LICS 1996*, pages 313–321, 1996.

[10] Parosh Aziz Abdulla and Giorgio Delzanno. Parameterized verification. *STTT 2016*, 18(5):469–473, 2016.

[11] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, USA, 2nd edition, 2003.

[12] Mohamed Faouzi Atig, K. Narayan Kumar, and Prakash Saivasan. Acceleration in multi-pushdown systems. In *TACAS 2016*, volume 9636 of *LNCS*, pages 698–714. Springer, 2016.

[13] John Aycock and Nigel Horspool. Simple generation of static single-assignment form. In David A. Watt, editor, *Compiler Construction*, pages 110–125. Springer, 2000.

[14] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[15] Pablo Barceló, Diego Figueira, and Leonid Libkin. Graph logics with rational relations. *Logical Methods in Computer Science*, 9(3), 2013.
[16] Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer, Cham, 2018.
[17] Boris Beizer. *Software Testing Techniques (2nd Ed.)*. Van Nostrand Reinhold Co., USA, 1990.
[18] Murphy Berzish, Yunhui Zheng, and Vijay Ganesh. Z3str3: A string solver with theory-aware branching. *CoRR*, abs/1704.07935, 2017.
[19] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, NLD, 2009.
[20] Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. *Electr. Notes Theor. Comput. Sci.*, 66(2):160–177, 2002.
[21] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.
[22] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS 1999*, pages 193–207, 1999.
[23] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.
[24] J. R. Büchi and S. Senger. Definability in the existential theory of concatenation and undecidable extensions of this theory. *Z. Math. Logik Grundlagen Math.*, 34(4), 1988.
[25] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10ˆ20 states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.
[26] Prantik Chatterjee, Subhajit Roy, Bui Phi Diep, and Akash Lal. Distributed bounded model checking. *ArXiv*, abs/2005.08063, 2020.
[27] Taolue Chen, Yan Chen, Matthew Hague, Anthony W. Lin, and Zhilin Wu. What is decidable about string constraints with the replaceall function. *Proc. ACM Program. Lang.*, 2(POPL), December 2018.
[28] Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proc. ACM Program. Lang.*, 3(POPL):49:1–49:30, January 2019.
[29] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *TACAS 2004*, pages 168–176. Springer, 2004.
[30] Edmund M. Clarke. Counterexample-guided abstraction refinement. In *TIME-ICTL 2003*, page 7, 2003.
[31] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
[32] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron A. Peled. State space reduction using partial order techniques. *STTT 1999*, 2(3):279–287, 1999.
[33] Edmund M. Clarke, Anubhav Gupta, and Ofer Strichman. SAT-based counterexample-guided abstraction refinement. *IEEE Trans. on CAD of*

Integrated Circuits and Systems, 23(7):1113–1123, 2004.
[34] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018.
[35] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model checking and the state explosion problem. In *LASER 2011*, pages 1–30, 2011.
[36] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Proc. IBM workshop on Logics of Programs*, volume 131 of *LNCS*, 1982.
[37] Loris D'Antoni and Margus Veanes. The power of symbolic automata and transducers. In *CAV 2017*, volume 10426 of *LNCS*, pages 47–67. Springer, 2017.
[38] Joel D Day, Vijay Ganesh, Paul He, Florin Manea, and Dirk Nowotka. The satisfiability of word equations: Decidable and undecidable theories. In *RP 2018*, pages 15–29. Springer, 2018.
[39] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011.
[40] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.
[41] ECMA, European Computer Manufacturers Association, et al. Ecmascript language specification, 2019.
[42] Javier Esparza and Pierre Ganty. Complexity of pattern-based verification for multithreaded programs. In *POPL 2011*, pages 499–510. ACM, 2011.
[43] Javier Esparza, Pierre Ganty, and Rupak Majumdar. A perfect model for bounded verification. In *LICS 2012*, pages 285–294. IEEE Computer Society, 2012.
[44] Javier Esparza, Pierre Ganty, and Tomás Poch. Pattern-based verification for multithreaded programs. *ACM Trans. Program. Lang. Syst.*, 36(3):9:1–9:29, 2014.
[45] A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *TCS 2001*, 256(1-2):63–92, 2001.
[46] Vijay Ganesh and Murphy Berzish. Undecidability of a theory of strings, linear arithmetic over length, and string-number conversion. *CoRR*, abs/1605.09442, 2016.
[47] Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin Rinard. Word equations with length constraints: What's decidable? In *Hardware and Software: Verification and Testing*, volume 7857 of *LNCS*, pages 209–226. Springer, 2013.
[48] Pierre Ganty, Rupak Majumdar, and Benjamin Monmege. Bounded underapproximations. *Formal Methods in System Design*, 40(2):206–231, 2012.
[49] Seymour Ginsburg. *The Mathematical Theory of Context-Free Languages*. McGraw-Hill, Inc., 1966.
[50] Seymour Ginsburg and Edwin H. Spanier. Bounded algol-like languages. *Transactions of the American Mathematical Society*, 113(2):333–368, 1964.
[51] Patrice Godefroid. Compositional dynamic test generation. In *POPL 2007*, page 47–54. ACM, 2007.
[52] Lukás Holík, Petr Janku, Anthony W. Lin, Philipp Rümmer, and Tomás Vojnar. String constraints with concatenation and transducers solved efficiently.

PACMPL, 2(POPL):4:1–4:32, 2018.
[53] Qinheping Hu and Loris D'Antoni. Automatic program inversion using symbolic transducers. *SIGPLAN Not.*, 52(6), June 2017.
[54] Jonathan Jacky. Model-based testing with Spec#. In *ICFEM 2004*, pages 5–6, 2004.
[55] Jonathan Jacky, Margus Veanes, Colin Campbell, and Wolfram Schulte. *Model-Based Software Testing and Analysis with C#.* Cambridge University Press, January 2008.
[56] Artur Jeundefined. Recompression: A simple and powerful technique for word equations. *J. ACM*, 63(1), February 2016.
[57] Artur Jez. Word equations in nondeterministic linear space. In *ICALP 2017*, volume 80 of *LIPIcs*, pages 95:1–95:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.
[58] Scott Kausler and Elena Sherman. Evaluation of string constraint solvers in the context of symbolic execution. In *ASE 2014*, pages 259–270. ACM, 2014.
[59] A. Kieżun, V. Ganesh, S. Artzi, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.*, 21(4):25, 2012.
[60] Edward Kit and Susannah Finzi. *Software Testing in the Real World: Improving the Process*. ACM Press/Addison-Wesley Publishing Co., USA, 1995.
[61] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 1 edition, 2008.
[62] Nancy G. Leveson. *Safeware - system safety and computers: a guide to preventing accidents and losses caused by technology*. Addison-Wesley, 1995.
[63] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *CAV 2014*, volume 8559 of *LNCS*, pages 646–662. Springer, 2014.
[64] Anthony Widjaja Lin and Pablo Barceló. String solving with word equations and transducers: towards a logic for analysing mutation XSS. In *POPL 2016*, pages 123–136. ACM, 2016.
[65] Zhenyue Long, Georgel Calin, Rupak Majumdar, and Roland Meyer. Language-theoretic abstraction refinement. In *FASE 2012*, volume 7212 of *LNCS*, pages 362–376. Springer, 2012.
[66] Ravichandhran Madhavan, Mikaël Mayer, Sumit Gulwani, and Viktor Kuncak. Automating grammar comparison. In *OOPSLA 2015*, page 183–200. ACM, 2015.
[67] G.S. Makanin. The problem of solvability of equations in a free semigroup. *Mathematics of the USSR-Sbornik*, 32(2), 1977.
[68] Sharad Malik and Lintao Zhang. Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM*, 52(8):76–82, August 2009.
[69] Yuri Matiyasevich. Computation paradigms in light of hilbert's tenth problem. In *New Computational Paradigms*, pages 59–85. Springer, 2008.
[70] Kenneth L. McMillan. *Symbolic model checking*. Kluwer, 1993.
[71] Laurent D. Michel and Pascal Van Hentenryck. Constraint satisfaction over bit-vectors. In *CP 2012*, pages 527–543. Springer, 2012.
[72] Christophe Morvan. On rational graphs. In *FOSSACS 2000*, pages 252–266. Springer, 2000.

[73] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI 2007*, pages 446–455, 2007.
[74] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley Publishing, 3rd edition, 2011.
[75] Doron A. Peled. All from one, one for all, on model-checking using representatives. In *CAV 1993*, volume 697 of *LNCS*, pages 409–423, 1993.
[76] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In *CP 2004*, pages 482–495. Springer, 2004.
[77] Wojciech Plandowski. Satisfiability of word equations with constants is in PSPACE. In *FOCS*, pages 495–500, 1999.
[78] Wojciech Plandowski. An efficient algorithm for solving word equations. In *STOC*, pages 467–476, 2006.
[79] W. V. Quine. Concatenation as a basis for arithmetic. *J. Symb. Log.*, 11(4), 1946.
[80] John Michael Robson and Volker Diekert. On quadratic word equations. In *STACS*, 1999.
[81] O. Saarikivi, K. Kähkönen, and K. Heljanko. Improving dynamic partial order reductions for concolic testing. In *ACSD 2012*, pages 132–141, 2012.
[82] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *2010 IEEE Symposium on Security and Privacy*, pages 513–528, 2010.
[83] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDSS*. The Internet Society, 2010.
[84] Klaus U. Schulz. Makanin's algorithm for word equations - two improvements and a generalization. In *IWWERT*, pages 85–150, 1990.
[85] Joseph D. Scott, Pierre Flener, and Justin Pearson. Constraint solving on bounded string variables. In *CPAIOR 2015*, pages 375–392, Cham, 2015. Springer.
[86] K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *HVC 2006*, pages 166–182, 2006. LNCS 4383.
[87] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. *SIGPLAN Not.*, 41(1):372–382, January 2006.
[88] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *CCS 2014*, pages 1232–1243. ACM, 2014.
[89] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. Progressive reasoning over recursively-defined strings. In *CAV 2016*, volume 9779 of *LNCS*, pages 218–240. Springer, 2016.
[90] TwistIt.tech. PHP tutorials. `https://www.makephpsites.com/php-tutorials/user-management-tools/changing-passwords.php`, 2019. [Online; accessed 2019-04-29].
[91] A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets*, volume 483 of *LNCS*, pages 491–515, 1990.
[92] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *ISSTA 2004*, pages 97–107, 2004.

[93] Hung-En Wang, Tzung-Lin Tsai, Chun-Han Lin, Fang Yu, and Jie-Hong R. Jiang. String analysis via automata manipulation with logic circuit representation. In *CAV 2016*, volume 9779 of *LNCS*, pages 241–260. Springer, 2016.

[94] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. *SIGPLAN Not.*, 42(6):32–41, June 2007.

[95] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song. A systematic analysis of xss sanitization in web application frameworks. In *ESORICS 2011*, pages 150–171. Springer, 2011.

[96] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Stranger: An automata-based string analysis tool for PHP. In *TACAS 2010*, volume 6015 of *LNCS*, pages 154–157. Springer, 2010.

[97] Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Julian Dolby, and Xiangyu Zhang. Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In *CAV 2015*, volume 9206 of *LNCS*, pages 235–254. Springer, 2015.

[98] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A Z3-based string solver for web application analysis. In *ESEC/FSE 2013*, pages 114–124. ACM, 2013.

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1998

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-428900

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2021