EPTCS 342

Proceedings of the

9th International Symposium on Symbolic Computation in Software Science

Hagenberg, Austria, September 8-10, 2021

Edited by: Temur Kutsia

Published: 6th September 2021
DOI: 10.4204/EPTCS.342
ISSN: 2075-2180
Open Publishing Association

Preface

Symbolic Computation is the science of computing with symbolic objects (terms, formulae, programs, representations of algebraic objects etc.). Powerful algorithms have been developed during the past decades for the major subareas of symbolic computation: computer algebra and computational logic. These algorithms and methods are successfully applied in various fields, including software science, which covers a broad range of topics about software construction and analysis.

Meanwhile, artificial intelligence methods and machine learning algorithms are widely used nowadays in various domains and, in particular, combined with symbolic computation. Several approaches mix artificial intelligence and symbolic methods and tools deployed over large corpora to create what is known as cognitive systems. Cognitive computing focuses on building systems which interact with humans naturally by reasoning, aiming at learning at scale.

The purpose of the International Symposium on Symbolic Computation in Software Science – SCSS is to promote research on theoretical and practical aspects of symbolic computation in software science, combined with modern artificial intelligence techniques. SCSS 2021 is the ninth edition in the SCSS symposium series. It was organized at the Research Institute for Symbolic Computation (RISC) of the Johannes Kepler University Linz. Due to the COVID-19 pandemic, the symposium was held completely online.

The SCSS program featured a keynote talk by Bruno Buchberger (Johannes Kepler University Linz) and three invited talks given by Tateaki Sasaki (University of Tsukuba), Martina Seidl (Johannes Kepler University Linz), and Stephen M. Watt (University of Waterloo). The symposium received 25 submissions with contributing authors from 17 countries. These submissions have been divided into two tracks: 16 in the category of regular papers and tool/dataset descriptions, and nine in the category of short and work-in-progress papers. Twenty PC members and 15 external reviewers took part in the refereeing process, after which 10 regular / dataset papers and 9 short contributions have been accepted for the presentation at the symposium. The accepted regular and tool papers are included in these proceedings. The short papers appeared in a collection published as a RISC technical report. In addition to the main program, a special session on Computer Algebra and Computational Logic has been held.

On behalf of the Program Committee, I thank the authors of the submitted papers for considering SCSS as a venue for their work and the keynote and invited speakers for their inspiring talks. The PC members and external reviewers deserve thanks for their careful reviews. The EasyChair conference management system has been a very useful tool for PC work. The technical support team at RISC greatly contributed to running the conference smoothly. Finally, I thank all the participants for contributing to the success of the symposium and EPTCS and arXiv for hosting the proceedings.

Temur Kutsia
Program Chair of SCSS 2021

T. Kutsia (Ed.): Symbolic Computation
in Software Science (SCSS'21)
EPTCS 342, 2021, pp. i–ii, doi:10.4204/EPTCS.342.0

Conference Information

General Chairs

Adel Bouhoula (Arabian Gulf University, Bahrain)
Tetsuo Ida (University of Tsukuba, Japan)

Program Chair

Temur Kutsia (RISC, Johannes Kepler University Linz, Austria)

Program Committee

David Cerna (Czech Academy of Sciences, Czechia, and Johannes Kepler University Linz, Austria)
Changbo Chen (Chinese Academy of Sciences, China)
Rachid Echahed (CNRS, Grenoble, France)
Seyed Hossein Haeri (UC Louvain, Belgium)
Mohamed-Bécha Kaâniche (Sup'Com, Carthage University, Tunisia)
Cezary Kaliszyk (University of Innsbruck, Austria)
Yukiyoshi Kameyama (University of Tsukuba, Japan)
Michael Kohlhase (University of Erlangen-Nuremberg, Germany)
Laura Kovács (Vienna University of Technology, Austria)
Zied Lachiri (ENIT, University of Tunis El Manar, Tunisia)
Christopher Lynch (Clarkson University, USA)
Mircea Marin (West University of Timisoara, Romania)
Yasuhiko Minamide (Tokyo Institute of Technology, Japan)
Yoshihiro Mizoguchi (Kyushu University, Japan)
Julien Narboux (Strasbourg University, France)
Michaël Rusinowitch (INRIA, France)
Wolfgang Schreiner (Johannes Kepler University Linz, Austria)
Sofiane Tahar (Concordia University, Canada)
Dongming Wang (CNRS, Paris, France)

External Reviewers

Behzad Akbarpour, Deena Awny, Xiaoyu Chen, Zecheng He, Dongchen Jiang, Hai Lin, Chenqi Mou, Hiroshi Ohtsuka, Florian Rabe, Adnan Rashid, Stefan Ratschan, Takafumi Saikawa, Jan Frederik Schaefer, Ionut Tutu, Laurent Vigneron

Organization

Temur Kutsia
Cleopatra Pau
Werner Danielczyk-Landerl
Ralf Wahner

Table of Contents

Symbolic Computation in Software Science: My Personal View

Bruno Buchberger
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University
Linz / Schloss Hagenberg, Austria
Bruno.Buchberger@risc.jku.at

In this note, I develop my personal view on the scope and relevance of symbolic computation in software science. For this, I discuss the interaction and differences between symbolic computation, software science, automatic programming, mathematical knowledge management, artificial intelligence, algorithmic intelligence, numerical computation, and machine learning. In the discussion of these notions, I allow myself to refer also to papers (1982, 1985, 2001, 2003, 2013) of mine in which I expressed my views on these areas at early stages of some of these fields.

The Intention of This Note

It is a great joy to see that the SCSS (Symbolic Computation in Software Science) conference series, this year, experiences its 9th edition. A big Thank You to the organizers, referees, and contributors who kept the series going over the years! The series emerged from a couple of meetings of research groups in Austria, Japan, and Tunisia, including my Theorema Group at RISC, see the home pages of the SCSS series since 2006. In 2012, we decided to define "Symbolic Computation in Software Science" as the scope for our meetings and to establish them as an open conference series with this title.

As always, when one puts two terms like "symbolic computation" and "software science" together, one is tempted to read the preposition in between - in our case "in" - as just a set-theoretic union. Pragmatically, this is reasonable if one does not want to embark on scrutinizing discussions. However, since I was one of the initiators of the SCSS series, let me take the opportunity to explain the intention behind SC *in* SS in this note. Also, this note, for me, is a kind of revision and summary of thoughts I had over the years on the subject of SCSS and related subjects. Hence, allow me to refer to a couple of my papers with basic considerations on the subject. I do not discuss, however, any of my technical contributions to the subject of SCSS (which would be, mainly, Gröbner bases and the Theorema system). In some way, this note continues, updates, and specializes the note on mathematics in the 21st century I gave at the beginning of SCSS 2013, see [3], from which I quote:

> In my view, mathematics of the 21st century will evolve as a unified body of mathematical logic, abstract structural mathematics, and computer mathematics with no boundaries between the three aspects. Working mathematicians will have to master the three aspects equally well and integrate them into their daily work. More specifically, working in mathematics will proceed on the object level of developing new mathematical content (abstract knowledge and computational methods) and, at the same time, on the meta-level of developing automated reasoning methods for supporting research on the object level. This massage of the mathematical brain by jumping back and forth between the object and the meta-level will guide mathematics onto a new level of sophistication. Symbolic computation is just a

T. Kutsia (Ed.): Symbolic Computation
in Software Science (SCSS'21)
EPTCS 342, 2021, pp. 1–13, doi:10.4204/EPTCS.342.1

way of expressing this general view of mathematics of the 21 st century and it also should be clear that software science is just another way of expressing the algorithmic aspect of this view.

Continuing the discussion on the intended meaning of "Symbolic Computation in Software Science" in this note will hopefully help to advocate the central importance of this topic for the future of mathematics, logic, and computer science. This should also motivate more and more people to submit papers to the conferences in the SCSS series.

What is Symbolic Computation?

In 1984, Academic Press London issued a call for designing a new journal for a new field that had emerged approximately since 1960. Various names were in use for this field: computer algebra, symbolic and algebraic manipulation, analytic computation, formula manipulation, computation in finite terms, symbolic computation, and others. As a response to this call, I submitted a proposal to Academic Press for a "Journal of Symbolic Computation". My proposal was selected and my clarification of the scope of "symbolic computation" formed also the Editorial of the journal, see [2].

I defined "*symbolic computation*" as the area that deals with algorithms on symbolic objects, and I proposed "*symbolic objects*" to be defined as finitary representations of infinite mathematical entities. Here, "*finitary*" means "storable in a computer memory". For example, finitely many generators with finitely many relations between words formed from the generators form a finitary object that may represent an infinite group (or, at least, a "large" group, i.e. a group whose number of elements is much much larger than the size of the finitary representation). Algorithms can only work on finitary objects and the flavor of "symbolic" is exactly the point that we want to solve problems on infinite (or "large") mathematical entities by finding algorithms that work on finitary (small), "symbolic", representations of these entities. Also, numerical computation works on finitary representations (for example, lists of rational numbers that represent a function consisting of infinitely many pairs of real numbers). In this sense, numerical computation is a subfield of symbolic computation. However, usually, for algorithms to be called "symbolic" we request that the representation of the abstract mathematical domains by finitary domains must be an isomorphism w.r.t. to the operations on the objects we study. In numerical computation, for the sake of efficiency, this request has to be given up. Instead, we are satisfied with "approximations".

Pragmatically, in the editorial of the Journal of Symbolic Computation, I named three main areas for symbolic computation: *computer algebra, automated reasoning, and "automatic programming"*. I also emphasized that all aspects of these areas should be in the scope of the Journal of Symbolic Computation: mathematical theory on which symbolic algorithms can be based, the algorithms with their correctness proofs and complexity analysis, the details of the implementation of the algorithms, languages and software systems for symbolic computation, and applications. Indeed, the three main branches of symbolic computation consider three important classes of "symbolic objects":

- *computer algebra*: symbolic objects that represent algebraic entities like terms that represent functions, differential operators, etc. or finite relations that represent residue class structures;
- *automated reasoning*: symbolic objects containing (quantified) variables that are considered as statements on (infinite) domains;
- *automatic programming*: symbolic objects containing variables that are considered as programs that define processes on potentially infinitely many inputs.

(Of course, these three sub-areas of symbolic computation are intimately connected and, in some precise way, even embedded in each other. The distinction between the three areas is more or less only a matter of "flavor".)

In other words, symbolic objects are finitary objects that have "semantics" attached to them where, typically, the semantics is "large", even infinite, not tangible by computers whereas the symbolic objects are "small", finitary, tangible by algorithms. Any field of mathematics can be studied under the "symbolic" view and, actually, in any field of mathematics, if we want to solve problems by algorithms, we have to find finitary representations for the objects in the field. Finding suitable finitary representations, by itself, may be a difficult - sometimes provably impossible - mathematical problem: Before embarking on deeper questions, deciding whether or not two symbolic objects represent the same abstract mathematical object and finding "canonical" representatives for symbolic objects may already be very difficult (sometimes provably impossible). By finding representations of mathematical objects in any field of mathematics, the field becomes "algebraic", and problems in the algebraic disguise of the field, essentially, become combinatorial problems. Thus, very sketchy, one may say: *symbolic computation, ultimately, is the "combinatorization" of all of mathematics via finitary representations of infinite mathematical entities.*

It is a common misunderstanding that symbolic computation is the trivial side of mathematics, i.e. some people believe that, whereas "pure" mathematics lives in difficult spaces needing deep and difficult thinking, algorithmic mathematics (which must be "symbolic" in the above sense) "just" puts everything to the computer and presses the start button. The truth is, that the "just" needs more and deeper mathematics than a mathematics that allows non-algorithmic constructions for problem-solving like the unlimited set quantifier, infinite summation, infinite unions, transition to residua class domains etc. (A trivial example: In "pure" mathematics, a Gröbner basis for given ideal generators can be "easily" found by just taking the ideal generated by the generators. However, the definition of the ideal generated by generators involves an infinite set construction!) Hence, with some provocation, in my view, mathematics only *starts* at the moment when it tries to solve problems by "symbolic computation".

Recently, in 2020, we issued a call for running for the editor-in-chief position of the Journal of Symbolic Computation (JSC). At that occasion, we asked the candidates to submit also their views on the scope of "symbolic computation" and of the JSC. Interestingly, the view of symbolic computation in the editorial of the JSC (and summarized above) was backed by all candidates and, basically, no dramatic changes or extensions were proposed except that "artificial intelligence" was mentioned a couple of times.

Mentioning artificial intelligence, for me, raises some nostalgia because, when I founded the Research Institute for Symbolic Computation (also in 1985), for some time I was torn between using "symbolic computation" or "artificial intelligence" as the main notion in the name of the new institute. At that time, bringing symbolic computation under the umbrella of artificial intelligence was quite tempting and also quite common: For example, finding symbolic integrals was considered an "artificial intelligence" task like playing chess, with lots of heuristics. Correspondingly, the most comprehensive symbolic computation software system at that time, MACSYMA, had "MAC" (= Machine Aided Cognition) in its name! And, of course, implementing heuristics is still a very important approach for improving the practical efficiency of methods for symbolic computation problems. However, in 1985, I deliberately decided against having "artificial intelligence" in the name of my institute since I wanted to emphasize the logical, mathematical, formal approach to problem-solving over the psychological, experimental aspect, which some people (then and now) believe that goes "beyond mathematics". I will go deeper into the analysis of the relationship between symbolic computation and artificial intelligence later in this note.

Anyway, although symbolic computation (in the sense of the editorial of the JSC) seems to be a quite

established and stable notion, as a matter of fact, in the JSC over the years one can observe that

- the majority of papers in the JSC is on computer algebra,
- more and more, but still much fewer, papers are in automated reasoning,
- only a few papers came in on automatic programming.

Symbolic Computation in Software Science

Software science is the science of the process of developing software. This process starts from problems in some "reality" (part of the real world) and creates software that solves the problems in an appropriate finitary model of this reality. Since the beginning of the software age, the software development process has matured from being a kind of "magic" and being an "art" to a decent engineering discipline called "software engineering". In parallel, since the very beginning, people have also tried to establish a "science" of software and the software development process to make the process more reliable, provably correct, faster, more flexible, more economic, and ultimately automatic or semi-automatic. Research in this direction is mostly summarized under the heading "theoretical computer science". Interestingly, the term "software science", which seems quite natural to me, in comparison to "theoretical computer science", is only used quite rarely. (This can easily be verified by googling the two notions and comparing the number of relevant results.)

Anyway, I think that "software science" is a quite useful notion that focuses on the actual development process of software and on its automation and, hence, has a high impact on one of the central technologies - if not the central technology - of our age.

Since the objects of software science are formal models (domains with finitary objects and algorithmic operators on the objects), automation or semi-automation of the software design process is essentially a "symbolic computation" process according to the definition of symbolic computation we considered above. In other words, it should be clear that symbolic computation is the area that naturally should include also the (semi-)automation of the software development process. Unfortunately, this logical analysis did not really create a big stream of papers on automating software development to the JSC (and neither to conferences in the area of symbolic computation like ISSAC, ACA, SYNASC, etc.). Therefore, in 2012, I argued that the topic "*Symbolic Computation in Software Science*" could and should get special attention by turning our group meetings into a conference series with this name.

Still, the idea that symbolic computation should have a major application in software science - in particular in the automation of the software development process - did not create a big echo in the symbolic computation community. Neither do many people who work in software engineering realize that the automation of the software development process is essentially a symbolic computation task. One reason for this is, surely, that there are strong conference series and journals in the area of automated reasoning and related subjects. (A side-remark: As some readers may know, when I built up RISC starting from 1985, I also devoted much of my time to building up the "Softwarepark Hagenberg". With this, I wanted to demonstrate that the mathematically deep field of symbolic computation has also the power to create something with a strong practical impact: I started the Softwarepark with 25 people. When I stepped back as the director of the Softwarepark Hagenberg in 2013, 2500 people working and/or studying in the Softwarepark. I hoped that the government and the administration of my university would have noticed and recognized the unique power that RISC / symbolic computation had created and had turned into innovation in the software foundations, into software development, and into software business. Therefore,

in 2013, I asked the government and university administration to establish an extra professorship "Software Science" in the frame of RISC with the task of continuing my work for directing and expanding the Softwarepark based on solid research on symbolic computation in software science. Indeed, in response to my request and argumentation, a professor position for "Software Science" was created but then, much to my displeasure, giving in to the pressure of the informatics department, the position was finally used for something "more useful" for the education of the informatics undergraduates.)

Now, what I called "automatic programming" in the preface of the JSC, could also be called "symbolic computation in software science". In more detail, I want to make this clear in this note. If seen in the right way, I think that symbolic computation in software science is / could be / should be the most / one of the most fascinating topics of the next stage of mathematics / logic / computer science. (I like to call mathematics, logic, and computer science together "thinking technologies" or just "full-stack mathematics". Unfortunately, "mathematics" sounds old-fashioned to some people, sounds "non-creative" to others, boring to others, intimidating to again others. However, one may bend and turn this as one likes, finally, at the top of the creative hierarchy of problem-solving and gaining knowledge by thinking, there is mathematics at higher and higher levels - whether certain people in politics, science, economy, philosophy, culture, media or the people at the beer table like it or not.)

A Stream of Problems on the Way

On the way from a problem description / a collection of problem descriptions to an algorithm / program / software system that solves the problem there are many creative steps each of which can be handled ad hoc for the particular problem at hand by a mathematician, computer scientist, developer. Each of these steps, however, can also be considered as a problem on the meta-level with some symbolic objects (like software requirements, programs, algorithm schemata, verification conditions, etc.) as input and symbolic objects as output for which we would like to have a general algorithmic solution.

In this section, we assume that all the symbolic objects on the way from a problem specification/requirements to an algorithm / piece of software are expressions that describe or at least try to describe something "in general terms". In particular, we assume that the problem specification (even a vague attempt of a specification that may need much clarification and reformulation) tries to describe the problem in general and not only by examples.

The important case that a problem specification, for certain reasons, can only be given by examples and cannot be explained in general terms, is analyzed in detail in the next section in the paragraph on machine learning.

In the majority of cases, problem specifications are explicit in the sense that they are specified by an expression $P[x,y]$ with input variable(s) x and output variables(s) y, and a solution algorithm A has to satisfy $P[x,A[x]]$, for all x. (However, there are important classes of algorithmic problems that cannot be described in explicit form. For example, a canonical simplifier A for an equivalence relation P cannot be described in this form. More generally, for example, the specification of operations on data structures by axioms or the construction of algorithmic isomorphic representations of mathematical domains is not an explicit specification. We cannot go into more details about this here.)

Depending on the situation, the initial (often vague) problem descriptions may be given in natural language, maybe mixed with drawings and diagrams, or in some formal language.

Also, it is important to distinguish between two extremes:

- Finding algorithms for fundamental, non-trivial, *stand-alone algorithmic problems*: In this case, the problem specification and the solution algorithm are completely formal, symbolic objects and

everything that happens between problem and solution should be amenable to algorithmic treatment on the meta-level, i.e. to symbolic computation. For such problems, typically, time and memory complexity is an issue. Examples: the problem of finding shortest paths in graphs; the problem of finding symbolic integrals; the problem of finding Gröbner bases; etc.

- Developing software for *an entire application*: In this case, the individual parts of the system (called "units") should implement a (big) number of functionalities, most of which are not really difficult. Only some of the functionalities may involve the algorithmic solution of fundamental problems. The algorithms for these functionalities, typically, are known and taken from reliable sources. The complexity of such systems originates from the huge number of units and the various (desired and undesired) interactions of the units. Also tuning of the known algorithms to the application at hand is an issue.

This distinction is important for the following reason: The application of formal methods for establishing the correctness of software only makes sense if we consider non-trivial algorithmic problems. In contrast, for most of the millions of units to be developed in large software systems a formal specification of the problem to be solved by the unit would be essentially identical to the code to be developed. In other words, a proposal for the code of a unit, in the case of "easy" problems, is a way for describing the problem to be solved. This is the reason why rapid prototyping and agile software development, in such situations, is so useful. It is also the reason why formal algorithm verification methods are rarely used in the practice of developing large software systems.

Example. In a calendar software system, probably, we want one unit that should check whether a proposed new calendar entry collides with one of the existing entries. Let us assume that a calendar entry is characterized by its start time and end time. The input to the unit will then consist of four time moments s_1, e_1, s_2, e_2 for the start time and the end time of the first and the second entry, respectively, with input condition $s_1 < e_1$ and $s_2 < e_2$. "After some thinking", the problem will then be described by most developers by a sentence like this: "The two calendar entries characterized by s_1, e_1, s_2, e_2 collide iff $s_2 \leq s_1 \leq e_2$ or $s_1 \leq s_2 \leq e_1$." Now it is clear that this "specification" of the problem is, basically, already the solution algorithm. Only some transformation into the syntax of the programming language used is necessary. No powerful algorithm verification method or algorithm synthesis method is necessary in such a case.

As simple as the example is, it is not too simple to guarantee the avoidance of severe flaws in the development. I tested the example out by presenting it to various (reasonably experienced) developers. Amazingly, a few came up with the following specification / code: "The two calendar entries characterized by s_1, e_1, s_2, e_2 collide iff either $s_2 \leq s_1 \leq e_2$ or $s_2 \leq e_1 \leq e_2$." This specification is "incorrect" because it does not include the case $s_1 < s_2 < e_2 < e_1$, which of course "everybody" would also consider as a collision, even a "particularly heavy one". (I put "incorrect" in quotation marks because, at the very first stage of uttering a request, the "customer is always right". Maybe, he really wants what he tells! Either one just implements what he tells or one may consider the subsequent discussion as a way to find out what he "really wants" or to change his mind about what he wants.) This shows that already in the "thinking" between a vague indication of a problem and its specification (by a general statement, not only by examples), severe mistakes may be made (or, considered differently, the request of the customer may undergo serious changes). In our example, we also could start a little "earlier" and just say: "The two calendar entries characterized by s_1, e_1, s_2, e_2 collide iff the time interval $[s_1, e_1]$ intersects with the time interval $[s_2, e_2]$." Now we could question the notion "intersects" and might agree on the following: "The two calendar entries characterized by s_1, e_1, s_2, e_2 collide iff there is a time moment x such that

$s_1 \leq x \leq e_1$ and $s_2 \leq x \leq e_2$." In this form, we can send the condition into a quantifier elimination algorithm and we will get an answer which will be equivalent to "$s_2 \leq s_1 \leq e_2$ or $s_1 \leq s_2 \leq e_1$." (Please try it out, it is worthwhile!) Hence, this simple example shows that, actually, already in the very early stage of discussing and clarifying even seemingly simple requirements a lot of systematic/formal thinking is involved, which in principle should be amenable to automating and, hence, symbolic computation!

Thus, we start at the very early stage of having a vague desire of achieving something by software and go through all the stages of developing a piece of software that fulfills the desire and, further, through all the stages of maintaining, updating, improving, and integrating pieces of software to fulfill more and more sophisticated desires. Through all these stages, we ask ourselves how much of this process can be (semi-)automated. This gives a rich list of R&D topics, which make up *the important topics in the scope of "symbolic computation in software science" as described in the calls for the SCSS series*, see the latest version in the call for SCSS 2021. This call contains, roughly, 20 important and quite diverse but strongly interconnected topics on the way from requirements to software.

I do not list these topics and comment on all these topics here. Rather, let me give some personal remarks that emphasize, and maybe expand, some of the subjects, themes, and objectives of the topics in the SCSS calls.

- My feeling is that relatively little research is available on (semi-)automating the development of *large software systems consisting of tons of simple "units"* of the type we have seen above in the example. Research has focused more on symbolic methods for algorithm verification and synthesis for non-trivial algorithms. In some way, this is unfortunate because the construction of tons of software is necessary today, semi-automation of this process is needed and could be a big business. Our research results are too much oriented on automating the invention of "important", "difficult" algorithms. However, the (semi-) automation of the development of huge amounts of simple programs and their interaction, in some way, is quite challenging, much needed, and asks for formal methods to guarantee the quality of the process.
- As a variant of developing big software systems consisting of many simple units we also should consider the task of re-engineering big software systems that were written years ago in programming languages that are antiquated now. Often, the documentation of such systems is lost or fragmentary, and finding out what the units should do, i.e. getting a problem specification from code, is a major task.
- In most cases, software development starts from vague requirements in *natural language (maybe with diagrams or drawings).* The task is to come up, maybe in an interactive dialogue, with a bunch of formal requirements. Here, we should allow natural language or, maybe, a simplified version of natural language as a symbolic language: The sentences that formulate requirements are "finitary" with infinite semantics since, normally, requirements have hidden universal quantifiers in it. (See the simple example above: The requirement is formulated for arbitrary calendar entries. In our first step towards formalization, the hidden universal quantifier goes over s_1, e_1, s_2, e_2.) We also should allow diagrams or drawings as symbolic objects: They are surely "finitary" and, usually, have infinite semantics, since a drawing normally tries to convey the important features of infinitely many possible individual situations. Specifying requirements by natural language text or drawings is very different from the specification of requirements by finitely many input/output pairs, see the analysis of "machine learning" in the next section. *Allowing natural language or drawings in requirements* is of course a big challenge but I think we should take this deliberately under the umbrella of SCSS because it will need much more than just ordinary NLP (natural language processing) and graphics. Rather, a systematic connection to logic is necessary. (In fact,

in dynamic geometry systems a lot of work in this direction is already done when graphical input explaining geometrical situations "in general positions" is allowed.)

- As we have seen in the simple example above, we also would like to go a step further and go from requirements in natural language and/or drawings right away to algorithms/programs that satisfy the requirements. As we have argued in the example, in the majority of "units" in application software systems this will not be significantly more difficult than coming up with formal requirements.
- The individual algorithms/programs in software systems do not live in empty air but inside a whole hierarchy of data structures and domains which, depending on the context, are called (algorithmic) "models" of reality. Such models consist of problem specifications, definitions of notions, knowledge, algorithms, and - in the ideal case - arguments/proofs why the operations/algorithms in the system meet their specifications. Hence, seen in this way, software systems can also be considered mathematical knowledge systems. Hence, (semi-)automation of building and maintaining such systems can also be seen under the umbrella of *Mathematical Knowledge Management (MKM).* We introduced this term a couple of years ago in a slightly different context, see the preface of the proceedings [4], which were expanded as the special issue [5]. We propose that SCSS and MKM should be considered together and, maybe, SCSS and MKM should be collocated in the future.
- In practice, the correctness of software is established by testing rather than by formal verification. *Testing* is a highly developed "technology" in software engineering: There is an arsenal of "automated software testing" systems available. They are well integrated into the various software development environments and they are quite helpful for managing large test suites for the consecutive versions of software systems. However, I think that much more could be done by applying formal methods for coming up with complete systems of test data from a given problem specification and program. Here, completeness means that we would get one test input/output for each equivalence class of inputs that generate the same program path during execution. Of course, in general, the set of these equivalence classes is not finite. However, in the practical case of large software systems consisting of a huge number of relatively simple units, the set of equivalence may well be finite, see the example above. As can be seen in the example, generating a complete system of equivalence classes for inputs might be essentially the same task as coming up with the code for the program. In fact, this automated generation of equivalence classes should start from the problem specification and *not* from a program code - as most of the commercial "white box" test generation programs do.

How Does Artificial Intelligence Fit into the Picture?

Undoubtedly, in the past two decades, artificial intelligence has gained enormous attention. This is due to the fact that, by the drastically increased computational power of current computer systems and the availability of huge databases of "labeled" data, a couple of difficult and urgent problems have received impressive solutions by artificial intelligence methods, as for example machine translation of natural languages.

Amazingly, there is still a lot of mystery, nebulosity, and misunderstanding around what artificial intelligence (AI) actually is and why it is / may be / is believed to be essentially different from all computational approaches so far. This nebulosity is all over the place: in politics, in the media, even in science, and, of course, with the man on the street. At times, I have the impression that even quite some

researchers in the AI area do not have a very clear picture of the distinctive characteristics of AI when compared with other computational approaches. Also, labeling a project with AI, may have a beneficial effect when it comes to funding, societal respect, political influence etc. Thus, it is tempting to keep the notion ambiguous. What amazes me, even more, is that the nebulosity about the essence of AI did not disappear since the field started in the middle of the fifties. I remember talks of AI evangelists around 1980, i.e. in the "first wave of AI research", who believed and spread that "AI can solve hard problems that cannot be solved by mathematics". And still, when I participate in political discussions about the importance of mathematical education (in the sense of training mathematical thinking), I hear the argument that, actually, the ability to do mathematics will be less and less important because "tedious" mathematical thinking, in the presence of "artificial intelligences" (plural!), will not be necessary anymore and that we should teach the youngsters more "creative" things than mathematics.

Now all such statements may be true or false according to which notion of artificial intelligence one has in mind. For clarifying this notion, I want to distinguish three possible characterizations of AI:

Hard Problems: Artificial Intelligence may be described as the field that tries to solve problems that, at a certain historic moment, are considered to be "hard" in the sense that they apparently need a decent amount of (human) "intelligence" to solve them. For example, playing chess or finding symbolic integrals, at some historic moment, were considered as needing human intelligence. Algorithms (invented by humans!) that finally were able to solve these problems were then (and still are) considered to be the result of "AI research".

Now, in my opinion, this definition of the notion of AI is quite shallow. It is the natural flow of science and technology that we can solve harder and harder problems automatically, i.e. by algorithms. However, from some point on, people think that now "algorithms are taking over", "artificial intelligence is replacing humans" etc. forgetting that this happened and happens already since centuries and that this is the very goal of science and technology. And, of course, whatever the methods behind automation were and are, we humans should stay in control and decide how far we let problems be solved and decisions be taken by algorithms. Anyway, the notion of a "hard" problem is relative and "hard" problems for which an algorithmic solution was finally found very soon are considered to be "easy" by the consumer. For example, car drivers nowadays take the functionality of a navigation system for granted. Some thirty years ago, the current functionality of navigation systems would have been considered unbelievably intelligent. In fact, the stack of scientific findings and algorithmic techniques involved in a navigation system for guiding a driver from A to B is quite deep.

In my opinion, one should not use the notion of "artificial intelligence" for "finding algorithms for hard problems" but rather continue to call this just "mathematical, algorithmic solution of hard problems". Attaching the label "AI" to algorithms depending on whether they solve hard or easy problems is more a question of marketing rather than a logically sound distinction.

Simulate the Brain: A completely different view (and branch) of artificial intelligence is artificial intelligence as the science of understanding and simulating biological structures that show "intelligence", notably the human brain. This type of AI research, historically, was one of the origins of the field of AI that started, maybe, 1943 with the investigations of W. McCulloch and W. Pitts who introduced a simple mathematical model of the functionality of a neuron. Of course, understanding and simulating the most complex biological systems, which are commonly considered to display "intelligence", is a highly fascinating and relevant undertaking. Well, why not call this type of research "artificial intelligence" in the same way as a technical realization of the phenomenon of flying could be called "artificial flying".

"Artificial intelligence" in the sense of brain simulation has little overlap with symbolic computation in software science except that, of course, there may be applications of symbolic computation in developing models of the brain. Also, studying biological structures (like the brain, like swarms of animals, or like the evolution of life on earth) motivated some of the algorithmic methods that today are called "AI methods", see next paragraphs.

"Intelligent" Methods: The third approach of characterizing artificial intelligence is by specifying certain algorithmic methods as "intelligent". These algorithms would constitute the area of "artificial intelligence". I hope I do not overlook something important but my impression is that, essentially, "machine learning" is the only such method or, better, class of methods that has not already been around before the term "artificial intelligence" was coined. The many other algorithmic methods that are often labeled as "AI methods", like automated reasoning, semantic networks, graph search, expert systems, regression, etc., in my view, are algorithmic methods that are not specific to AI. They are, so to say, usual algorithmic methods and were applied also to problems that, for some reason, got the label "AI", see the remarks about hard problems above.

In my view, machine learning methods cannot actually be specified by the way how they work but, rather, by the way the problems these methods should solve are specified. As we have seen in the previous section, the fundamental part of algorithm and software development is the transition from a given problem specification P to an algorithm (program, system) A that solves the problem for any admissible input. As long as the steps for going from a problem specification to a solution algorithm are done by a human this is just the "usual" business of mathematics/informatics. If finding these steps is (partially) supported by algorithms (invented by humans) this is what we can call "symbolic computation in software science". How and when does "machine learning" come in and why, if at all, is this different from "usual" mathematics and "usual" (maybe quite sophisticated) symbolic computation in software science?

The point is that, in many situations, when we want to specify a problem, we do not have a specification "in general terms" available. For example, let's consider the seemingly simple problem of deciding whether a given English sentence contains information of the type "somebody cooperates with somebody else". An algorithm for this problem should produce the answer "NO" in case no such information is in the input sentence and should produce "YES" and the two "somebodies" if such information is in the sentence. Now, of course, before trying to invent such an algorithm, we will ask: What exactly do you mean by "cooperate"? Among the English speaking community, under the natural assumption of a long experience of using English in thousands of different situations, it would be natural the start to explain "cooperate" in terms of a couple of other notions like "working together", "having a common goal", ... Oh, "having a common goal" may not always be sufficient for speaking about "cooperation". One may have a common goal but fight against each other. Thus, "supporting each other" etc. should be added. Some more subtle details should be explained, some other things excluded etc. A long list of sentences explaining the meaning of "cooperate" would be necessary. Then one could, in the attempt of finding an algorithm for this little problem, try to put these numerous explanations into algorithmic rules (assuming that we already have access to a powerful grammar parsing algorithm for all of English). As a result, we would hope that this rule system would be able to do the job. For example, if we now would input "Peter and Ann found a way to help each other for passing the exam", the algorithm should answer "YES", "Peter", "Ann". If we would input "Peter and Ann passed the exam on the same day", it should answer "NO". Should it really answer "NO"? Shouldn't it rather answer "DON'T KNOW" or "COULD BE" or "COULD BE BUT NOT EXPLICITLY MENTIONED".

I now want to explain what, in my view, is the essence of the machine learning approach. For this,

we need not at all bother about what "learning" is. I just consider those methods that, over the years, have been named "machine learning" methods. The common feature of these methods is not how they proceed but the type of specification of the problems to which they are applied: Namely, they all are applied to problems of the kind above where a spelled-out complete specification is not possible or, at least, not feasible. Now, the fundamental idea of machine learning for solving such problems is:

- Instead of spending time trying to specify the problem by a huge number of general definitions, cases, rules, etc., one spends the time giving a huge number of examples of input instances together with the answers. (In this paper, we consider only "supervised learning".) In this context, the answers are called "labels".
- One sets up an algorithm from a certain class of relatively simply structured algorithms (like the class of neural networks, the class of hyperplanes in a high-dimensional space, the class of nested if-then-else expressions, etc.) with some constants $c_1, \ldots, c_n$ (for example the weights at the inputs of neurons in neural networks) in the algorithm left unspecified. For each choice of numerical values for the $c_1, \ldots, c_n$, the algorithm would produce an answer for each admissible input for the problem.
- One uses techniques of mathematical optimization (or other, experimental techniques, for example techniques that mimic biological evolution) to change the initial values for $c_1, \ldots, c_n$ iteratively until the answer of the algorithm to more and more inputs from the set of labeled data would give the answer specified by the label. In the jargon of machine learning, this iteration is called "training a model".
- One stops the iteration on the $c_1, \ldots, c_n$ when sufficiently many answers are identical to the labels. Practically, at the beginning of the whole operation, one partitions the set of labeled input into a "training set" that is used for the iterations and a "test set" on which the algorithm with the current values for the $c_1, \ldots, c_n$ - which in the jargon of machine learning is called the "trained model" - is tested.

The impressive success of this approach in the past two decades hinges on three ingredients:

- a huge amount of *mathematical* research on good and, partly provably convergent, techniques for improving the algorithm parameters $c_1, \ldots, c_n$; such research was partly already available in the first phase of AI between 1960 and 1980, but it did not convince because of the next two ingredients were not available,
- huge corpora of labeled data; for example, in the spectacular application of machine translation, a huge amount of "labeled data" is now available in the form of files that contain an original book and its translation - by humans - to some other language,
- high-performance computing; the number of iterations of the machine learning steps for determining suitable $c_1, \ldots, c_n$ and the computational effort in each step is huge and is only manageable by computers in recent years.

In principle, the approach is not radically new. Examples of historical "learning from examples" problems are: Given points in the plane, find the coefficients $c_1, \ldots, c_n$ of a polynomial that goes through the given points (the interpolation problem). Given a function with some properties on differentiability, an interval, and a distance, find the coefficients $c_1, \ldots, c_n$ of a polynomial that is closer to the function than the given distance everywhere on the interval (approximation problem). Given points in the plane, find the coefficients of a straight line such that the distance to all points is minimal (regression problem).

Given a function with certain differentiability properties, find the coefficients $c_1, \ldots, c_n$ of a finite Fourier approximates of the function. Etc.

Artificial Intelligence in the form of machine learning falls neatly into the "automatic programming" view: It is the method of choice in cases where the problem is not specified by general (formal or natural language) statements but, rather, is specified only by a (huge) number of examples of admissible input and desired output. In the case of general specifications of problems, the transition from the problem to a solving algorithm, in principle, is a reasoning process that is executed by humans or, in the symbolic computation approach advocated in this paper, is a reasoning process (partly) executed by symbolic computation methods. In the case of problems that are specified only by examples, this process can still be automated by the machine learning approach.

From the simple summary of the machine learning approach I gave above, one important deficiency of the machine learning approach should be clear: The algorithm which we get for a given problem just does the job of delivering (in sufficiently many cases) desired answers. However, in general, no reason can be given why, for example, the particular neural network that translates one natural language to the other mimics certain fundamental insights about the environment both languages share as their semantics. This is, in fact, similar to the situation in the historical predecessors of "learning from examples": The Fourier analysis just does the job of finding an optimal Fourier sum. In the example, where the function to be represented is the frequency spectrum of a musical tone, the representation by a finite Fourier sum has a reasonable "explanatory" power: The tone is composed of tones and overtones that occur in the physical "music" world (for example, when picking the strings of a guitar). However, if a Fourier representation of some arbitrary other function is executed, there will be some outcome but there may not be any reasonable interpretation of what this representation means in the reality from which the function is taken.

The problem of weak explanatory power in the models (algorithms) created in machine learning is well known. Lots of research was recently started to extract "meaning" from such models. This research area is called "explainable AI".

In the frame of the analysis of this paper, I summarize: The machine learning approach can be well subsumed under the general target of (semi-)automating the process of software development ("automatic programming"). It can be viewed as a numerical, rather than a symbolic, approach to automatic programming. Thus, it is probably a very good idea to integrate machine learning into the scope of the SCSS series because, of course, the interaction of symbolic and numerical computation as the two possible approaches to compute on finitary representations of abstract mathematical domains is of utmost importance. The integration of machine learning into the scope of SCSS can generate a stream of new ideas in both directions: Applying symbolic methods to mathematical sub-problems of machine learning (e.g. the determination of weights in neural networks) and applying machine learning to symbolic algorithms (e.g. "learning" a priori complexity estimates for computation-intensive methods like Gröbner bases etc.).

However, there is no reason to establish a flavor of "intelligence beyond mathematics" when speaking about machine learning: I hope I have been able to show the machine learning is just another mathematical method. As in the past, of course, we can hope and expect also for the future that more and more powerful algorithmic problem-solving methods will be invented.

Personally, when speaking to people who do not (want to) understand the timeless, universal, always new, creative power of mathematics, I like to use the term *algorithmic intelligence* for what we are doing: Algorithmic intelligence is the *human* intelligence that produces algorithms for more and more challenging problems in all areas of human activity. By an algorithm, an infinite class of individual problem instances can then be treated by a completely *unintelligent* machine. People who do not really

understand what is going on may *believe that these machines display "intelligence"*. The algorithmic intelligence - by reflection, i.e. jumps to higher and higher meta-levels - also provides more and more sophisticated algorithms for producing algorithms. The incompleteness theorem of Kurt Gödel (1931), in a somewhat liberal interpretation, shows that this tour through higher and higher levels of algorithmization has no upper bound. In comparison to "artificial intelligence", the term "algorithmic intelligence" is used quite rarely, which can be verified by Googling. However, my impression is that "algorithmic intelligence" appears in more serious discussions about the essence of AI. Therefore, I like to expand the abbreviation "AI" as "algorithmic intelligence".

Implicitly, I expressed this view already in the early days of AI, see [1]. At the "Spring School of AI" in Teisendorf, 1982, I contributed a long article summarizing the most important "symbolic" methods for automating the algorithm/software development process that were known at that time. And we had long, intensive, and quite controversial discussions at this conference on the question of whether AI is something that goes beyond mathematics. As you may guess, my answer then was "no" with essentially the arguments which I expanded above. In my hectic years of research on methods for "algorithmic intelligence" and research management, I never found the time and occasion to spell out these arguments in a paper. Thus, I am grateful that I am given the opportunity here.

References

[1] Bruno Buchberger (1982): *Computer-unterstützter Algorithmenentwurf (Computer-Aided Algorithm Design)*. In Wolfgang Bibel & Jörg H. Siekmann, editors: *Proceedings of the Frühjahrsschule Künstliche Intelligenz (Spring School in Artificial Intelligence), Teisendorf, Germany, 15.-24. März 1982*, *Informatik-Fachberichte* 59, Springer, pp. 141–201, doi:10.1007/978-3-642-68828-7_4.

[2] Bruno Buchberger (1985): *Symbolic Computation (An Editorial)*. *Journal of Symbolic Computation* 1(1), pp. 1–6, doi:10.1016/S0747-7171(85)80025-0.

[3] Bruno Buchberger (2013): *Mathematics of the 21st Century: A Personal View*. In Laura Kovács & Temur Kutsia, editors: *Proceedings of the Fifth International Symposium on Symbolic Computation in Software Science (SCSS 2013)*, *RISC Report Series* 13-06, Johannes Kepler University, Linz/Hagenberg, Austria, p. 1. Available at https://www.risc.jku.at/publications/download/risc_4737/TR_13_06_SCSS2013.pdf. (See also the link to the slides of this talk on the website of SCSS 2013 at https://www.risc.jku.at/conferences/scss2013/program.html.).

[4] Bruno Buchberger & Olga Caprotti, editors (2001): *Electronic Proceedings of the First International Workshop on Mathematical Knowledge Management (MKM 2001)*. RISC, Johannes Kepler University, Linz/Hagenberg, Austria. Available at https://www.risc.jku.at/conferences/MKM2001/Proceedings/.

[5] Bruno Buchberger, Gaston H. Gonnet & Michiel Hazewinkel (2003): *Mathematical Knowledge Management. Special issue of the Annals of Mathematics and Artificial Intelligence* 38(1-3), pp. 1–2, doi:10.1023/A:1022900528196.

ArGoT: A Glossary of Terms extracted from the arXiv

Luis Berlioz
University of Pittsburgh
Pennsylvania, USA
lab232@pitt.edu

We introduce ArGoT, a data set of mathematical terms extracted from the articles hosted on the arXiv website. A term is any mathematical concept defined in an article. Using labels in the article's source code and examples from other popular math websites, we mine all the terms in the arXiv data and compile a comprehensive vocabulary of mathematical terms. Each term can be then organized in a dependency graph by using the term's definitions and the arXiv's metadata. Using both hyperbolic and standard word embeddings, we demonstrate how this structure is reflected in the text's vector representation and how they capture relations of entailment in mathematical concepts. This data set is part of an ongoing effort to align natural mathematical text with existing Interactive Theorem Prover Libraries (ITPs) of formally verified statements.

1 Introduction and Motivation

Mathematical writing usually adheres to strict conventions of rigor and consistent usage of terminology. New concepts are usually introduced in characteristically worded definitions (with patterns like *if and only if* or *we say a group is abelian...*). This feature can be used to train language models to detect if a term is defined in a text. Using this, we have created ArGoT (**ar**Xiv **G**lossary **o**f **T**erms), a silver standard data set of terms defined in the Mathematical articles of the arXiv website. We showcase several interesting applications of this data. The data set includes the articles and paragraph number in which each term appears. By using article metadata, we show that this can be an effective way of assigning an arXiv mathematical category[1] to each term. Another application is to join the terms with more than one word into a single token. These phrases usually represent important mathematical concepts with a specific meaning. We show how standard word embedding models like word2vec [13] and GloVe [16] capture this by embedding phrases instead of individual words. Even more, the word-vector can be used to predict which mathematical field the term belongs to, and hypernimity relations.

All these properties makes ArGoT a data set that will be of interest to the broader NLP research community by providing abundant examples for automated reasoning and NLU systems. Our main objective is to organize a comprehensive dependency graph of mathematical concepts that can be aligned with existing libraries of formalized mathematics like mathlib.[2] The data is downloadable from https://sigmathling.kwarc.info/resources/argot-dataset-2021/ and the all the code that went into producing it is in: https://github.com/lab156/arxivDownload

This data set was created as part of the Formal Abstracts project. Our group has benefited from a grant from the Sloan Foundation (G-2018-10067) and from the computing resources startup allocation #TG-DMS190028 and #TG-DMS200030 on the Bridges-2 supercomputer at the Pittsburgh Supercomputing Center (PSC).

[1] arXiv's categories within mathematics: https://arxiv.org/archive/math

[2] https://github.com/leanprover-community/mathlib

T. Kutsia (Ed.): Symbolic Computation
in Software Science (SCSS'21)
EPTCS 342, 2021, pp. 14–21, doi:10.4204/EPTCS.342.2

Term	Count
lie algebra	20524
hilbert space	16881
function	14920
banach space	14461
metric space	12882
_inline_math_-module	12731
topological space	12518
disjoint union	11436
vector space	11337
simplicial complex	10943

Table 1: Most common multiword entries in the data base.

Classification Task			
Method	**Precision**	**Recall**	**F1**
SGD-SVM	0.88	0.87	0.87
Conv1D	0.92	0.92	0.92
BiLSTM	0.93	0.93	0.93
NER Task			
ChunkParse	0.32	0.68	0.43
LSTM-CRF	0.69	0.65	0.67

Table 2: Training metrics on the classification and NER tasks.

2 Description of the Term-Definition Extraction Method

In [2, 9], the authors describe the method used to obtain the training data for a text classification model that identifies definitions and the Named Entity Recognition (NER) model that identifies the term being defined.

The classification task consists of training a binary classifier to determine whether a paragraph is a definition or not. We use the `\begin{definition}...\end{definition}` in the article's LaTeX source to identify true examples. To gather non-definitions, we randomly sample paragraphs out of the same articles. The source of the training data is the LaTeX source code of the articles available from the arXiv website. A total of 1,552,268 paragraphs labeled as definitions or non-definitions were produced for training. It was split as follows: 80% training 10% testing and 10% validation. This data was used to train three different and common classification models:

- The Stochastic Gradient Descent with Support Vector Machines (SGD-SVM).
- The one-dimensional convolutions (Conv1D) neural network.
- And Bidirectional LSTM (BiLSTM).

For the first method, we used the implementation distributed with scikit-learn library [15]. The last two were implemented in Tensorflow. Table 2 shows the most common metrics of performance for each method.

The definitions are then fed into a NER model to identify the term being defined in them. The data used to train the NER model comes from the Wikipedia English dump[3] and several mathematical websites like PlanetMath[4] and The Stacks Project.[5]

We tested two different implementations of the NER system, the first is the *ChunkParse* algorithm available from the NLTK library [3]. The second is a time-distributed LSTM (LSTM-CRF) [11]. Both architectures use a similar set of features that in addition to the words that form the text, detect if the word is capitalized, its part-of-speech (POS) and parses punctuation e.g. to tell if a period is part of an abbreviation or an end of line. To compare the two implementations, we used the `ChunkScore` method in the NLTK library [3]. The results appear in Table 2.

[3] `https://dumps.wikimedia.org/`
[4] `https://planetmath.org/`
[5] `https://stacks.math.columbia.edu/`

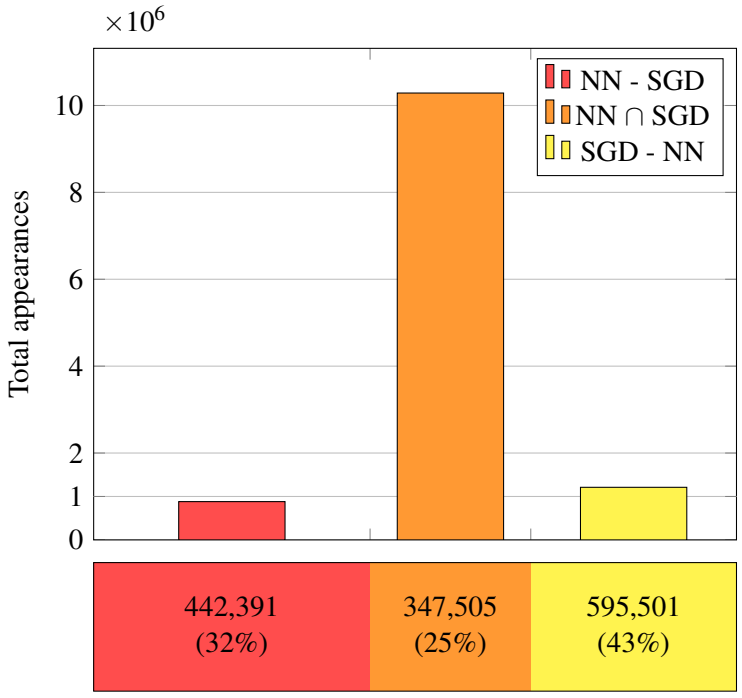


Figure 1: Comparison of the two glossaries. The bar graph on top counts the total appearances of a term in both the NN and SGD glossaries. The bottom compares the relative sizes of the NN-only, intersection, and SGD-only distinct terms.

We have compiled two different and independent glossaries by running the algorithm through all of the arXiv's mathematical content. The first one is based on neural networks (NN), it uses LSTM for both the classification and NER tasks. In contrast, the second one combines the SGD and ChunkParser method to provide a completely independent approach to the previous model.

It is interesting to compare the results obtained using the two models. For the classification task, we have observed Cohen's kappa (κ) inter-rater agreement of 93% between the results produced by the two methods. This corresponds to a high degree of agreement between the two classifiers [4].

As for the final results, Figure 1 compares the two glossaries by counting the number of times a term appears in either glossary, and the number of distinct terms. The results point to a high consistency of the two systems on a relatively small set of 350,000 terms.

Table 1 lists some of the most frequently found terms in the data set.

2.1 Format and Design of the Data Set

The `ArGoT` data set is distributed in the form of compressed XML files that follow the same naming convention the arXiv's bulk download distribution.[6] For instance, Table 3 shows a sample entry in the fifth file corresponding to July, 2014. The definition's statement and terms (definiendum) are specified in the `stmnt` and `dfndum` tags respectively and the paragraph `index` is specified as an attribute of the `definition` tag.

[6] arXiv Bulk Data Access: `https://arxiv.org/help/bulk_data`

```
<article name="1407_005/1407.2218/1407.2218.xml" num="89">
<definition index="51">
    <stmnt> Assume _inline_math_. We define the following space-time
    norm if _inline_math_ is a time interval _display_math_ </stmnt>
    <dfndum>space-time norm</dfndum>
</definition>
</article>
```

Table 3: Example of an entry in the term's data set. The statement of the definition is contained in the <stmnt> tag. The terms (definiendum) are listed as <dfndum> tags. Each entry contains all the information to recover, article's name and paragraph's position.

Category:	Count	Category:	Count
math.FA	5922	math.GN	108
math.AP	2045	math.RT	85
math.PR	1022	math.SG	77
math.DS	833	math.GT	76
math.OA	595	math.CO	61
math.CA	535	math.ST	61
math.DG	483	math.KT	50
math-ph	466	math.GM	48
math.OC	398	math.AG	35
math.CV	304	math.RA	33
math.NA	275	math.HO	32
math.GR	226	math.CT	23
math.MG	173	math.AT	15
math.LO	168	math.QA	10
math.SP	163	math.AC	8
math.NT	131		

Table 4: Category profile for the term: *Banach Space*. The codes are part of the metadata for each arXiv article.

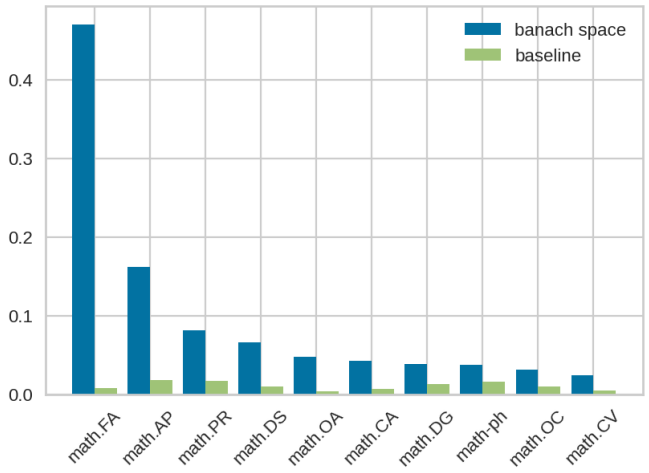


Figure 2: Comparison between the term's category distribution and baseline distribution. Only categories with the highest values for the term are shown.

3 Augmenting Terms with arXiv's Metadata

Each mathematical article in the arXiv is classified in one or more *categories* by the author at the time of submission. Categories include `math.FA` and `math.PR` which stand for Functional Analysis and Probability respectively. The full list is available at `https://arxiv.org/archive/math`. This is part of the arXiv's metadata and also records information like the list of authors, math subject classification (MSC) codes, date of submission, etc.

By counting the categories in which a certain term is used, we get an idea of the subjects that it belongs to. In Table 4, we see the category profile of a very common term. Since the number of articles in each category varies significantly, we also take into account the baseline distribution, that is, the ratio of articles in each category to the total number of articles. Hence, it is possible to give an empirical score of a term's pertinence to a certain category by comparing its category profile with the baseline distribution. In order to measure how much of an outlier a term is to the baseline distribution, we use the KL-divergence:

$$D_{\mathrm{KL}}(P\|Q) = \sum_{x\in X} P(x)\log(P(x)/Q(x)),$$

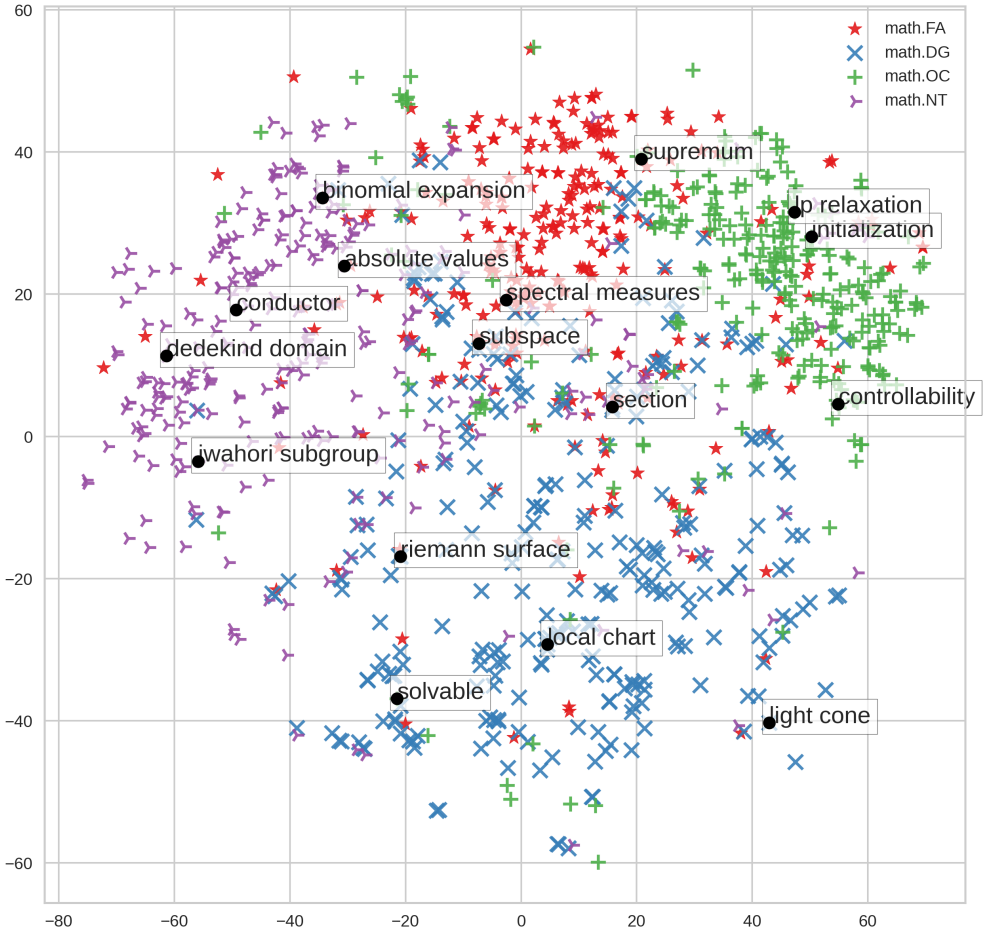


Figure 3: t-SNE visualization of the word vectors of selected terms in the data set. The terms are selected to be specific to the four categories in the picture. Points with a label are selected at random.

where P and Q are the probability distributions of the term and the baseline respectively. And, X is the set of all the categories.

The next step is to generate word embeddings. To prepare for this, we modify the text by joining multiword terms in ArGoT to produce individuals tokens. After normalizing the text, i.e. converting to lowercase and removing punctuation and special characters; the result is a large amount of text that is ready to be consumed by either the word2vec or GloVe algorithms. In Figure 3, we observe a t-SNE (t-distributed stochastic neighbor embedding) visualization of a word2vec model produced this way. In this image, each term is assigned its most frequent category. Notice that even though the ArGoT data set has no access to the arXiv categories, the vectors in the same category cluster together. We consider this as a strong indication of alignment between clusters and categories.

4 Using Hyperbolic Word Embeddings to Extract Hypernymy Relations

It is natural to want to organize mathematical concepts into taxonomies of various sorts. For instance, the SMGloM project [8] introduced a rich standard for mathematical ontologies. Another approach aims

to create a semantic hierarchy of concepts such that for a given term we can enumerate all its hypernyms [18].

This can be achieved by counting the co-occurrence [10] of terms in definitions. This approach has certain drawbacks, for instance, it relies on co-occurrence examples for each pair of terms, this ends up producing an abundance of disconnected (i.e. not co-occurring) terms [1].

Another possibility, involves the use of *hyperbolic word embeddings*, in this setting the hypernimity relation becomes a geometric vector in hyperbolic space. This implies that every two terms in the embedding can be compared by using the hyperbolic metric. This type of word embeddings is known to outperform euclidean models in the representation of hierarchical structures [14].

We used PoincareGlove [17] to create hyperbolic word embeddings. This algorithm modifies the GloVe *euclidean* objective function to use a hyperbolic metric instead. In addition to the same text input as word2vec and GloVe, this model requires a small set of examples in order to interpret the embedding. For general purpose English text, WordNet [6] is the standard choice. In WordNet, every entry is assigned an integer level in a hypernymy hierarchy (this is the `max_depth` attribute of the NLTK's WordNet API).[7]

To generate something analogous to WordNet levels for mathematical content, we opted for the PlanetMath data set. This is due to its relatively small size, broad coverage of mathematical knowledge and independence of the arXiv data. Given two term-definition pairs (t_1, D_1) and (t_2, D_2), we say that term t_2 *depends* on the term t_1 if D_2 contains t_1. For small sets of term-definition pairs with no interdependence, this simple criterion is enough to create a directed graph (V, E) where V is the set of all the terms and E is the set of all the dependency relations. To assign a level $\lambda(v)$ to every vertex $v \in V$, solve the following integer linear program:

$$\min \sum_{(v,w)\in E} \lambda(w) - \lambda(v), \quad \text{such that} \quad \lambda(w) - \lambda(v) \geq 1 \quad \forall (v,w) \in E.$$

This linear model appears in [7] as a subtask of a directed graph drawing algorithm. There, it is used to estimate the ideal number of levels to draw a directed graph.

Table 5 shows the nearest neighbors of four different terms. The neighbors are found using the Euclidean distance. The terms are sorted in order of the average value of their y-coordinates (which in the upper-half plane model represents the variance of the underlying Gaussian distribution). This is referred to as the `IS-A` rating.

5 Conclusions and Further Work

We introduced `ArGoT`, an comprehensive glossary of mathematics automatically collected from the mathematical content on the arXiv website. Essentially, it is set of term-definition pairs, where each pair can be contextualized in a large semantic network of mathematical knowledge, i.e., dependency graph. We also showed how this network is reflected in the latent space of its vector embeddings. This has great potential for use in experimentation of natural language algorithms, by providing a source of logically consistent data.

This project is an ongoing effort to align mathematical concepts in natural language with online repositories of formalized mathematics like `mathlib`.[8] As described in [12], this type of alignment is called *automatically found aligment*.

[7] `https://www.nltk.org/howto/wordnet.html`

[8] `https://github.com/leanprover-community/mathlib`

Term	IS-A	**Term**	IS-A
hyperbolic_metric	-1.11		
euclidean_metric	-0.59	digraph	-0.51
metrics	-0.58	undirected_graph	-0.35
riemannian_metric	-0.46	undirected	-0.20
riemannian	-0.42	**directed_graph**	0.0
riemannian_manif	-0.40	graph	1.24
curvature	-0.27		
metric	0.0		
banach_algebra	-1.11	probability_distr	-0.24
normed_space	-0.98	**random_variable**	0.0
banach_spaces	-0.38	expectation	0.23
banach	-0.29	distribution	0.46
closed_subspace	-0.25	probability	0.67
banach_space	0.0		
norm	0.79		

Table 5: Query results sorted by IS-A score (terms in upper lines tend to depend semantically on lower lines). Cosine similar words were sorted by the IS-A rating of the term in bold font.

In the near future we plan to further improve on the classification and NER tasks by creating a data set using solely the neural version of the classifier and NER model. Also, by using state-of-the-art methods like the masked transformer language model [5] to further improve the results. We also plan to compile the complete dependency graph in one large graph database.

References

[1] Rami Aly, Shantanu Acharya, Alexander Ossa, Arne Köhn, Chris Biemann & Alexander Panchenko (2019): *Every Child Should Have Parents: A Taxonomy Refinement Algorithm Based on Hyperbolic Term Embeddings*. In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, Association for Computational Linguistics, Florence, Italy, pp. 4811–4817, doi:10.18653/v1/P19-1474.

[2] Luis Berlioz (2019): *Creating a Database of Definitions From Large Mathematical Corpora*. In Edwin C. Brady, James H. Davenport, William M. Farmer, Cezary Kaliszyk, Andrea Kohlhase, Michael Kohlhase, Dennis Müller, Karol Pak & Claudio Sacerdoti Coen, editors: *Joint Proceedings of the FMM and LML Workshops, Doctoral Program and Work in Progress at the Conference on Intelligent Computer Mathematics 2019 co-located with the 12th Conference on Intelligent Computer Mathematics (CICM 2019), Prague, Czech Republic, July 8-12, 2019*, *CEUR Workshop Proceedings* 2634, CEUR-WS.org. Available at http://ceur-ws.org/Vol-2634/WiP2.pdf.

[3] Steven Bird, Ewan Klein & Edward Loper (2009): *Natural Language Processing with Python*. O'Reilly. Available at http://www.oreilly.de/catalog/9780596516499/index.html.

[4] Jacob Cohen (1960): *A Coefficient of Agreement for Nominal Scales*. *Educational and Psychological Measurement* 20(1), pp. 37–46, doi:10.1177/001316446002000104.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee & Kristina Toutanova (2019): *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. In Jill Burstein, Christy Doran & Thamar Solorio, editors: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Association for Computational Linguistics, pp. 4171–4186, doi:10.18653/v1/n19-1423.

[6] Christiane Fellbaum (2010): *WordNet*. In: *Theory and applications of ontology: computer applications*, Springer, pp. 231–243, doi:10.1093/ijl/17.2.161.

[7] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North & Kiem-Phong Vo (1993): *A Technique for Drawing Directed Graphs*. *IEEE Trans. Software Eng.* 19(3), pp. 214–230, doi:10.1109/32.221135.

[8] Deyan Ginev, Mihnea Iancu, Constantin Jucovschi, Andrea Kohlhase, Michael Kohlhase, Akbar Oripov, Jürgen Schefter, Wolfram Sperber, Olaf Teschke & Tom Wiesing (2016): *The SMGloM Project and System: Towards a Terminology and Ontology for Mathematics*. In Gert-Martin Greuel, Thorsten Koch, Peter Paule & Andrew J. Sommese, editors: *Mathematical Software - ICMS 2016 - 5th International Conference, Berlin, Germany, July 11-14, 2016, Proceedings*, *Lecture Notes in Computer Science* 9725, Springer, pp. 451–457, doi:10.1007/978-3-319-42432-3_58.

[9] Deyan Ginev & Bruce R. Miller (2019): *Scientific Statement Classification over arXiv.org*. *CoRR* abs/1908.10993. Available at `http://arxiv.org/abs/1908.10993`.

[10] Marti A. Hearst (1992): *Automatic Acquisition of Hyponyms from Large Text Corpora*. In: *COLING 1992 Volume 2: The 14th International Conference on Computational Linguistics*, doi:10.3115/992133.992154.

[11] Zhiheng Huang, Wei Xu & Kai Yu (2015): *Bidirectional LSTM-CRF Models for Sequence Tagging*. *CoRR* abs/1508.01991. Available at `http://arxiv.org/abs/1508.01991`.

[12] Cezary Kaliszyk, Michael Kohlhase, Dennis Müller & Florian Rabe (2016): *A Standard for Aligning Mathematical Concepts*. In Andrea Kohlhase, Paul Libbrecht, Bruce R. Miller, Adam Naumowicz, Walther Neuper, Pedro Quaresma, Frank Wm. Tompa & Martin Suda, editors: *Joint Proceedings of the FM4M, MathUI, and ThEdu Workshops, Doctoral Program, and Work in Progress at the Conference on Intelligent Computer Mathematics 2016 co-located with the 9th Conference on Intelligent Computer Mathematics (CICM 2016), Bialystok, Poland, July 25-29, 2016*, *CEUR Workshop Proceedings* 1785, CEUR-WS.org, pp. 229–244. Available at `http://ceur-ws.org/Vol-1785/W24.pdf`.

[13] Tomás Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado & Jeffrey Dean (2013): *Distributed Representations of Words and Phrases and their Compositionality*. *CoRR* abs/1310.4546, doi:10.5555/2999792.2999959.

[14] Maximilian Nickel & Douwe Kiela (2017): *Poincaré Embeddings for Learning Hierarchical Representations*. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan & Roman Garnett, editors: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 6338–6347. Available at `https://proceedings.neurips.cc/paper/2017/hash/59dfa2df42d9e3d41f5b02bfc32229dd-Abstract.html`.

[15] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot & Edouard Duchesnay (2011): *Scikit-learn: Machine Learning in Python*. *J. Mach. Learn. Res.* 12, pp. 2825–2830. Available at `http://dl.acm.org/citation.cfm?id=2078195`.

[16] Jeffrey Pennington, Richard Socher & Christopher D. Manning (2014): *Glove: Global Vectors for Word Representation*. In Alessandro Moschitti, Bo Pang & Walter Daelemans, editors: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, ACL, pp. 1532–1543, doi:10.3115/v1/d14-1162.

[17] Alexandru Tifrea, Gary Bécigneul & Octavian-Eugen Ganea (2019): *Poincare Glove: Hyperbolic Word Embeddings*. In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, OpenReview.net. Available at `https://openreview.net/forum?id=Ske5r3AqK7`.

[18] Chengyu Wang, Xiaofeng He & Aoying Zhou (2017): *A Short Survey on Taxonomy Learning from Text Corpora: Issues, Resources and Recent Advances*. In: *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, Association for Computational Linguistics, Copenhagen, Denmark, pp. 1190–1203, doi:10.18653/v1/D17-1123.

Implementing Security Protocol Monitors

Yannick Chevalier
Irit, Université Paul Sabatier
Toulouse
France
ychevali@irit.fr

Michaël Rusinowitch
Lorraine University, Cnrs, Inria
Nancy
France
michael.rusinowitch@loria.fr

Cryptographic protocols are often specified by narrations, *i.e.*, finite sequences of message exchanges that show the intended execution of the protocol. Another use of narrations is to describe attacks. We propose in this paper to compile, when possible, attack describing narrations into a set of tests that honest participants can perform to exclude these executions. These tests can be implemented in monitors to protect existing implementations from rogue behaviour.

1 Introduction

Cryptographic protocols are designed to prescribe message exchanges between agents in a hostile environment in order to guarantee some security properties. In particular security properties such as confidentiality or authentication are violated when there exists an execution of the protocol in which they do not hold. However it has often been found that under certain circumstances, and after its deployment, a protocol failed to adequatly protect its participants. These circumstances usually involve one or more sessions, and the participation of a dishonest agent hereafter called the intruder. When the attack is on a specific implementation of a protocol, its mitigation usually amounts to fixing this implementation.

However, some attacks are related to the exchanges of messages prescribed by the protocol, and not in the actual handling of these messages by participants. In that case, the only recourse is—when available—to alter the sequence of acceptable messages. This can be implemented by changing the format of the messages exchanged or by stopping an execution once it has been detected that the attack may be under way. We consider in this paper only the second approach, in which the participants behaviour is altered in order to reject some possible executions of the protocol.

Let us consider for example the Needham-Schroder Public Key (*NSPK*) mutual authentication protocol [11] described by the following sequence of messages between roles A and B:

$$\begin{array}{rl} A \textbf{ knows} & A, B, K_A, K_B, K_A^{-1} \\ B \textbf{ knows} & A, B, K_A, K_B, K_B^{-1} \\ A \to B & : \mathrm{enc}(A, N_A, K_B) \\ B \to A & : \mathrm{enc}(N_A, N_B, K_A) \\ A \to B & : \mathrm{enc}(N_B, K_B) \end{array}$$

The attack on this protocol discovered by Lowe [10] is described as follows, where $I(A)$ denotes the

T. Kutsia (Ed.): Symbolic Computation
in Software Science (SCSS'21)
EPTCS 342, 2021, pp. 22–34, doi:10.4204/EPTCS.342.3

intruder impersonating the agent A:

$$\begin{aligned} A \to I &: \mathrm{enc}(A, N_A, K_I) \\ I(A) \to B &: \mathrm{enc}(A, N_A, K_B) \\ B \to I(A) &: \mathrm{enc}(N_A, N_B, K_A) \\ I \to A &: \mathrm{enc}(N_A, N_B, K_A) \\ A \to I &: \mathrm{enc}(N_B, K_I) \\ I(A) \to B &: \mathrm{enc}(N_B, K_B) \end{aligned}$$

This execution is an attack because B believes he has participated in a session with A whereas A never exchanged a message with B. As is the case in this attack narration, we assume from now on and without loss of generality [8] that in an attack, every message is sent to or received from the intruder. A fix, proposed in [10], consisting in altering the second message to include the name of the sender. The only drawback to such fixes is that implementations of the amended protocol are not interoperable with implementations of the original protocol. For widely deployed real-life protocols, interoperability must be maintained and thus the amended version coexists with the original one for years, leaving open an attack vector for attackers.

Our proposal aims at keeping the original version, but extended with additional tests. This extension involves the creation of a monitor for the actions of honest participants that furthermore may have access to some secret pieces of information held by these participants. In the case of Lowe's attack, this access is unnecessary, as it suffices for B to check that the message he receives at the third step is equal to the message sent by A.

We present in this paper an algorithm to implement a security protocol monitor. Given the input messages that participants are willing to share with the monitor, it basically amounts to computing the conditions to be checked in order to exclude a given narration from the possible executions of the protocol.

Related works This article is based on the refinement relation between traces introduced in [4]. An extension to the case where an attack can be excluded based on the information in only one session of a participant has been proposed in [9].

By contrast our approach stems from the line of work initiated in [3, 2] where the authors advocate for the prevention of attacks through detection and eventually retaliation against the attacker. Also, [7] presents in more details an architecture in which the analysis we present in this paper can be conducted with a better control on the messages, and also introduce the idea of applicative firewalls for security protocols.

Outline We recall in Sec. 2 how to represent protocols and roles and how to implement them as active frames. In Sec. 3 we formally introduce protocol monitors to control messages and manage knowledge shared by collaborating agents, in order to detect and block attacks. In Sec. 4 we show how to synthesize monitors from tests that can be derived automatically. We conclude in Sec. 6.

2 Role-based Protocol Specifications

2.1 Messages and basic operations

We consider an infinite set of free constants $\mathscr{C}$ and an infinite set of variables $\mathscr{X}$. For each signature $\mathscr{F}$ (*i.e.* a set of function symbols with arities), we denote by $\mathrm{T}(\mathscr{F})$ (resp. $\mathrm{T}(\mathscr{F}, \mathscr{X})$) the set of terms

over $\mathscr{F} \cup \mathscr{C}$ (resp. $\mathscr{F} \cup \mathscr{C} \cup \mathscr{X}$). The former is called the set of ground terms over $\mathscr{F}$, while the latter is simply called the set of terms over $\mathscr{F}$. Variables are denoted by x and decorations thereof, but for a distinguished subset $(v_i)_{i\in\mathbb{N}}$ employed to denote positions in a sequence. Terms are denoted by s, t, and finite sets of terms are written $E, F, \ldots$, and decorations thereof, respectively. In a signature $\mathscr{F}$ a *constant* is either a *free constant* or a function symbol of arity 0 in $\mathscr{F}$. Given a term t we denote by $\mathrm{Var}(t)$ the set of variables occurring in t and $\mathrm{Cons}(t)$ the set of free constants occurring in t. A (ground) substitution σ is an idempotent mapping from $\mathscr{X}$ to $\mathrm{T}(\mathscr{F},\mathscr{X})$ ($\mathrm{T}(\mathscr{F})$) and its support $\mathrm{Supp}(\sigma) = \{x \,|\, \sigma(x) \neq x\}$ is a finite set. The application of a substitution σ on a term t (resp. a set of terms E) is denoted $t\sigma$ (resp. $E\sigma$) and is equal to the term t (resp. the set of terms E) where all variables $x \in \mathrm{Supp}(\sigma)$ have been replaced by the term $x\sigma$.

Terms are manipulated by applying *operations* on them. These operations are defined by a subset of the signature $\mathscr{F}$ called the *set of public constructors*. A context $C[v_1,\ldots,v_n]$ is a term in which $\mathrm{Var}(C[v_1,\ldots,v_n]) \subseteq \{v_1,\ldots,v_n\}$, $\mathrm{Const}(C[v_1,\ldots,v_n]) = \emptyset$, and all non-variable symbols are public constructors, including possibly non-free constant. We will specify the effects of operations on the messages and the properties of messages by equations. When the index n is clear, we omit the possible variables list and denote contexts C. An *equational presentation* $\mathscr{E} = (\mathscr{F}, E)$ is defined by a set E of equations $u = v$ with $u, v \in \mathrm{T}(\mathscr{F},\mathscr{X})$. The *equational theory* generated by $(\mathscr{F}, E)$ on $\mathrm{T}(\mathscr{F},\mathscr{X})$ is the smallest congruence containing all instances of axioms in E (free constants can also be used for building instances) [6]. We write $s =_{\mathscr{E}} t$ as the congruence relation between two terms s and t. By abuse of terminology we also call $\mathscr{E}$ the equational theory generated by the presentation $\mathscr{E}$ when there is no ambiguity.

A *deduction system* is defined by a triple $(\mathscr{E}, \mathscr{F}, \mathscr{F}_p)$ where $\mathscr{E}$ is an equational presentation on a signature $\mathscr{F}$ and $\mathscr{F}_p$ a subset of *public* constructors in $\mathscr{F}$.

Example 1. Public key cryptography. For instance the following deduction system models public key cryptography:

$$\begin{array}{l}(\{\mathrm{dec}(\mathrm{enc}(x,y),y^{-1}) = x\},\\ \{\mathrm{dec}(\_,\_),\mathrm{enc}(\_,\_),\_^{-1}\},\\ \{\mathrm{dec}(\_,\_),\mathrm{enc}(\_,\_)\})\end{array}$$

The equational theory is reduced here to a single equation that expresses that one can decrypt a ciphertext when the inverse key is available. The inverse function $\_^{-1}$ is not public, as it cannot be computed in reasonable time by participants.

Example 2. Nonce generation. Nonces are random values that are critical to the analysis of cryptography and cryptographic protocols. To give an agent the capacity to generate new values, we assume the existence of an infinite set of constants $\mathscr{C}_{\mathscr{N}}$ away from $\mathscr{C}$ such that each value in this set can be generated:

$$\mathscr{N} = (\emptyset, \mathscr{C}_{\mathscr{N}}, \mathscr{C}_{\mathscr{N}})$$

Note this model makes sense only in the case where the attacker is only one agent, or a set of information sharing agents [13], as an agent cannot otherwise construct nonces generated by another, independent, agent.

Test systems. In order to express verifications performed by an agent on received messages we introduce test systems:

Definition 1. (Test systems) Let $\mathscr{D}$ be a deduction system with an equational theory $\mathscr{E}$. A *$\mathscr{D}$-test system* $S[v_1,\ldots,v_n]$ is a finite set of equations denoted by $(C_i \stackrel{?}{=} C'_i)_{i\in\{1,\ldots,n\}}$ with $\mathscr{D}$-contexts $C_i[v_1,\ldots,v_n], C'_i[v_1,\ldots,v_n]$. It is satisfied by a substitution σ, and we denote by $\sigma \models S[v_1,\ldots,v_n]$, if for all $i \in \{1,\ldots,n\}$ the equality $C[v_1,\ldots,v_n]_i\sigma =_{\mathscr{E}} C'_i[v_1,\ldots,v_n]\sigma$ holds.

As usual we simply denote a test system S if the maximal indice n is clear from the context.

2.2 Traces and active frames

We model messages with terms. The sequence of messages received and sent by a principal is a *trace*, and is thus a finite sequence of labeled messages:

Definition 2. (Trace) A *trace* is a finite sequence of messages each with label (or polarity) ! or ?.

Messages with label ! (resp. ?) are said to be "sent" (resp."received"). Given a trace $\Lambda = !/?t_1, \ldots, !/?t_n$ we write $?\Lambda$ (resp. $!\Lambda$) as a short-hand for $?m_1, \ldots, ?m_n$, (resp. $!m_1, \ldots, !m_n$). Given a trace $\Lambda = !/?m_1, \ldots, !/?m_n$ we denote by $\sigma_\Lambda = \{v_1 \mapsto m_1, v_n \mapsto m_n\}$ the substitution mapping each variable v_i to the ith message occurring in Λ. To simplify notation we also denote by $C[v_1, \ldots, v_n] \cdot \Lambda$, or more simply $C \cdot \Lambda$, the application of the substitution σ_Λ on the context $C[v_1, \ldots, v_n]$. Accordingly, we say that a trace Λ satisfies an equality $C_1 = C_2$, and denote it by $\Lambda \models C_1 \stackrel{?}{=} C_2$, whenever $C_1 \cdot \Lambda =_{\mathscr{E}} C_2 \cdot \Lambda$.

Operations on traces Let Λ be a trace. We say that a Λ is positive (*resp.* negative) if all its labels are ! (*resp.* ?). We denote $\Lambda^?$ (*resp.* $\Lambda^!$) the subsequence of Λ of terms labeled with ? (*resp.* !). We denote $-\Lambda$ the trace in which all the labels in Λ are inverted. Finally, we denote $\mathrm{input}(\Lambda)$ (resp. $\mathrm{output}(\Lambda)$) the trace $-\Lambda^?$ (resp. $-\Lambda^!$).

Active frames An *active frame* represents the actions of a principal participating in a protocol. It is a sequence of steps, and at step i the principal either sends a message, constructed from the messages received at steps $j < i$, or receives and message, and accepts it if it satisfies some tests constructed from it and messages received at steps $j < i$. To simplify exposition, at a step i, we call these messages received at steps $j < i$ the messages *already known* at step i, or just *already known* if the step is clear from context. As in the case of traces, messages sent are labeled $!v_i$ (and v_i is an output variable) and those received are labeled $?v_i$ (and v_i is an input variable). Since the available contexts depend upon the deduction system, the notion of active frame is also parameterised by a deduction system.

Definition 3. Given a deduction system $\mathscr{D}$, a *$\mathscr{D}$-active frame* is a sequence $(T_i)_{1 \leq i \leq k}$ where

$$T_i = \begin{cases} !v_i \text{ with } v_i \stackrel{?}{=} C_i[v_1, \ldots, v_{i-1}] & \text{(send)} \\ \textbf{or} & \\ ?v_i \text{ with } S_i[v_1, \ldots, v_i] & \text{(receive)} \end{cases}$$

Without loss of generality and reusing the above notations, a simple recursion shows that we can assume that all variables in $C_i[v_1, \ldots, v_{i-1}]$ are labeled with ? at a step $j < i$, and that all variables in $S_i[v_1, \ldots, v_i]$ are labeled with ? at a step $j \leq i$. Without loss of generality, from now on we assume that this is the case for all the active frames we consider.

Example 3. The principal A in the description of the NSPK protocol can be modeled by an active frame as follows, with the caveat that we have renamed the v_i variables for more clarity:

$$\begin{array}{l} (?x_{N_A} \text{ with } \emptyset, ?x_A \text{ with } \emptyset, ?x_B \text{ with } \emptyset, ?x_{K_A} \text{ with } \emptyset, ?x_{K_B} \text{ with } \emptyset, ?x_{K_A^{-1}} \text{ with } \emptyset, \\ !x_{msg_1} \text{ with } x_{msg_1} \stackrel{?}{=} \mathrm{enc}(\langle x_A, x_{N_A} \rangle, x_{K_B}), \\ ?x_r \text{ with } \emptyset \\ !x_{msg_2} \text{ with } x_{msg_2} \stackrel{?}{=} \mathrm{enc}(\pi_2(\mathrm{dec}(x_r, x_{K_A^{-1}})), x_{K_B})) \end{array}$$

Algebraically, by describing a principal's actions, active frames are partial operations on the set of traces and map a sequence of messages sent by someone else and accepted to the sequence of received and sent messages by a principal. We formalize these notions as follows:

Definition 4. Let $\mathscr{D}$ be a deduction system with equational theory $\mathscr{E}$. Let $\varphi = (T_i)_{1\leq i\leq n}$ be an active frame, where the T_i's are as in Definition 3, and where the input variables are $?v_{\alpha_1},\ldots,?v_{\alpha_k}$. Let Λ be a positive trace of length k, θ be the renaming of variables $\left\{v_{\alpha_j}\mapsto v_j\right\}_{1\leq j\leq k}$, and S be the union of the test systems in φ. The *evaluation* of φ on Λ is denoted $\varphi\cdot\Lambda$. It is defined, and we say that φ *accepts* s, if $S\cdot s$ is satisfiable. In that case, it is the trace $(m_1,\ldots,m_n)$ where:

$$m_i = \begin{cases} !C_i\cdot\theta\sigma_\Lambda & \text{If } v_i \text{ has label ! in } T_i \\ ?v_i\cdot\theta\sigma_\Lambda & \text{If } v_i \text{ has label ? in } T_i \end{cases}$$

Example 4. Let Λ_A be the trace of the principal A in the the specification of the NSPK protocol in the introduction, $r=\mathrm{tr}(A)$, and ϕ_A be the active frame of Ex. 3. Let M be the message $\mathrm{msg}(B,\mathrm{enc}(\langle N_A,N_b\rangle,K_A))$. We have:

$$\mathrm{input}(\Lambda_A) = (!N_A,!A,!B,!K_A,!K_B,!K_A^{-1},!M)$$

and $\phi_A\cdot\mathrm{input}(\Lambda_A)$ is the trace:

$$\begin{array}{l}(?N_A,?A,?B,?K_A,?K_B,?K_A^{-1},\\ !\mathrm{msg}(B,\mathrm{enc}(\langle A,N_A\rangle,K_B)),\\ ?M,!\mathrm{msg}(B,\mathrm{enc}(\pi_2(\mathrm{dec}(\mathrm{payload}(M),K_A^{-1})),K_B))\end{array}$$

Modulo the equational theory, this trace is equal to:

$$\begin{array}{l}(?N_A,?A,?B,?K_A,?K_B,?K_A^{-1},\\ !\mathrm{msg}(B,\mathrm{enc}(\langle A,N_A\rangle,K_B)),?M,!\mathrm{msg}(B,\mathrm{enc}(N_b,K_B))\end{array}$$

It is not coincidental that in Ex. 4 the traces $\varphi_A\cdot\mathrm{input}(\Lambda_A)$ and Λ_A are equal as it means that within the active frame, the sent messages are composed from received ones in such a way that when someone sends the messages expected in Λ_A, the execution of A is described by Λ_A. This relation gives us a criterion to define what an implementation of a trace is.

Definition 5. An active frame φ is an *implementation* of a trace Λ if φ accepts $\mathrm{input}(\Lambda)$ and $\varphi\cdot\mathrm{input}(\Lambda)=_{\mathscr{E}}\Lambda$.

If a trace admits an implementation we say this trace is *executable*. Conversely we say that a trace t is an *execution* of an active frame φ whenever φ is an implementation of t.

2.3 Computation of an implementation

We present in this section a method, parameterised by the deduction system $\mathscr{D}$, to compute an active frame implementing an executable trace. To build such an implementation we need to compute, given a message t sent at step i, a $\mathscr{D}$-context C_i that evaluates to t when applied to the previously received messages. This reachability problem is unsolvable in general. Hence we have to consider systems that admit a reachability algorithm, formally defined below:

Definition 6. Given a deduction system $\mathscr{D}$ with equational theory $\mathscr{E}$, a *$\mathscr{D}$-reachability algorithm* $\mathscr{A}_\mathscr{D}$ computes, given a positive trace Λ of length n and a term t, a $\mathscr{D}$-context $\mathscr{A}_\mathscr{D}(s,t)=C[v_1,\ldots,v_n]$ such that $C\cdot\Lambda=_{\mathscr{E}}t$ iff there exists such a context and $\bot$ otherwise.

For the many theories that admit a reachability algorithm, it can be employed as an oracle to compute the contexts in sent messages and therefore to derive an implementation of a trace s. We thus have the following theorem (see a proof in [4]).

Theorem 1. *If a $\mathscr{D}$-reachability algorithm exists then it can be decided whether a trace s is executable and if so one can compute an implementation of s.*

2.4 Computation of a prudent implementation

An implementation does not necessarily checks the conformity of the messages with the intended patterns, *e.g.*, the active frame in Ex. 4 neither checks that x_r is really an encryption with the public key x_{K_A} of a pair, nor that the first argument of the encrypted pair has the same value as the nonce x_{N_A}.

Any of the algorithms proposed so far in the literature for the compilation of cryptographic protocols would require at least these tests. We now present an algorithm that computes these kinds of checks for an arbitrary deduction system. It formalizes a check as an equation between $\mathscr{D}$-contexts over messages received so far, including the initial knowledge. For example, and reusing the notations of Ex. 3 it computes that upon reception of the message the initiator must, among other tests, check the validity of the equation:

$$\pi_1(\mathrm{dec}(x_r, x_{K_A^{-1}})) \stackrel{?}{=} x_{N_A}$$

We formalize in the definition below which traces Λ' are acceptable by an agent expecting a trace Λ. We define the acceptable traces as the refinements of Λ, that is traces Λ' such that every test system accepting Λ also accepts Λ'.

Definition 7. Let Λ, Λ' be two positive traces of identical length. We say that Λ' *refines* Λ if, for any pair of $\mathscr{D}$-contexts (C_1, C_2) one has $C_1 \cdot \Lambda = C_2 \cdot \Lambda$ implies $C_1 \cdot \Lambda' = C_2 \cdot \Lambda'$.

Consider for example the following traces Λ and Λ':

$$\begin{cases} \Lambda' = (!\mathrm{enc}(a,k), !\mathrm{enc}(a,k'), !k, !\mathbf{k}'', !a) \\ \Lambda = (!\mathrm{enc}(a,k), !\mathrm{enc}(a,k'), !k, !\mathbf{k}', !a) \end{cases}$$

since all equalities that can be checked on σ can be checked on σ'. Two traces s, s' that refine one another are *equivalent*. This definition is an adaptation to our setting of the classic notion of static equivalence [1].

When the behaviour of a principal is defined by a trace Λ, we expect that any implementation of that principal accepts the trace input(Λ). Thus, and as long as only equality tests are considered, we expect any implementation of the trace Λ to also accept any refinement of input(Λ). We define a *prudent implementation* of Λ as an implementation that only accepts inputs that refine the inputs in Λ.

Definition 8. An active frame φ is a *prudent implementation* of a trace Λ if φ is an implementation of Λ and any trace Λ' accepted by φ is a refinement of input(Λ).

As already noted in [4], most deduction systems considered in the context of cryptographic protocols analysis have the property that it is possible to compute, given a positive trace, a finite set of context pairs that summarizes all possible equalities. Given a positive trace Λ we denote P_Λ the (infinite) set of context pairs (C_1, C_2) such that $C_1 \cdot s = C_2 \cdot s$.

Definition 9. A deduction system $\mathscr{D}$ has the *finite basis property* if for each positive trace Λ one can compute a finite set P_Λ^f of pairs of $\mathscr{D}$-contexts such that, for each positive trace Λ':

$$P_\Lambda \subseteq P_{\Lambda'} \text{ iff } P_\Lambda^f \subseteq P_{s'}$$

Let us now assume that a deduction system $\mathscr{D}$ has the finite basis property. There thus exists an algorithm $\mathscr{A}'_{\mathscr{D}}$ that takes a positive trace Λ as input, computes a finite set P_Λ^f of context pairs (C, C') and returns as a result the test system $S_\Lambda : \left\{ C \stackrel{?}{=} C' \mid (C, C') \in P_\Lambda^f \right\}$. For any positive trace Λ' of length n, by definition of S_Λ we have that $S_\Lambda \cdot \Lambda'$ is satisfiable if and only if s' is a refinement of s. We are now ready to present our algorithm for the compilation of strands into active frames.

Algorithm Given a trace Λ and assuming that the deduction system $\mathscr{D}$ has a reachability algorithm and the finite basis property, and let Λ be a trace of length n, and let us denote Λ^i for $1 \leq i \leq n$ the prefix of length i of Λ, and $\Lambda(i)$ the ith element of Λ. We construct a prudent implementation $\varphi_\Lambda = (T_i)_{i=1,\ldots,n}$ of Λ as follows:

$$T_i = \begin{cases} !v_i \text{ with } v_i \stackrel{?}{=} \mathscr{A}_\mathscr{D}(\Lambda^{i-1}, t_i) & \text{If } \Lambda(i) = !t_i \\ ?v_i \text{ with } \mathscr{A}'_\mathscr{D}(\Lambda^i) & \text{If } \Lambda(i) = ?t_i \end{cases}$$

By construction we have the following theorem[4]:

Theorem 2. *Let $\mathscr{D}$ be a deduction system that has a $\mathscr{D}$-ground reachability algorithm and has the finite basis property. Then for any executable trace Λ one can compute a prudent implementation φ_Λ of Λ.*

2.5 Protocol implementation and execution

It is customary to describe a protocol by giving its intended execution, either using a message sequence chart or an Alice&Bob notation. We note that the same notation is also employed to described *e.g.* attacks on that protocol. Beyond their syntax, the characteristic of such description is to associate to a generic principal (a rôle, in the case of a protocol specification, a participant in the case of an attack description) a trace describing its actions, and how these actions interact with the other principal actions. This association of a participant with a trace is formalised by a function mapping *strands* [14], *i.e.* principals, rôles, etc., to traces. We define a protocol to be just one such mapping.

Definition 10. (Protocol) A *protocol* is a couple $P = (\Xi_P, \mathrm{tr}_P)$ where Ξ_P is a finite set of strands and tr_P maps Ξ_P to the set of traces.

When a protocol is intended to be a protocol specification, we refer to strands as the rôles of that protocol (*e.g.* the rôles A and B in the NSPK protocol. A strand ξ is *positive* in a protocol P if $\mathrm{tr}_P(\xi)$ is a positive trace.

In the preceding definition the function tr_P prescribes for each role $\xi \in \Xi_P$ the sequence of actions to be performed by an agent playing this role in any protocol instance. In the following, when there is no ambiguity in the considered protocol, we identify a strand ξ with its trace $\mathrm{tr}(\xi)$.

We have worked so far on the implementation of the trace of a role in a protocol, but the definitions lift to the level of an implementation of a protocol as follows.

Definition 11. (Protocol implementation) An *implementation of a protocol* $P = (\Xi_P, \mathrm{tr}_P)$ is a couple (Ξ_P, Φ_P) where Φ maps each role $\xi \in \Xi_P$ to an active frame such that $\Phi(\xi)$ is an implementation of $\mathrm{tr}_P(\xi)$. It is *prudent* if moreover for each $\xi \in \Xi_P$, $\Phi(\xi)$ is a prudent implementation of $\mathrm{tr}_P(\xi)$.

From now on we consider only prudent implementations of protocols, *i.e.* implementations whose execution is a refinement of the protocol specification.

Definition 12. (Protocol execution) Let $P = (\Xi_P, \Phi_P)$ be a protocol implementation. A triple $E = (\Xi_E, \mathrm{tr}_E, R_E)$ where:

1. Ξ_E is a set of strands away from Ξ_P;
2. (Ξ_E, tr_E) is a protocol;
3. $R_E : \Xi_E \to \Xi_P \cup \{I\}$.

is a *protocol execution* of P if, for each $\xi \in \Xi_E$, if $R_E(\xi) \neq I$ then $\mathrm{tr}_E(\xi)$ is an execution of $\Phi_P(R_E(\xi))$.

The strand I denotes an *Intruder* who does not necessarily follows the directions prescribed by the protocol. A protocol execution is *honest* if $R_E(\Xi_E) \subseteq \Xi_P$. Strands in Ξ_E are called the *participants* of the protocol execution E. The function R_E maps each (honest) participant to its rôle in the protocol.

3 Protocol monitor

To mitigate an attack on a protocol, a monitor has to coordinate the participants to detect and stop an instance of a known flaw. This coordination is built according to data the participants are willing to share to prevent the attack. Our monitor construction relies on the description of the data the participants are willing to share, a description of the attack, and a description of the expected behaviour of the participants, and we compute tests (when possible) to distinguish an instance of the attack from the normal execution.

In Def. 13, for each participant A, $\mathrm{tr}_M(A)$ contains the same inputs as $\mathrm{tr}_P(A)$, and the messages sent in $\mathrm{tr}_M(A)$, are the pieces of data shared by A with the monitor.

Definition 13. (Protocol Monitor) Let $P = (\Xi_P, \mathrm{tr}_P)$ and $M = (\Xi_M, \mathrm{tr}_M)$ be two protocols. We say that M *is a monitor for* P if 1. $\Xi_M = \Xi_P$; 2. M is executable; and 3. For each $\xi \in \Xi_M$ we have $\mathrm{input}(\mathrm{tr}_M(\xi)) = \mathrm{input}(\mathrm{tr}_P(\xi))$.

Proposition 1. *Let $P = (\Xi_P, \mathrm{tr}_P)$ be a protocol, $M = (\Xi_P, \mathrm{tr}_M)$ be a monitor of P, $I_X = (\Xi_P, \Phi_X)$ be any implementation of $X \in \{P, M\}$, and $E = (\Xi_E, \Phi_E, R_E)$ be an honest execution of I_P.*

If I_P is prudent and $\Phi_M(R_E(\xi))$ accepts $\mathrm{input}(\mathrm{tr}_E(\xi))$, then $\Phi_M(R_E(\xi)) \cdot \mathrm{input}(\mathrm{tr}_E(\xi))$ is a refinement of $\mathrm{tr}_M(R_E(\xi))$.

Proof. Assume there exists $\xi_R \in \Xi_M$ and $\xi_e \in \Xi_E$ with $R_E(\xi_e) = \xi_R$ such that $\Phi_M(\xi_R) \cdot \mathrm{input}(\mathrm{tr}_E(\xi_e))$ is not a refinement of $\mathrm{tr}_M(\xi_R)$. That is, there exists pairs of contexts C_1, C_2 such that $\mathrm{tr}_M(\xi_r) \models C_1 = C_2$ but $\Phi_M(\xi_R) \cdot \mathrm{input}(\mathrm{tr}_E(\xi_e)) \not\models C_1 = C_2$. Without loss of generality we can assume that C_1, C_2 are built upon the input variables of $\Phi_M(R_E(\xi))$, that is, with $\theta : \left\{v_{i_j} \mapsto v_j\right\}_{1 \le j \le k}$, where i_j is the jth input step of $\mathrm{tr}_M(\xi_R)$:

$$\begin{cases} \mathrm{input}(\mathrm{tr}_M(\xi_R)) \models C_1\theta = C_2\theta \\ \mathrm{input}(\Phi_M(\xi_R) \cdot \mathrm{input}(\mathrm{tr}_E(\xi_e))) \not\models C_1\theta = C_2\theta \end{cases}$$

Since I_M is an implementation of M, by definition the second assertion is equal to $\mathrm{input}(\mathrm{tr}_E(\xi_e)) \not\models C_1\theta = C_2\theta$. By definition of a monitor, we have $\mathrm{input}(\mathrm{tr}_M(\xi_R)) = \mathrm{input}(\mathrm{tr}_P(\xi_R))$. Thus, we have:

$$\begin{cases} \mathrm{input}(\mathrm{tr}_P(\xi_R)) \models C_1\theta = C_2\theta \\ \mathrm{input}(\mathrm{tr}_E(\xi_R)) \not\models C_1\theta = C_2\theta \end{cases}$$

Hence $\mathrm{input}(\mathrm{tr}_E(\xi_R))$ is not a refinement of $\mathrm{input}(\mathrm{tr}_P(\xi_R))$, and thus $\Phi_P(\xi_R)$ cannot be a prudent implementation of $\mathrm{tr}_P(\xi_R)$. □

Definition 14. (Execution Log) Let $P = (\Xi_P, \mathrm{tr}_P)$ be a protocol, $I_P = (\Xi_P, \Phi_P)$ be an implementation of P, $E = (\Xi_E, \mathrm{tr}_E, R_E)$ be an execution of I_P, $<_E$ be an arbitrary total order on the participants, and $I_M = (\Sigma_P, \phi_M)$ be an implementation of a monitor M of P. The *execution log* of E for monitor M is the concatenation of the traces:

$$\mathrm{output}(\phi_M(R_E(\xi_e)) \cdot \mathrm{input}(\mathrm{tr}_E(\xi_e)))$$

for $\xi_e \in \Xi_E$ such that $R_E(\xi_e) \neq I$ in the increasing order with respect to $<_E$.

Proposition 2. *Let $P = (\Xi_P, \mathrm{tr}_P)$ be a protocol, $I_P = (\Xi_P, \Phi_P)$ be an implementation of P, $E = (\Xi_E, \mathrm{tr}_E, R_E)$ be an execution of I_P, $<_E$ be an arbitrary total order on the participants, and $I_M = (\Xi_P, \Phi_M)$ be an implementation of a monitor M of P. Then there exists a unique execution log of E for M.*

Proof. For each $\xi_e \in \Sigma_E$ let $\varphi_e = \Phi_M(R_E(\xi_e)$ be the active frame executed by ξ_e, and let $\text{in}_e = \text{input}(\text{tr}_E(\xi_e))$ denote the messages received by ξ_e. Since sent messages are built by a context over preceding messages an easy recurrence shows that the value of each message in $\varphi_e \cdot \text{in}_e$ is uniquely defined by the values in $\text{input}(\text{tr}_E(\xi_e))$. Thus $\text{output}(\varphi_e \cdot \text{in}_e)$ is uniquely defined for each participant $\xi_e \in \Xi_E$. Since the order $<_E$ is total the concatenation of these traces is unique. □

Since the ordering $<_E$ is arbitrary, we usually omit any reference to it. By Prop. 2 the execution log depends only on the monitor, not on its implementation. Accordingly we denote it $\log_{I_P,M}(E)$. Assuming there exists a $\mathscr{D}$-reachability algorithm, it is possible to compute an implementation of M whenever M is executable. Thus given a monitor M the function $\log_{I_P,M}(E)$ can be effectively computed.

4 Generating an attack-preventing monitor

4.1 Attack presentation

In our setting attacks are simply specified as protocol executions without reference to any violated security property. The flexibility entailed by this choice however implies that, in order to prevent the given execution, one also has to provide what should have been the correct execution for the subset of participants involved in the attack. This setting leads to the definition of an attack presentation sharing the same set of participants playing the same roles, but having different traces.

Definition 15. (Attack definition) Let $I_P = (\Xi_P, \Phi_P)$ be a protocol implementation. An *attack definition* on I_P is a tuple $(\Xi_E, \text{tr}_A, \text{tr}_N, R_E)$ such that $(\Xi_E, \text{tr}_A, R_E)$ is an execution of I_P and $(\Xi_E \setminus R_E^{-1}(I), \text{tr}_N, R_E)$ is an honest execution of I_P.

Given an attack definition $(\Sigma_E, \text{tr}_A, \text{tr}_N, R_E)$, $(\Sigma_E, \text{tr}_A, R_E)$ refers to the *attack execution* while $(\Sigma_E, \text{tr}_N, R_E)$ refers to the *normal execution* of the protocol expected for the honest participants involved. Though this is not enforced by the definition and not needed in the rest of this paper, it is expected that the initial segments of the traces corresponding to the initial knowledge and the generation of nonces should be the same for each participant in the two executions.

Definition 16. (Attack presentation) Let $P = (\Xi_P, \text{tr}_P)$ be a protocol, $I_P = (\Xi_P, \Phi_P)$ be an implementation of P, $M = (\Xi_P, \text{tr}_M)$ be a monitor of P, and $\mathscr{A} = (\Xi_E, \text{tr}_A, \text{tr}_N, R_E)$ be an attack definition on I_P. Then the *presentation of $\mathscr{A}$ to M* is the couple $(\log_{I_P,M}((\Xi_E \setminus R_E^{-1}(I), \text{tr}_A, R_E)), \log_{I_P,M}((\Xi_E \setminus R_E^{-1}(I), \text{tr}_N, R_E)))$

Detectable attacks. We note that the two traces in an attack presentation may be equivalent. In this case, no test performed by the monitor could enable it to distinguish between the normal and the attack execution, and the latter would not be preventable. We say that an attack $\mathscr{A}$ is *detectable* by the monitor M if its presentation (Λ, Λ') to M is such that Λ and Λ' are not equivalent.

This definition leads to the problem of deciding whether an attack is detectable by a monitor.

Decision Problem 1. $\mathit{AttackDetectability}_{\mathscr{D}}(s, s')$

Input: The presentation (Λ, Λ') of an attack $\mathscr{A}$ on the protocol implementation I_P with a monitor M;

Output: YES if $\mathscr{A}$ is detectable by M

This problem is related to the classic static equivalence problem by the following theorem, proved in the appendix.

Theorem 3. *Let $\mathscr{D}$ be a deduction system, and $\mathscr{N}$ be the deduction system of Ex. 2. Then $AttackDetectability_{\mathscr{D}\cup\mathscr{N}}$ on strands that do not contain symbols of $\mathscr{C}_{\mathscr{N}}$ is polynomial-time reducible to $StaticEquivalence_{\mathscr{D}}$.*

The latter $StaticEquivalence_{\mathscr{D}}$ decision problem is well-studied and in most cases of deduction systems of interest was found to be decidable, which implies that the $AttackDetectability_{\mathscr{D}\cup\mathscr{N}}$ problem is also decidable for most deduction systems of interest.

4.2 Monitor Synthesis

In our setting an attack definition relies on humans to specify also the intended execution, but this execution is not present when searching whether a concrete execution is an attack. Thus we need to synthesize tests that will detect whether an execution is an attack by relying solely on the contents of the actual execution.

Let (Λ,Λ') be a detectable attack presentation. By definition there exists at least one equation $C_1 \stackrel{?}{=} C_2$ either in P^f_Λ or in $P^f_{\Lambda'}$ that is not satisfied by the other trace. We add it to the tests of the monitor. If the equation is in P^f_s the monitor interrupts the protocol if it is not satisfied, whereas if it is in $P^f_{s'}$ the monitor interrupts the protocol if it is satisfied.

5 Attack detection in practice

We present in this section a simple example, the ISO/IEC 9797-1 protocol, especially its manual authentication mecanism 7a described in [12]. The normal run of the protocol is, after a human user sent D and R to the two devices A and B:

$$\begin{array}{rl} A \textbf{ knows} & A,B,D,R \\ B \textbf{ knows} & A,B,D,R \\ A \to B & : h(A,D,k_A,R) \\ B \to A & : h(B,D,k_B,R) \\ A \to B & : k_A \\ B \to A & : k_B \end{array}$$

A dishonest participant i can sent back the first message directly to a honest participant a willing to play the rôle A, and completely impersonate B during the session:

$$\begin{array}{rl} a \to I & : h(A,D,k_A,R) \\ I \to a & : h(A,D,k_A,R) \\ a \to I & : k_A \\ I \to a & : k_A \end{array}$$

We let $P = (\{A,B\},\mathrm{tr}_P)$ be the definition of the protocol, $E = (\{a,i\},\mathrm{tr}_E,\{a\mapsto A, i\mapsto I\})$ be the execution of the protocol P describing the attack, and $M = (\{A,B\},\mathrm{tr}_M)$ with:

$$\begin{cases} \mathrm{tr}_M(A) & = (?A,?B,?D,?R,?k_A,!h(A,D,k_A,R),?h(A,D,k_B,R),!h(A,D,k_B,R),?k_B) \\ \mathrm{tr}_M(B) & = \mathrm{input}(\mathrm{tr}_P(B)) \end{cases}$$

The implementation of this monitor would be:

$$\Phi_M(A) = \quad (?v_1;?v_2;?v_3;?v_4;!v_6 \text{ with } \left\{x_6 \stackrel{?}{=} h(x_1,x_3,x_5,x_4)\right\};?v_7,!v_8 \text{ with } \left\{x_8 \stackrel{?}{=} x_7\right\})$$

The two logs for the regular execution and the attack are respectively, with this implementation:

$$\begin{cases} (!h(A,D,k_A,R);!h(B,D,k_B,R)) & \text{(normal)} \\ (!h(A,D,k_A,R);!h(B,D,k_A,R)) & \text{(attack)} \end{cases}$$

and the test $x_1 \stackrel{?}{=} x_2$ is satisfied by the log of attack trace but not by the log of the normal execution. Thus the monitor can reject the attack from the log when this equality is satisfied. A more robust monitor would send the last two messages k_A and k_B as in that case we know of no other attack even when keys are guessable or the hash function is weak [5].

6 Conclusion

In future work we plan to generate monitor implementations from several roles, and to study test simplification techniques for efficiency. We also need to extend the monitor construction in order to protect a protocol from all the refinements of an attack.

References

[1] Martín Abadi & Véronique Cortier (2006): *Deciding knowledge in security protocols under equational theories*. *Theor. Comput. Sci.* 367(1-2), pp. 2–32, doi:10.1016/j.tcs.2006.08.032.

[2] Wihem Arsac, Giampaolo Bella, Xavier Chantry & Luca Compagna (2009): *Validating Security Protocols under the General Attacker*. In Pierpaolo Degano & Luca Viganò, editors: *Foundations and Applications of Security Analysis, ARSPA-WITS 2009, York, UK, March 28-29, 2009, Revised Selected Papers*, *Lecture Notes in Computer Science* 5511, Springer, pp. 34–51, doi:10.1007/978-3-642-03459-6_3.

[3] Giampaolo Bella, Stefano Bistarelli & Fabio Massacci (2003): *A Protocol's Life After Attacks...* In Bruce Christianson, Bruno Crispo, James A. Malcolm & Michael Roe, editors: *Security Protocols, 11th International Workshop, Cambridge, UK, April 2-4, 2003, Revised Selected Papers*, *Lecture Notes in Computer Science* 3364, Springer, pp. 3–10, doi:10.1007/11542322_2.

[4] Yannick Chevalier & Michaël Rusinowitch (2010): *Compiling and securing cryptographic protocols*. *Inf. Proc. Lett.* 110(3), pp. 116–122, doi:10.1016/j.ipl.2009.11.004.

[5] Stéphanie Delaune, Steve Kremer & Ludovic Robin (2017): *Formal Verification of Protocols Based on Short Authenticated Strings*. In: *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pp. 130–143, doi:10.1109/CSF.2017.26.

[6] Nachum Dershowitz & David A. Plaisted (2001): *Rewriting*. In: *Handbook of Automated Reasoning*, Elsevier and MIT Press, pp. 535–610, doi:10.1016/B978-044450813-3/50011-4.

[7] Maria-Camilla Fiazza, Michele Peroli & Luca Viganò (2015): *Defending Vulnerable Security Protocols by Means of Attack Interference in Non-Collaborative Scenarios*. *Front. ICT* 2015, doi:10.3389/fict.2015.00011.

[8] Mei Lin Hui & Gavin Lowe (1999): *Safe Simplifying Transformations for Security Protocols*. In: *Proceedings of the 12th IEEE Computer Security Foundations Workshop, CSFW 1999, Mordano, Italy, June 28-30, 1999*, IEEE Computer Society, pp. 32–43, doi:10.1109/CSFW.1999.779760.

[9] Zhiwei Li & Weichao Wang (2012): *Towards the attacker's view of protocol narrations (or, how to compile security protocols)*. In Heung Youl Youm & Yoojae Won, editors: *7th ACM Symposium on Information, Computer and Communications Security, ASIACCS '12, Seoul, Korea, May 2-4, 2012*, ACM, pp. 44–45, doi:10.1145/2414456.2414481.

[10] Gavin Lowe (1996): *Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR*. *Software - Concepts and Tools* 17(3), pp. 93–102, doi:10.1007/3-540-61042-1_43.

[11] Roger M. Needham & Michael D. Schroeder (1978): *Using Encryption for Authentication in Large Networks of Computers*. Commun. ACM 21(12), pp. 993–999, doi:10.1145/359340.359342.

[12] Ludovic Robin (2018): *Vérification formelle de protocoles basés sur de courtes chaines authentifiées. (Formal verification of protocols based on short authenticated strings)*. Ph.D. thesis, University of Lorraine, Nancy, France. Available at https://tel.archives-ouvertes.fr/tel-01767989.

[13] Paul F. Syverson & Catherine A. Meadows (2000): *Dolev-Yao is no better than Machiavelli*. In: *Proceedings of the first* Workshop on Issues in the Theory of Security (WITS'00), doi:10.21236/ADA464936.

[14] F. Javier Thayer, Jonathan C. Herzog & Joshua D. Guttman (1998): *Strand Spaces: Why is a Security Protocol Correct?* In: *Security and Privacy - 1998 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 3-6, 1998, Proceedings*, IEEE Computer Society, pp. 160–171, doi:10.1109/SECPRI.1998.674832.

A Relation with the notion of static equivalence

The notions of equivalence *wrt* the refinement relation and static equivalence are strongly related. The different setting is justified by the different handling of nonces: in [1] contexts can contain any constant, so the secret constants in a trace have to be protected using π-calculus' ν operator, while we disallow non-public constants in contexts, which means that no constants can be used but the ones in the deduction system or those published (explicitly or implicitly) in the sequence of messages. We prove in Theo. 3 that the two notions are identical modulo the generation of new constants with the deduction system $\mathscr{N}$ of Ex. 2.

The *AttackDetectability* problem defined in this paper is new, but it is strongly related to the static equivalence problem. In order to show this relation, let us introduce *frames*, which are strands with a hidden set of constants.

Definition 17. (Frames, [1]) A *frame* is a couple $(\tilde{n}, s)$ where $\tilde{n}$ is a set of constants and s is a positive trace, and is usually denoted $\nu\tilde{n}.s$.

Technically the definition in [1] replaces *positive trace s* by a substitution of domain $x_1,\ldots,x_n$ that we have noted σ_s. The application of a context C on a frame $\nu\tilde{n}.s$ is equal to the application of C on s. We are now ready to define the static equivalence problem for a deduction system $\mathscr{D}$.

Decision Problem 2. *StaticEquivalence*$_{\mathscr{D}}(\varphi_1,\varphi_2)$

Input: Two frames $\varphi_i = \nu\tilde{n}_i.s_i$ for $i = 1,2$

Output: YES if the frames have an equal length and for all pair C_1, C_2 of public contexts and all function $\theta : \mathrm{Var}(C_1) \cup \mathrm{Var}(C_2) \to \{x_1,\ldots,x_n\} \cup (\mathscr{C} \setminus (\tilde{n}_1 \cup \tilde{n}_2))$ we have $C_1\theta\sigma_{s_1} = C_2\theta\sigma_{s_1}$ if, and only if, $C_1\theta\sigma_{s_2} =_{\mathscr{E}} C_2\theta\sigma_{s_2}$.

Attack detectability is related to static equivalence with the following theorem:

Theorem 3. *Let $\mathscr{D}$ be a deduction system, and $\mathscr{N}$ be the deduction system of Ex. 2. Then AttackDetectability$_{\mathscr{D}\cup\mathscr{N}}$ on strands that do not contain symbols of $\mathscr{C}_{\mathscr{N}}$ is polynomial-time reducible to StaticEquivalence$_{\mathscr{D}}$.*

Proof. Given a trace s we let $\varphi_s = \nu\mathrm{Const}(s).[s_1,\ldots,s_n]$. This construction is clearly polynomial time. Let t_1, t_2 be the two traces in the presentation of the attack $\mathscr{A}$ on I_P to M.

First, if t_1 and t_2 are of different length or have different label sequence, one can respond to the *AttackDetectability* in polynomial time. So let us assume the two strands have the same length and the same label sequence. Also, we assume that t_1, t_2 do not contain the symbols of the $\mathscr{N}$ deduction system.

Let us prove that t_1 and t_2 are discernable wrt the deduction system $\mathscr{D} \cup \mathscr{N}$ if, and only if, the frames $\varphi_{s_1}, \varphi_{s_2}$ are not statically equivalent wrt the deduction system $\mathscr{D}$.

First let us assume that the attack presentation (t_1, t_2) is detectable, and wlog assume that t_1 does not refine t_1 for the deduction system $\mathscr{D} \cup \mathscr{N}$. Thus there exists two $\mathscr{D} \cup \mathscr{N}$-contexts such that $C_1 t_1 =_{\mathscr{E}} C_2 t_1$ but $C_1 t_2 \neq_{\mathscr{E}} C_2 t_2$. Since we assume that constants occurring in s_1, s_2 are away from $\mathscr{C}_{\mathscr{N}}$, we construct C_1', C_2' and θ' as follows. For each constant $c \in \mathscr{C}_{\mathscr{N}}$ occurring in C_1 or C_2:

- replace in the contexts c with a new variable x_c;
- define θ as follows:

$$\theta(x) = \begin{cases} x_i & \text{if } x \in \mathrm{Var}(C_1) \cup \mathrm{Var}(C_2) \\ c & \text{if } x = x_c \end{cases}$$

By construction C_1', C_2' are $\mathscr{D}$-public contexts and θ maps each variable of these contexts to either a variable or to a constant away from s_1, s_2. Thus C_1', C_2' and θ' are witnesses that the frames φ_s and $\varphi_{s'}$ are not $\mathscr{D}$-statically equivalent.

Conversely assume that the two frames $\varphi_{s_1}, \varphi_{s_2}$ are not statically equivalent. Then there exist $\mathscr{D}$ contexts C_1, C_2 and $\theta : \mathrm{Var}(C_1) \cup \mathrm{Var}(C_2) \to \mathscr{C} \setminus (\mathrm{Const}(s_1) \cup \mathrm{Const}(s_2))$ such that wlog $C_1 \theta \varphi_{s_1} =_{\mathscr{E}} C_2 \theta \varphi_{s_1}$. Replacing each free constant c by a constant in $\mathscr{C}_{\mathscr{N}}$ yields appropriate contexts for the $\mathscr{D} \cup \mathscr{N}$ attack detectability. □

Sensitive Samples Revisited: Detecting Neural Network Attacks Using Constraint Solvers*

Amel Nestor Docena Thomas Wahl Trevor Pearce Yunsi Fei
Khoury College of Computer Sciences, Northeastern University, Boston, USA
{docena.a|t.wahl|pearce.tr|y.fei}@northeastern.edu

Neural Networks are used today in numerous security- and safety-relevant domains and are, as such, a popular target of attacks that subvert their classification capabilities, by manipulating the network parameters. Prior work has introduced *sensitive samples*—inputs highly sensitive to parameter changes—to detect such manipulations, and proposed a gradient ascent-based approach to compute them. In this paper we offer an alternative, using *symbolic constraint solvers*. We model the network and a formal specification of a sensitive sample in the language of the solver and ask for a solution. This approach supports a rich class of queries, corresponding, for instance, to the presence of certain types of attacks. Unlike earlier techniques, our approach does not depend on convex search domains, or on the suitability of a starting point for the search. We address the performance limitations of constraint solvers by partitioning the search space for the solver, and exploring the partitions according to a balanced schedule that still retains completeness of the search. We demonstrate the impact of the use of solvers in terms of functionality and search efficiency, using a case study for the detection of *Trojan attacks* on Neural Networks.

1 Introduction

Given recent advances in the field of Deep Learning, Neural Networks (DNN)—the preferred data structure for many learning tasks—are used today in numerous application areas, including security- and safety-relevant domains. Their use by unsuspecting end users increasingly makes them the target of attacks that manipulate (a small fraction of) the network parameters, attempting to override the decision-making functionality of the network, at least for some inputs. Examples include the hijacking of image recognition software for access control, and the misguidance of autonomous vehicles. Society has a vital interest in detecting these kind of attacks, in order to mitigate or prevent them.

Inspired by recent work by He et al. [7], we consider in this paper the "cloud scenario": the defender is the designer of the network, with full access to all parameters and the training data. To facilitate widespread use, after training they deploy the network using some DNN inference cloud service, which can, however, ultimately not be trusted. They therefore wish to determine inputs, called *sensitive samples* [7], that are sensitive to parameter manipulations and thus able to distinguish the original, trusted network, N, from a manipulated one, N'.

In the aforementioned recent work, a sensitive sample X^* is defined as an input that maximizes the difference between N and N', resulting in an optimization problem. Provided the sample space is convex, X^* can be found using *(projected) gradient ascent* (PGA). PGA is an efficient technique, but it is also—from a user perspective—somewhat demanding: in addition to the convexity of the sample space, we must compute the differential of the objective function, as well as the projection into the sample space, both of which can be numerically hard problems.

*Partially supported by the US National Science Foundation under grant # SaTC-1929300.

T. Kutsia (Ed.): Symbolic Computation
in Software Science (SCSS'21)
EPTCS 342, 2021, pp. 35–48, doi:10.4204/EPTCS.342.4

The goal of this paper is to cast the task of finding sensitive samples as a Boolean *satisfiability modulo real arithmetic* problem, and use an SMT solver to crack it. Such solvers do not require convex search spaces, and they are black boxes: finding a solution for the specified satisfiability ("sat") problem is left entirely to the solver.

The flexibility of SMT solving does not come for free. Our sat problem locks as follows: given the network N, find a sensitive sample X_s such that for all networks N' s.t. $N' \neq N$, we have $N(X_s) \neq N'(X_s)$. This is in fact a sat problem in a *quantified* logic. We approximate it by restricting the domain of adversarial networks N' to come from some type of attack commonly applied to N, such as a *Trojan attack* [10]. We build a representative Trojan-attack model N' and obtain a sat problem over quantifier-free Boolean logic modulo real arithmetic. Input X_s has qualities resembling a *test case* for detecting the attack: if positive, the attack is present; if negative, we cannot fully guarantee the integrity of the cloud model N'.

The second challenge with using SMT solvers is that, given their symbolic nature, they cannot compete in efficiency with "concrete" evaluation-based solvers like PGA-based search engines. The bottleneck in the SMT solving process is the presence of RELU ("rectified linear unit"; see Sec. A.1) activation functions, which introduce non-linear, non-differentiable arithmetic into the mathematical model of the neural network (see also [8, 2]). We therefore propose in this paper a (generic) *greedy* algorithm to improve the performance of sat-solving formulas over many RELU instances. Our technique factors the RELU functions out of the formula (reducing its complexity substantially), and examines the many *cases* that the combination of RELU functions present in a schedule that is determined *a priori* using a very fast profiling step.

To evaluate our technique, we consider *Trojan attacks*, which turn the trusted model N into the manipulated cloud model N' [10]. We demonstrate that (i) our technique can determine sensitive samples fairly efficiently if used in conjunction with the greedy algorithm mentioned above, that (ii) these samples effectively label Trojanned models as such, thus detecting the attack, and that (iii) *benign* models N' are *not* flagged by our technique, which would constitute a false positive. A benign model N' is one that suffers only harmless inference deviations from N, not to be blamed on an attack, such as due to differences in floating-point arithmetic implementations.

2 Defender Model and Problem Definition

In this work we assume there is a party called *defender* that has access to a trusted trained machine learning model N. The defender seeks to deploy N to some publicly available DNN query service, which we refer to here simply as *the cloud*, to be accessed by *end users* via a narrow DNN query API. The cloud provider has full access to the deployed model ("white-box"); the end user submits inputs to the service and retrieves a classification result in the form of probabilities for each class.

Further, there is a party known as *attacker* (often identical to the cloud provider) set to manipulating the numeric model parameters, i.e., the weights and biases, resulting in a new model N'. The precise goals for such manipulation vary and include altering the network's functionality, for some inputs, e.g., using a Trojan attack [9]. As in prior work [7], we assume that the attacker does not manipulate N's hyper-parameters, e.g., by adding extra layers, or adding neurons to a layer.

We note that, once the defender has deployed the model N, they can access it only via the same narrow interface that is available to standard end users ("black-box"). That is, the details of N' are unknown to them.

Problem definition. We address in this paper the following *idealized* problem for the defender. Given the network N and a type T of network attacks (such as Trojan attacks), determine a *detection threshold* $\bar{\beta}$ and an input X_s called *sensitive sample* [7], such that the following holds for any potential cloud model N': if N and N' disagree in their response to input X_s by at most $\bar{\beta}$, then N' is not the result of an attack of type T against N. In addition, we typically want sample X_s to be "similar" to the inputs given in the training set, so that the use of the sample by the defender to probe N' does not trigger a "spy alarm" by the cloud provider, which could lead to non-uniform treatment of the defender, compared to other end users.

The idea is: if input X_s gives rise to a difference of more than $\bar{\beta}$, networks N and N' disagree non-trivially, which must be reported, using witness X_s. (For this to make sense, we cannot simply choose $\bar{\beta} = 0$; see Sec. 3.2.) Otherwise, we consider the cloud model uncompromised, as far as attacks of type T.

The above problem description is idealistic since it contains an implicit universal quantification over all models N' compromised via a type-T attack. This results in a formula in the expensive (although in principle decidable) SMT theory of quantified non-linear real arithmetic (NRA). In this paper we approximate this problem, by determining an input X_s that is a sensitive sample for a *typical instantiation* N' of a type T-attacked network. This results in a more manageable formula in quantifier-free real arithmetic (QF-NRA). We then check the effectiveness of the sample thus obtained against other network instances. In general, however, we cannot guarantee the integrity of the cloud model in the "otherwise" case in the previous paragraph.

3 Symbolic Specifications of NNs and Sensitive Samples

Our method of choice to tackle the problem defined in the previous section is via logical constraint solvers. This requires formalizing the neural network (NN), the attack type, and the notion of sensitivity in the language of the solver. In the Technical Appendix, we give a background of NNs, plus the inner workings of a Trojan attack.

3.1 Fully-Connected Neural Networks

Consider an L-layer fully-connected NN, (see Technical Appendix, Sec. A.1). We formulate the linear function and subsequent non-linear activation function in each hidden layer $l < L$ as

linear combination: $Z^{[l]} = W^{[l]\intercal} A^{[l-1]}$

hidden layer activation: $A^{[l]} = f(Z^{[l]})$,

where $W^{[l]}, Z^{[l]}, A^{[l]}$ represent the parameters (weights and bias), the linear (pre-activation) output vector, and the activated output vector, respectively. To encode the linear-combination intermediate result Z in the SMT-Lib language [1], we declare it to be an uninterpreted real-valued constant, and then constrain it to be equal to a linear expression over the components of weights w_i and previous-layer activation a_i:

```
(declare-const z_h Real)
(assert (= z_h (+ (* w_1 a_1) (* w_2 a_2)...(* w_n a_n) bias)))
```

For the activation in hidden layers, ReLU is a typical choice. We define it in the SMT-Lib language relationally, as a function `relu`: $\mathbb{R} \times \mathbb{R} \to \{0,1\}$:

```
(define-fun relu ((z Real) (a Real)) Bool
  (= a (ite (<= z 0.0) 0.0 z))) ; intuitively, a = relu(z)
```

For the purpose of adding queries to the encoded neural network, e.g., to retrieve a sensitive sample, we define the output of the network to be the logit computed by the pre-activation output function F, rather than the result of the output activation function σ, which can give rise to complex symbolic encodings. This is feasible because conditions over the final activated value $\sigma(F(X,W))$ can be translated "backwards" to conditions over the logit vector, $Z^{[L]}$. For instance, if $\sigma = \text{sigmoid}$ and we wish to constrain our sample to be classified as label $t = 1$ with probability at least 80%, we translate the condition $\sigma(Z^{[L]}) \geq 0.8$ to the constraint $Z^{[L]} \geq \sigma^{-1}(0.8) = 1.386$ (σ^{-1} is called the *logit function*).

3.2 Sensitive Sample Queries

The *sensitivity* of a sample X, notated β, is measured by the deviation of its prediction when run on a tweaked model from its original prediction. Selecting some parameters of interest $w \in W$ for study, Lee et al. define sensitivity as $\beta = \|\sigma(X,w) - \sigma(X,w+\delta)\|$, for some suitable norm $\|\cdot\|$. A sensitive sample then is an input X^* that maximizes this sensitivity assuming that the w have been tweaked by some δ: $X^* = \text{argmax}_I \|\sigma(I,w) - \sigma(I,w+\delta)\|$ [7]. For this X^*, its corresponding sensitivity β^* is optimal.

In this paper, instead of solving an optimization problem, we determine input samples that give rise to a suitable "target sensitivity". Requiring this sensitivity to be positive is not enough: differences in the compiler, the computation environment, the available hardware and other unknowns (which impact the precise semantics of floating-point arithmetic [6]) will typically cause some deviations in the output between the client's platform and the cloud. In the absence of an attack, these deviations would show up as false positives. We therefore model floating-point vagaries using an application-dependent *detection threshold* $\bar{\beta}$ beyond which any observed sensitivity is blamed on the presence of an attack, while sensitivities below it are assumed to be harmless. We show later in our experiments that such deviations tend to be far smaller than differences due to an attack, making the two quite easily separable via a suitable detection threshold. Thus, an input X_s witnesses the presence of an attack iff its sensitivity $\beta_s \geq \bar{\beta}$.

In contrast to PGA where the sensitivity of a sample is determined when a SS is retrieved, (if the model fails to converge, however, then no optimal SS is retrieved); in our approach, we set a threshold for the sensitivity first and determine whether we can find a satisfying assignment for our prescribed SS. In the following subsection we spell out an attack symbolically to determine a desired X_s, displaying flexible specifications, leading to a case where we detect a Trojan attack.

3.2.1 Symbolically Encoding Sensitive Sample Conditions

Consider an attack on weight parameters whose goal is to manipulate the original value of some target logit $Z_t^{[L]} \in Z^{[L]}$ to a desired value $Z_t^{[L]'}$ and, consequently, to belie the prediction. In detecting this attack, we select $w \in W$ on which we assume weight changes, δ. These w are referred to as the *parameters of interest* (POI) [7], selected based on knowledge about an attack, which are representative of the actual parameters that have been perturbed for detection. We specify the conditions for the sensitive sample as follows:

$$\begin{aligned} X_s = X_0 + \hat{\gamma} \;\wedge\; X_s \in S \;\wedge\; X_s \models C \;\wedge\; Z^{[L]} = F(X_s,W) \wedge \\ Z^{[L]'} = F(X_s, W+\delta) \;\wedge\; \forall \delta_i :\; \delta_i \in [a,b]_i \;\wedge\; \left| Z_t^{[L]'} - Z_t^{[L]} \right| \geq \bar{\varepsilon} \;. \end{aligned} \tag{1}$$

The sensitive sample X_s takes the form $X_s = X_0 + \hat{\gamma}$, where X_0 is a training sample modified by some suitable transform variables $\hat{\gamma}$. We distinguish the variable we seek for satisfiability from the rest of the parameters that the defender defines to bound the search with a hat; in the case of Eq. (1), these are the

$\hat{\gamma}$. While, the variables that are given or defined by the defender are: the training sample, X_0; the sample space S and additional constraints C; the DNN architecture, $F(\cdot)$; the trained weights, W; the weight change vector δ applied to the POI and its assumed range of values; and the logit detection threshold $\bar{\varepsilon}$, which sets the sensitivity of the sample that satisfies the detection threshold. We expound on these conditions further.

By constraining $\hat{\gamma}$ to within a small radius around 0, we force the SS to be close to the training sample X_0; thereby, appear "natural", not artificial, to an input analyzer that may be used by the service provider to detect whether their inference is being monitored or tested. Prior work has used similar mechanisms to enforce similarity of the sensitive sample to the training data [7]. Moreover, the sample space S, which the SS is an element of, can be convex or non-convex, as can be the additional constraints C. Such constraints might state that the sample should be classified into a particular label by the network. Further, when X_s is forward-propagated through the network, the output, expressed as a formula over the logit vector $Z^{[L]}$, must of course agree with the network formula $F(X_s, W)$, a function of X_s and W.

For our assumed tweaked model—$F(X_s, W+\delta)$, a function of X_s, the W and corresponding weight-change vector δ—we are assuming a reasonable range for the weight perturbations δ_i (components of δ) applied to the POI. The choice of POI and range of weight-change is attack-dependent. The actual weight deltas are not known to us, but we know the range of the trained weights. We can therefore get a sense of a plausible range of the deltas based on the nature of the attack (later we present suitable choices for the case of a Trojan attack). Note that we are expressing sensitivity in terms of the logits. We refer to $\bar{\varepsilon}$ as the *satisfying logit threshold*, whose equivalent sensitivity (measured in terms of the output probability, by applying σ to the logits) satisfies the detection threshold $\bar{\beta}$. In Sec. 5, we present an example to configure this metric that models sensitivity given a detection threshold.

So in a nutshell, we wish to solve for $\hat{\gamma}$ such that the sensitive sample X_s captures a bandwidth $\bar{\varepsilon}$ when the POI have been tweaked by δ. In the next specification where we take on a Trojan attack, the parameters assumed as placeholders take shape.

3.2.2 Detecting a Trojan Attack

After a Trojan attack, [9] observed that some weights from the target layer through the output layer will be inflated, causing a jump in the output towards the target masquerade t; while, the rest will be re-adjusted to compensate for the inflation, making the Trojanned model to behave like the original model when the trigger is absent. To devise sensitivity conditions to detect this attack, we translate these observations as a special configuration of Eq. (1), as explained in the following.

We detect whether our model has been Trojanned towards some label t, which we select for scrutiny. As POI for this attack, we choose the weights connected to the output layer (a similar strategy was employed in [7]); the weights from the target layer (which only the attacker knows) all the way up to just before the output layer are assumed unchanged. Among the POI, we let w_e^L be the weights expected to be inflated; while, the rest of the weights potentially deflated as w_d^L. That is: we assume positive weight deltas $\delta_e > 0$ applied to the former weights, and non-positive weight deltas $\delta_d \leq 0$ applied to the latter. The corresponding activated neurons connected by these weights are $A_e^{[L-1]}$ and $A_d^{[L-1]}$, respectively. We formulate the original logit for label t as $Z_t^{[L]} = w_e^{[L]\intercal} A_e^{[L-1]} + w_d^{[L]\intercal} A_d^{[L-1]}$. The perturbed logit, with the corresponding weight deltas, is given by

$$Z_t^{[L]'} = (w_e^{[L]} + \delta_e)^\intercal A_e^{[L-1]} + (w_d^{[L]} + \delta_d)^\intercal A_d^{[L-1]} \ .$$

Since a Trojan attack raises the prediction towards the target masquerade t in the presence of a trigger, we set the difference between the perturbed logit and the original logit to be positive and non-trivial,

$\bar{\varepsilon} > 0$. Upon subtraction, we get $Z_t^{[L]'} - Z_t^{[L]} = \delta_e^\intercal A_e^{[L-1]} + \delta_d^\intercal A_d^{[L-1]} = \delta^\intercal A^{[L-1]} \geq \bar{\varepsilon}$. This suggests that we model our sensitive sample to have a non-trivial net sensitivity on the possible weight perturbations that can occur among the POI in a Trojan attack. With this as sensitivity condition, the sensitive-sample specification from Eq. (1) becomes:

$$\begin{array}{l} X_s = X_0 + \hat{\gamma} \wedge X_s \in S \wedge X_s \models C \wedge Z^{[L]} = F(X_s, W) \wedge \\ Z^{[L]'} = F(X_s, W+\delta) \wedge \delta_d \leq 0 < \delta_e \wedge \delta^\intercal A^{[L-1]} \geq \bar{\varepsilon} \, . \end{array} \tag{2}$$

In Sec. 5, we determine suitable values for these parameters that bound $\hat{\gamma}$ by detecting a real-world Trojan attack. But before that, we devise an algorithm to tackle scalability of this approach in the next section.

4 Improving Scalability using ReLU Profiling

For non-trivial networks, the SMT models designed in Sec. 3 represent hard decision problems. In this section we get to the bottom of the complexity, and design an algorithm to improve the scalability.

4.1 Root-Cause Analysis: Scalability Bottleneck

We analyze the root cause of the scalability problems. As also observed in other work [8], the main culprit is the "branches" that each application of a ReLU activation function represents: they cause the network model to be a piece-wise linear, rather than linear, mathematical function of the inputs. We can (vastly) underapproximate the SMT model for the sample query, by replacing each ReLU activation function by either the identity or the constant-zero function—we say the ReLU node is *fixed as identity* or *fixed as zero*— and simultaneously forcing the inputs to the function to be respectively non-negative or negative:

```
; id: R x R -> Boolean
(define-fun id ((z Real) (a Real)) Bool
  (and (>= z 0) (= a z)))                ; z >= 0 and a = z

; zero: R x R -> Boolean
(define-fun zero ((z Real) (a Real)) Bool
  (and (< z 0) (= a 0))                  ;  z < 0 and a = 0
```

Performing this replacement on one ReLU node roughly cuts the solution space for the solver to explore in half. We now design an algorithm for more efficient sensitive-sample search exploiting these insights, and demonstrate its impact in Sec. 5.

4.2 Greedy ReLU Branch Exploration using Decision Profiling

Solving a query for a sample input intuitively requires exhaustively exploring all combinations of branch decisions that the ReLU nodes might make—ReLU *combinations* for short—for a given input. The number of such combinations is, of course, exponential in the number of ReLU nodes, resulting in poor scalability. A key idea, however, is that we are free to choose the *order* in which the combinations are examined. An optimistic approach might try first ReLU combinations that are deemed "more likely" to permit a satisfying assignment, *easier to solve* for short. Completeness of this approach can be guaranteed by exploring *harder to solve* combinations later, rather than discarding them.

But what makes one RELU combination easier than others? We borrow here the idea of *branch prediction* done by runtime-optimizing compilers: As the owner of the network model and the training data used to obtain it, we can perform an inexpensive *decision profiling* step, which determines, for each RELU node, how often it acts as the identity function, and how often as the zero function, measured over the training data. We call the larger of these two

Node	#IDENT	#ZERO	direction	d-bias
Z1_1	377	623	NUL	62%
Z1_2	220	780	NUL	78%
Z1_3	196	804	NUL	80%
Z1_4	295	705	NUL	70%
Z1_5	828	172	POS	83%
Z2_1	744	256	POS	74%
Z2_2	901	99	POS	90%
Z2_3	0	1000	NUL	100%
Z2_4	898	102	POS	90%
Z2_5	1000	0	POS	100%

numbers, as a percentage of the training data size, the *decision bias*, *d-bias* for short, of a RELU node: a large d-bias towards "identity" suggests the node acts more often as the identity function than as the zero function. The table above illustrates, for a toy network of two hidden layers with 5 neurons each and a dataset of 1000 inputs, the number if times a RELU node acted as identity or as zero, the direction of the bias, and the d-bias value (formally defined in (3) below).

Since the sensitive samples are designed to be (slight) perturbations of the training data, we expect it to be beneficial to assume that the RELU nodes exhibit a branching behavior similar to that over the training data. We say a RELU node has been *fixed* if it has been fixed according to its d-bias (ties resolved in some arbitrary but deterministic way). To *unfix* a (fixed) RELU node means to reinstate the RELU function, in place of the identity or zero function that it had been replaced with.

Equipped with these definitions, we deem a RELU combination easier to solve than another if the RELU nodes that have been fixed in the former form a superset of those fixed in the latter. Additionally, we sort the RELU nodes according to their d-bias and use this ordering to make RELU combinations easier to solve. The motivation is that fixing a RELU node with a large d-bias is more likely to preserve many satisfying solutions than fixing a RELU node with a small d-bias (near 50%).

To formalize these ideas, let $(X_1,\ldots,X_k)$ be an input to the network. Consider a neuron j, and let Z_j be the function that computes the pre-activation value of the neuron, i.e., the input to the RELU function at j. (The output computed at j is therefore $A_j = \text{RELU}(Z_j(X_1,\ldots,X_k))$.) We say j *acts as identity on* $(X_1,\ldots,X_k)$ if $Z_j(X_1,\ldots,X_k) \geq 0$; otherwise j *acts as zero* on this input. For a set $\mathbf{X}$ of network inputs (such as a training set), the *d-bias* of neuron j (a number in $[0.5,1]$) is defined as

$$\textit{d-bias}(j) = \max \left\{ \begin{array}{l} |\{(X_1,\ldots,X_k) \in \mathbf{X} : Z_j(X_1,\ldots,X_k) \geq 0\}|, \\ |\{(X_1,\ldots,X_k) \in \mathbf{X} : Z_j(X_1,\ldots,X_k) < 0\}| \end{array} \right\} \quad / \quad |\mathbf{X}| \,. \tag{3}$$

Alg. 1 implements the RELU-aware search strategy we sketched above. It takes as input a NN model $F(X_s,W)$ along with a training data set $\mathbf{X}$, and a formula ϕ over the model inputs $X_1,\ldots,X_k$. Formula ϕ typically encodes some kind of condition for an input that we are trying to find, e.g., the condition of $X_1,\ldots,X_k$ being a sensitive sample for $F(X_s,W)$. The algorithm begins by computing the d-biases of all RELU nodes over the training set. It then fixes each RELU node according to its d-bias, i.e., it replaces, in ϕ, each RELU activation function by the identity or the zero function, depending on the direction of the bias.

If the modified formula ϕ, which represents an underapproximation of the original ϕ, permits a solution, this solution is genuine, and we return it. Otherwise, we have to weaken the formula, by unfixing one of the fixed RELU nodes. Here we unfix nodes with *small* d-bias first: a small bias means the predictive power of the decision profiling is weak, so unfixing this node may re-enable many solutions. This order heuristic is implemented via the sorting step in Line 3; ties are resolved arbitrarily.

Algorithm 1 Greedy ReLU branch exploration using decision profiling

Input: network model $F(X_s, W)$, training data $\mathbf{X}$, formula ϕ
Output: a model input $X_1, \ldots, X_k$ satisfying ϕ, if one exists; otherwise "unsat"

```
1: compute the d-biases of all ReLU nodes in F(X_s, W) w.r.t. data set X
2: in φ, fix each ReLU node according to its d-bias
3: sort the ReLU nodes in F(X_s, W) in order of increasing d-bias: A_1, ..., A_l
4: for j := 1 to l do
5:     if there exists X_1, ..., X_k: (X_1, ..., X_k) ⊨ φ then
6:         return "solution: X_1, ..., X_k"
7:     unfix ReLU node A_j
8: return "unsat"
```

We emphasize that we have merely used heuristics to determine the *order* in which different ReLU combinations are searched. Theoretical completeness of the algorithm is not affected, since all combinations are, in the worst case, examined. If we were to pass ϕ directly to the SMT solver, we would leave it to the solver to examine these combinations, oblivious to the information offered by the profiling data.

5 Evaluation

We conducted experiments to retrieve samples sensitive against a Trojan attack, as motivated in Sec. 3.2.2, and checked their effectiveness in detecting the attack. We then assessed the scalability of the technique to larger networks, both without and with Alg. 1. We used Microsoft's Z3 as SMT solver. The experiments were run on an Intel Core i7-10750H CPU at 2.60GHz and 16GB of RAM.

5.1 Victim Network

Our benchmarks for detecting a Trojan attack come from the *Belgium Traffic Signs dataset* [11, `https://btsd.ethz.ch/shareddata`]. We re-sized the images to 14x14 pixels and turned them grayscale. We trained a fully-connected NN of dimensions 30x20x10x1 to identify whether a traffic sign indicates a speed limit (label 0) or STOP (label 1). For this mini-image classification task, this NN—despite being small—has 100% validation accuracy, precision, and recall.

5.2 Configuring the parameter space

We partition the sensitive-sample parameter space into *attack parameters* and *sample parameters*. Our configuration was defined independently of the attack simulation.

Attack parameters. They model the Trojan attack and include the following:

Assumed target masquerade, t: STOP-sign label.

POI, $w \in W$: weights attached to the output layer.

Weight deltas, δ: We assume the top weights, in terms of value, to be inflated and few, since the attack is supposedly stealthy; the rest are assumed potentially deflated, (i.e., non-positive delta). For this experiment where the trained weights are within $[0, 1]$, we assumed 30% are inflated by $[0.05, 0.25]$ units; (we are not setting a larger sub-range since we are modelling an attack that is stealthy).

Sample parameters. They include the sample space and the detection threshold. We further added as constraint the predicted label of the sample on the original model.

Sample space: In order to make the sensitive sample appear like a regular input, we randomly picked a speed limit sign X_0 from the test data (see Fig. 1 (left)) and added transform variables, $\hat{\gamma}$, over the entire region of the pixel dimensions. The SS is the pointwise sum of the pixel values of X_0 and the assignment to the transform variables.

Predicted output label: We explicitly required that the originally predicted output label remain as a speed limit sign even after the SS transformation.

Detection threshold, $\bar{\beta}$, and the satisfying logit threshold, $\bar{\varepsilon}$: We first stipulated a detection threshold that is significant to warrant weight perturbations: we set $\bar{\beta} = 0.01$; that is, a sensitivity of 1 percentage point in probability output or more is attributed to an attack in model parameters. Based on this requirement, we determined $\bar{\varepsilon}$ by computing the initial logit and then setting a satisfying logit threshold. By forward propagation on the original network, we computed the logit of the random sample X_0 to be $Z^{[4]} = -4.8510$. In this experiment, we set $\bar{\varepsilon} = 1$, which models a tweaked logit $Z^{[4]\prime} \geq -3.8510$. The corresponding modeled sensitivity under this setting is $\beta \geq 0.0131$, which satisfies the detection threshold.

5.3 Effectiveness of SS

After solving for the transform values, we tested the obtained sensitive sample in detecting the Trojanned model. We deem the sample effective if the observed sensitivity is at least the detection threshold. If it is below, this would be a false negative. Furthermore, to assess the possibility of false positives, we subjected the original model to minor perturbations as they may occur "innocently" on the cloud. Concretely, we compared the prediction of the sensitive samples on a version where model parameters are originally stored in *float16* precision to one where the parameters are stored in *float32*. If the sensitivity to such innocent perturbations (SIP) remains below the detection threshold, then the sample does not represent a false positive.

5.4 Results

Naive approach: This approach passes Eq. (2) to the SMT solver (w/o Alg. 1). After solving for the transform variables (which took about one minute), we obtained the sample shown in Fig. 1 (right): it has a sensitivity of $\beta = 0.0744$ (summarized in the first row of Table 2 in the Appendix). This sensitivity means that the prediction of the Trojanned model is 7.44 percentage points away from the original prediction. Moreover, given $SIP = 7.41\text{E-}06$, the sample does not exhibit a false positive.

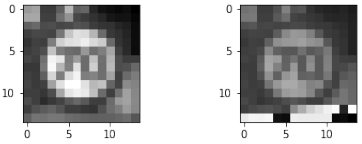


Figure 1: Randomly taken test sample (left) and transformed, *sensitive* sample (right)

Scaling of (2) to larger networks: Sec. B of the Technical Appendix shows results of attempting to scale the naive approach to larger networks. Sec. C shows raw size data for various SMT encodings.

Employing Alg. 1: We allotted 20/60/120sec as the increasing time per iteration of the **for** loop in Line 4, before the algorithm moves on to the next RELU combination. This reflects the greedy character of the algorithm: we want to first try many instantiations of RELU nodes as "identity" or "zero", believing that one of them will give us a solution. If this approach fails, we increase the time allotment per iteration. This serves the goal of delaying advancing to later iterations, with fewer fixed RELU nodes, as much as possible, which eventually degenerates the algorithm to one that simply has the solver explore the many RELU combinations. In Table 1, the iteration number where a satisfying assignment was finally found is given in column "**iter. #**". Small numbers here indicate success of the philosophy purported by Alg. 1. Column "**runtime**" shows the total time it took to find a satisfying assignment, i.e., the sum across all iterations, for a respective iteration timeout of 20/60/120sec. If a solution is found for an iteration timeout of 20sec, one would normally stop the algorithm. In order to evaluate scalability, we include results for larger time allotments. But despite the extension, the solver wounded up with the same result as with the initial allocation.

Network size	iter. #	sensitivity	SIP	total runtime (s)
30x20x10x1	6	0.1234	1.08E-05	111/312/ 612
40x20x20x1	4	0.8342	2.99E-05	65/185/ 365
50x30x10x1	14	0.3520	1.33E-05	264/784/1564
50x30x20x1	12	0.0249	4.00E-05	230/670/1330
60x40x20x1	2	0.9859	5.04E-06	36/ 75/ 135
90x60x30x1	—	—	—	(out of resources)

Table 1: Search with ReLU profiling algorithm

Comparison naive approach/Alg. 1: Consider the case of the 30x20x10x1 NN model. Alg. 1 found a satisfying assignment in iteration #6. This means that the activation functions of five neurons—those with the smallest d-bias—were freed from their fixed instantiation to the identity or zero functions, and reverted to exact RELU semantics. This combination yielded a sensitivity of 12.34 percentage points, a higher sensitivity than the solution found by the naive approach. The time reported for Alg. 1's 20s/iter. run (111s) is larger though. This is not surprising, since Alg. 1 "wastes" five SMT runs. In fact, the motivation for using this algorithm is not to find solutions in "easy" cases faster. It is, instead, to increase scalability to larger models. Indeed, Alg. 1 was able to solve all models except the 90x60x30x1 case, while the naive approach timed out for most.

Presently, the method is not able to handle deeper and larger networks, such as state-of-the-art convolutional neural networks. Nevertheless, the DNN models that we presented are valid networks; we showed that SMT solvers can be used for such NN queries, e.g., sensitive-sample generation, when appropriately formalized. To deal with the complexity of DNNs, research has been conducted into dedicated theories for NN queries, e.g., Reluplex [8]. Using a dedicated solver may potentially address the scalability issue further: we first use Alg. 1 to eliminate some ReLU nodes, (e.g. the ones with high bias), while others are left in. For those left in, Reluplex can be applied. The investigation of this possibility can be picked up in future work.

6 Related Work

This work was inspired by the *sensitive-sample fingerprinting* technique proposed by He et al. [7], which uses classic machine learning techniques based on (projected) gradient ascent to determine sensitive samples. This is very efficient, but requires a starting point for the search and a *convex* solution space. Lack of convexity can lead to sub-optimality or, worse, failure to converge. The goal here was to solve a similar problem, but bring the flexibility of symbolic constraint solvers to bear: we can specify any search space and sample conditions, as long as they are definable in the underlying logic. However, definability does not imply efficient processibility, which is why Sec. 4 presents an algorithm for improved satisfiability checking.

Solving an optimization problem, the gradient-based technique returns samples that maximize the sensitivity. The authors conclude that any output discrepancy confirms the presence of an attack ("guarantee[s] zero false positives" [7]). As discussed in Sec. 3.2, this is not quite true: due to platform-dependencies of floating-point computations [6], DNN model inference is *not* deterministic. We address this problem using an empirical non-zero sensitivity detection threshold (Sec. 2).

Using symbolic techniques in deep learning is still a relatively young area; examples include [5, 12]. The Reluplex SMT solver [8] introduces a theory of real arithmetic extended by the RELU function as a primitive operation. We can view our greedy Alg. 1 as an alternative dedicated way of handling RELU nodes. A stand-alone comparison of both methods is left for future work.

Our method needs to be instantiated for different attack types (to avoid an unrealistic universal quantification over "all" adversarial models). We have focused on Trojan attacks, a survey can be found in [10]. Strategies for formalizing other types of DNN attacks are left as future work. "Fingerprinting", using inputs characteristic of model manipulations, is one way of detecting such attacks; there are others. For instance, we can conclude the model has been compromised if, for some class, the minimal trigger that causes a misclassification is substantially smaller than for other classes and thus is likely supported by a Trojan [13]. In that approach, the defender does not need access to the trusted model or the training data. The approach is designed specifically for backdoor-style attacks relying on a trigger. Other methods perform statistical analyses, e.g., determining the probability distribution of potential triggers [3] or of prediction results under perturbations [4]. Such analyses may not be realistic in a cloud environment.

7 Summary

We revisited the technique of retrieving sensitive samples in detecting NN attacks. A previous approach solves an optimization problem, using an efficient projected gradient-based search, in order to find the optimal sensitivity [7], which faces, however, various technical preconditions and is somewhat rigid. Our approach performs the search via a symbolic constraint solver. This permits a flexible specification of desirable features of the sample. We argue that a sample need not be optimal to be effective, as long as its sensitivity is above a threshold that delineates it from innocent perturbations that can occur upon upload of the NN model to the cloud. This alternative comes with the price of efficiency, however. To address scalability, we devised a greedy algorithm that searches through all possible combinations of the ReLU node behaviors in an "easiest-first" order. This algorithm has applications in symbolic NN analysis beyond sensitive-sample search. Future work includes investigating the performance impact of using dedicated solvers for NN queries, such as Reluplex [8].

References

[1] Clark Barrett, Pascal Fontaine & Cesare Tinelli (2016): *The Satisfiability Modulo Theories Library (SMT-LIB)*. `www.SMT-LIB.org`.

[2] James Bornholt: *Can you train a neural network using an SMT solver?* `https://www.cs.utexas.edu/~bornholt/post/nnsmt.html` (access May 2021).

[3] Huili Chen, Cheng Fu, Jishen Zhao & Farinaz Koushanfar (2019): *DeepInspect: A Black-box Trojan Detection and Mitigation Framework for Deep Neural Networks*. In: *International Joint Conference on Artificial Intelligence (IJCAI)*, doi:10.24963/ijcai.2019/647.

[4] Yansong Gao, Change Xu, Derui Wang, Shiping Chen, Damith Chinthana Ranasinghe & Surya Nepal (2019): *STRIP: a defence against trojan attacks on deep neural networks*. In: *Annual Computer Security Applications Conference (ACSAC)*, doi:10.1145/3359789.3359790.

[5] Mirco Giacobbe, Thomas A. Henzinger & Mathias Lechner (2020): *How Many Bits Does it Take to Quantize Your Neural Network?* In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, doi:10.1007/978-3-030-45237-7_5.

[6] Yijia Gu, Thomas Wahl, Mahsa Bayati & Miriam Leeser (2015): *Behavioral Non-portability in Scientific Numeric Computing*. In: *International Conference on Parallel and Distributed Computing (Euro-Par)*, doi:10.1007/978-3-662-48096-0_43.

[7] Zecheng He, Tianwei Zhang & Ruby B. Lee (2019): *Sensitive-Sample Fingerprinting of Deep Neural Networks*. In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, doi:10.1109/CVPR.2019.00486.

[8] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian & Mykel J. Kochenderfer (2017): *Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks*. In: *Computer Aided Verification (CAV)*, doi:10.1007/978-3-319-63387-9_5.

[9] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang & Xiangyu Zhang (2018): *Trojaning Attack on Neural Networks*. In: *Network and Distributed System Security (NDSS)*, The Internet Society, doi:10.14722/ndss.2018.23291.

[10] Yuntao Liu, Ankit Mondal, Abhishek Chakraborty, Michael Zuzak, Nina Jacobsen, Daniel Xing & Ankur Srivastava (2020): *A Survey on Neural Trojans*. In: *21st International Symposium on Quality Electronic Design (ISQED)*, doi:10.1109/ISQED48828.2020.9137011.

[11] Markus Mathias, Radu Timofte, Rodrigo Benenson & Luc Van Gool (2013): *Traffic sign recognition – How far are we from the solution?* In: *International Joint Conference on Neural Networks (IJCNN)*, doi:10.1109/IJCNN.2013.6707049.

[12] Luca Pulina & Armando Tacchella (2012): *Challenging SMT solvers to verify neural networks*. *AI Commun.*, doi:10.3233/AIC-2012-0525.

[13] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng & Ben Y. Zhao (2019): *Neural Cleanse: Identifying and Mitigating Backdoor Attacks in Neural Networks*. In: *2019 IEEE Symposium on Security and Privacy (S&P)*, doi:10.1109/SP.2019.00031.

Technical Appendix

A Background

A.1 Neural Networks

A Neural Network (NN) is a parameterized, layered function that maps a vector X of features from some n-dimensional input space to an output y, which may be discrete if the task is classification, or real if

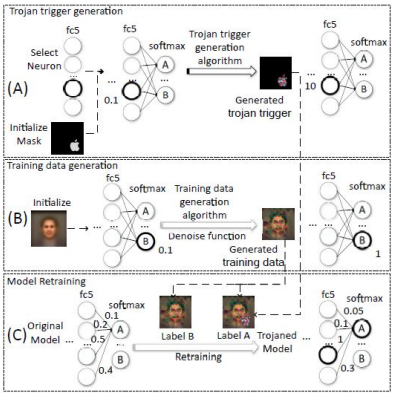
regression. Given parameters W (a matrix of weights and biases), each layer consists of a linear function l_i over the layer's inputs and its weight parameters, and a subsequent non-linear activation function f_i. Function l_i is a simple dot product, translated by the bias, for a fully-connected layer. The activation for inner layers is typically implemented via the *rectified linear unit*, defined as $\text{RELU}(b) = \max\{b, 0\}$. The activated values become the inputs of the next layer. The result of the final linear function, known as *logit*, i.e., computed at the output layer, is passed to a special activation function σ. For a binary classification task, the output activation is typically a sigmoid function; for a multi-label classification, it is a softmax function. Output y is then a set of probabilities indicating which among the labels the input X most likely belongs to. For a regression task, the output activation is a linear function, which yields an output over the real numbers. Thus given a network of L layers, we summarize the function computed by the NN as

$$y \;=\; \sigma(l_L(f_{L-1}(\ldots f_1(l_1(X,W))\ldots))) \;=\; \sigma(F(X,W)),$$

where F denotes the pre-activation output, i.e., the logit value computed for input X.

A.2 Trojan Attack on Neural Networks

A *Trojan* attack on NN perturbs model parameters in order to cause a misclassification towards a target label on inputs with an embedded trigger; the modified network is said to be *Trojanned*. But, when the Trojanned model is presented with inputs without the trigger, the prediction is unchanged. This makes the attack hard to detect by unsuspecting users.



There are three steps to Trojan a neural network (see figure on the right, taken from [9]). The attacker is assumed to be able to modify the model parameters. First, the attacker generates a Trojan trigger, which is a snippet embedded to a test input that excites certain neurons so that the prediction is skewed towards a target masquerade. This trigger is initialized with a mask or shape. [9] suggests that the target layer from which to establish a link with the trigger is somewhere near the middle layer, wherein the neurons that are most connected are selected.

The authors define weight connection of a neuron as the L1-norm of the weights connected to the neuron from the previous layer. The values of the initialized trigger are set by optimizing the activation values of the select neurons to intended values, supposedly large. This establishes a strong connection between the trigger and the select neurons, so that in the presence of the former the latter have strong activations, which would influence the model prediction.

After generating the trigger, the attacker stamps it to some training samples for each of the output labels, wherein the label for all these stamped samples is the target masquerade. If the attacker does not have access to the training data, they can generate the latter by model inversion. The batch of samples to be used for retraining the NN would consist of both unstamped and stamped samples. This last step of retraining the original NN Trojans the network, such that the weights are re-adjusted so that the new model predicts the target masquerade whenever the trigger is present in a test input, but predicts normally when absent, making the attack stealthy.

B Scalability of naive sample search via SMT

Using the same training data and basic network architecture, we trained larger networks, by expanding the layer sizes. We then simulated a Trojan attack on each of these. In order to assess scalability, we kept all SS parameters as set earlier. (In practice, one might determine individual SS parameters for each network.) Although we were able to transform sensitive samples in some networks, Table 2 shows the poor scaling of the naive approach to larger networks, especially where the size of the layers close to the output increases.

Network size	sensitivity	SIP	runtime
30x20x10x1	0.0744	7.41E-06	71s
40x20x20x1	—	—	(out of resources)
50x30x10x1	0.0595	8.85E-07	4
50x30x20x1	—	—	(out of resources)
60x40x20x1	—	—	(out of resources)
90x60x30x1	—	—	(out of resources)

Table 2: Sensitivities and running time for sample search without RELU profiling

C Raw size data for encodings of various networks

To convey an idea of the raw complexity of the SMT approach, Table 3 shows various size data of the network models and the SMT encoding (2) of the SS search.

NN dimensions	# NN param.	# RELU nodes	SMT: LOC	SMT: file size
30x20x10x1	6,751	60	7,853	354kB
40x20x20x1	9,141	80	10,363	468kB
50x30x10x1	11,701	90	12,943	592kB
50x30x20x1	12,021	100	13,333	613kB
60x40x20x1	15,101	120	16,503	770kB
90x60x30x1	25,051	180	26,753	1,284kB

Table 3: Complexity of SMT approach: NN dimensions, number of weights and biases, number of ReLU nodes, lines of code of the SMT encoding, and file size

Querying RDF Databases with Sub-CONSTRUCTs

Dominique Duval
LJK - Univ. Grenoble Alpes
dominique.duval@univ-grenoble-alpes.fr

Rachid Echahed
LIG - Univ. Grenoble Alpes
rachid.echahed@imag.fr

Frédéric Prost
LIG - Univ. Grenoble Alpes
frederic.prost@univ-grenoble-alpes.fr

Graph query languages feature mainly two kinds of queries when applied to a graph database: those inspired by relational databases which return tables such as SELECT queries and those which return graphs such as CONSTRUCT queries in SPARQL. The latter are object of study in the present paper. For this purpose, a core graph query language GrAL is defined with focus on CONSTRUCT queries. Queries in GrAL form the final step of a recursive process involving so-called GrAL patterns. By evaluating a query over a graph one gets a graph, while by evaluating a pattern over a graph one gets a set of matches which involves both a graph and a table. CONSTRUCT queries are based on CONSTRUCT patterns, and sub-CONSTRUCT patterns come for free from the recursive definition of patterns. The semantics of GrAL is based on RDF graphs with a slight modification which consists in accepting isolated nodes. Such an extension of RDF graphs eases the definition of the evaluation semantics, which is mainly captured by a unique operation called Merge. Besides, we define aggregations as part of GrAL expressions, which leads to an original local processing of aggregations.

1 Introduction

Graph database query languages are becoming ubiquitous. In contrast to classical relational databases where SQL language is a standard, different languages [1] have been proposed for querying graph databases, like SPARQL [9] or Cypher [5]. Among the most popular models for representing graph databases, one may quote for instance the *sets of triples* (or *RDF graphs* [10]) used by SPARQL or the *property graphs* used by Cypher. In addition to the lack of a standard model to represent graph databases, there are different kinds of queries in the context of graph query languages. One may essentially distinguish two classes of queries: those inspired by relational databases which return tables such as SELECT queries and those which return graphs such as CONSTRUCT queries in SPARQL. Such CONSTRUCT queries are graph-to-graph queries specific to graph databases.

The graph-to-graph queries received less attention than the graph-to-table queries. For instance, for the SPARQL language, a semantics of SELECT queries and subqueries is proposed in [6], a semantics of CONSTRUCT queries in [7] and a semantics of CONSTRUCT queries with nested CONSTRUCT queries in FROM clauses in [2, 8], where the outcome of a subCONSTRUCT is a graph. All these works consider graphs as sets of triples. Unfortunately, such a definition of graphs prevents having a uniform semantics of all patterns and in particular for BIND patterns.

In this paper, we focus on graph-to-graph queries and subqueries for RDF graphs and we propose a new semantics for CONSTRUCT subqueries which departs from the one in [2, 8]. First, we propose to change slightly the definition of graphs by allowing isolated nodes. Indeed, this new definition of graphs allows us to have a uniform semantics of all patterns. In fact, we define CONSTRUCT *subpatterns* rather

T. Kutsia (Ed.): Symbolic Computation
in Software Science (SCSS'21)
EPTCS 342, 2021, pp. 49–64, doi:10.4204/EPTCS.342.5

than CONSTRUCT subqueries. For this purpose, we introduce a core query language GrAL based on RDF graphs. The syntactic categories of GrAL include both queries and patterns. When evaluating a CONSTRUCT query over a graph one gets a graph, whereas when evaluating a CONSTRUCT pattern over a graph one gets a set of matches which involves both variable assignments and a graph. In fact, a CONSTRUCT query first acts as a CONSTRUCT pattern and then returns only the constructed graph. As the definition of patterns is recursive, CONSTRUCT subpatterns are obtained for free.

In order to define the semantics of GrAL, we introduce an algebra of operations over sets of matches, where a match is a morphism between graphs. We propose to base the semantics of GrAL upon an algebra on sets of matches, like the semantics of SQL is based upon relational algebra. All operations in our algebra essentially derive from a unique operation called *Merge*, which generalizes the well-known *Join* operation. As stated earlier, we consider graphs consisting of classical RDF triples possibly augmented with some additional isolated nodes. This slight extension helps formulating the semantics of patterns and queries without using some cumbersome notations to handle, for instance, environments defined by variable bindings. The proposed algebra is used to define an evaluation semantics for GrAL. As for aggregations, they are handled locally inside expressions. The semantics of the various patterns and queries is uniform, as it is based on instances of the *Merge* operation.

The paper is organized as follows. Section 2 introduces the algebra designed to express the semantics of the query language GrAL. In Section 3, the language GrAL is defined by its syntax and semantics. Concluding remarks are given in Section 4.

2 The Graph Query Algebra

The Graph Query Algebra is a family of operations which are used in Section 3 for defining the evaluation of queries in the Graph Algebraic Query Language GrAL. Graphs and matches are introduced in Section 2.1, then operations on sets of matches are defined in Section 2.2.

2.1 Graphs and matches

In this paper, graphs are kinds of generalised RDF graphs that may contain isolated nodes. Let $\mathscr{L}$ be a set, called the set of *labels*, union of two disjoint sets $\mathscr{C}$ and $\mathscr{V}$, called respectively the set of *constants* and the set of *variables*.

Definition 2.1 (graph)**.** Every element $t = (s, p, o)$ of $\mathscr{L}^3$ is called a *triple* and its members s, p and o are called respectively the *subject*, *predicate* and *object* of t. A *graph* X is made of a subset X_N of $\mathscr{L}$ called the set of *nodes* of X and a subset X_T of $\mathscr{L}^3$ called the set of *triples* of X, such that the subject and the object of each triple of X are nodes of X. The nodes of X which are neither a subject nor an object are called the *isolated nodes* of X. The set of *labels* of a graph X is the subset $\mathscr{L}(X)$ of $\mathscr{L}$ made of the nodes and predicates of X, then $\mathscr{C}(X) = \mathscr{C} \cap \mathscr{L}(X)$ and $\mathscr{V}(X) = \mathscr{V} \cap \mathscr{L}(X)$. Given two graphs X_1 and X_2, the graph X_1 is a *subgraph* of X_2, written $X_1 \subseteq X_2$, if $(X_1)_N \subseteq (X_2)_N$ and $(X_1)_T \subseteq (X_2)_T$, then obviously $\mathscr{L}(X_1) \subseteq \mathscr{L}(X_2)$. The *union* $X_1 \cup X_2$ is the graph defined by $(X_1 \cup X_2)_N = (X_1)_N \cup (X_2)_N$ and $(X_1 \cup X_2)_T = (X_1)_T \cup (X_2)_T$, then $\mathscr{L}(X_1 \cup X_2) = \mathscr{L}(X_1) \cup \mathscr{L}(X_2)$.

We will not use the *intersection* $X_1 \cap X_2$, which could be defined by $(X_1 \cap X_2)_N = (X_1)_N \cap (X_2)_N$ and $(X_1 \cap X_2)_T = (X_1)_T \cap (X_2)_T$: then the intersection of two graphs without isolated nodes might have isolated nodes and $\mathscr{L}(X_1 \cap X_2)$ might be strictly smaller than $\mathscr{L}(X_1) \cap \mathscr{L}(X_2)$, as for instance when $X_1 = \{(x,y,z)\}$ and $X_2 = \{(y,z,x)\}$ so that $X_1 \cap X_2 = \{x\}$.

Example 2.2. We introduce here a toy database representing a simplified view of a social media network. We will use it as a running example to illustrate various definitions. The network consists in *authors publishing messages*. Each message is *timestamped at* a certain *date* (a day). A message can *refer to* other messages and an author may *like* a message. An instance of such a network is described by the following graph G_0 (written "à la" RDF):

$G_0 =$
```
auth1 publishes mes1 .  auth1 publishes mes2 .
auth2 publishes mes3 .  auth3 publishes mes4 .  auth3 publishes mes5 .
mes1 stampedAt date1 .  mes2 stampedAt date2 .
mes3 stampedAt date1 .  mes4 stampedAt date4 .  mes5 stampedAt date4 .
mes3 refersTo mes1 .  mes4 refersTo mes1 .  mes4 refersTo mes2 .
auth1 likes mes3 .  auth1 likes mes4 .  auth1 likes mes5 .
auth2 likes mes1 .  auth2 likes mes4
```

The meaning of G_0 is that author `auth1` has published messages `mes1` and `mes2`, which have been stamped respectively at dates `date1` and `date2`, etc.

Definition 2.3 (match)**.** A *match* m from a graph X to a graph G, denoted $m : X \to G$, is a function from $\mathscr{L}(X)$ to $\mathscr{L}(G)$ which *preserves nodes* and *preserves triples* and which *fixes* $\mathscr{C}$, in the sense that $m(X_N) \subseteq G_N$, $m^3(X_T) \subseteq G_T$ and $m(x) = x$ for each x in $\mathscr{C}(X)$. The set of all matches from X to G is denoted $Match(X,G)$. An *isomorphism* of graphs is an invertible match.

When n is an isolated node of X then the node $m(n)$ does not have to be isolated in G. A match $m : X \to G$ determines two functions $m_N : X_N \to G_N$ and $m_T : X_T \to G_T$, restrictions of m and m^3 respectively. A match $m : X \to G$ is an isomorphism if and only if both functions $m_N : X_N \to G_N$ and $m_T : X_T \to G_T$ are bijections. This means that a function m from $\mathscr{L}(X)$ to $\mathscr{L}(G)$ is an isomorphism of graphs if and only if $\mathscr{C}(X) = \mathscr{C}(G)$ with $m(x) = x$ for each $x \in \mathscr{C}(X)$ and m is a bijection from $\mathscr{V}(X)$ to $\mathscr{V}(G)$: thus, X is the same as G up to variable renaming. It follows that the symbol used for naming a variable does not matter as long as graphs are considered only up to isomorphism.

Definition 2.4 (image of a graph by a function)**.** Let X be a graph. Every function $f : \mathscr{V}(X) \to \mathscr{L}$ can be extended in a unique way as a function $f' : \mathscr{L}(X) \to \mathscr{L}$ that fixes $\mathscr{C}$. The *image* $f(X)$ of X by f is the graph made of the nodes $f'(n)$ for $n \in X_N$ and the triples $(f')^3(t)$ for $t \in X_T$. The function f can be extended in a unique way as a match $f^\sharp : X \to f(X)$.

Definition 2.5 (compatible matches)**.** Two matches $m_1 : X_1 \to G_1$ and $m_2 : X_2 \to G_2$ are *compatible*, written as $m_1 \sim m_2$, if $m_1(x) = m_2(x)$ for each $x \in \mathscr{V}(X_1) \cap \mathscr{V}(X_2)$. Given two compatible matches $m_1 : X_1 \to G_1$ and $m_2 : X_2 \to G_2$, let $m_1 \bowtie m_2 : X_1 \cup X_2 \to G_1 \cup G_2$ denote the unique match such that $m_1 \bowtie m_2 \sim m_1$ and $m_1 \bowtie m_2 \sim m_2$ (which means that $m_1 \bowtie m_2$ coincides with m_1 on X_1 and with m_2 on X_2).

We will see in Section 3 that the execution of a query in GrAL is a graph-to-graph transformation, which main part is a graph-to-set-of-matches transformation.

Definition 2.6 (set of matches, assignment table)**.** Let X and G be graphs. A set $\underline{m}$ of matches, all of them from X to G, is denoted $\underline{m} : X \Rightarrow G$. The *assignment table* $Tab(\underline{m})$ of $\underline{m}$ is the two-dimensional table with the elements of $\mathscr{V}(X)$ in its first row, then one row for each m in $\underline{m}$, and the entry in row m and column x equals to $m(x)$.

Thus, the assignment table $Tab(\underline{m})$ describes the set of functions $\underline{m}|_{\mathscr{V}(X)} : \mathscr{V}(X) \Rightarrow \mathscr{L}$, made of the functions $m|_{\mathscr{V}(X)} : \mathscr{V}(X) \to \mathscr{L}$ for all $m \in \underline{m}$. The set of matches $\underline{m} : X \Rightarrow G$ is determined by the graphs X and G and the assignment table $Tab(\underline{m})$. This property is used hereafter to describe some examples.

Example 2.7. Here are some examples of matches in the graph G_0 (defined in Example 2.2).

- Let P_{ps} be the following graph (written "à la" SPARQL: variable names begin with "?"):

$$P_{ps} = \boxed{\texttt{?a publishes ?m . ?m stampedAt ?d}}$$

The set $\underline{m}_{ps}$ of all matches from P_{ps} to G_0 is:

$\underline{m}_{ps} : P_{ps} \Rightarrow G_0$ with $Tab(\underline{m}_{ps}) =$

?a	?m	?d
auth1	mes1	date1
auth1	mes2	date2
auth2	mes3	date1
auth3	mes4	date4
auth3	mes5	date4

- Let P_{pl} be the graph:

$$P_{pl} = \boxed{\texttt{?a1 publishes ?m . ?a2 likes ?m}}$$

The set $\underline{m}_{pl}$ of all matches from P_{pl} to G_0 is:

$\underline{m}_{pl} : P_{pl} \Rightarrow G_0$ with $Tab(\underline{m}_{pl}) =$

?a1	?m	?a2
auth1	mes1	auth2
auth2	mes3	auth1
auth3	mes4	auth1
auth3	mes4	auth2
auth3	mes5	auth1

- Let P'_{pl} be the following subgraph of P_{pl}, made of two isolated nodes:

$$P'_{pl} = \boxed{\texttt{?a1 . ?a2}}$$

The subset $\underline{m}'_{pl}$ of $\underline{m}_{pl}$ made of the restrictions to P'_{pl} of the matches in $\underline{m}_{pl}$ is:

$\underline{m}'_{pl} : P'_{pl} \Rightarrow G_0$ with $Tab(\underline{m}'_{pl}) =$

?a1	?a2
auth1	auth2
auth2	auth1
auth3	auth1
auth3	auth2

- Let P_{prp} be the graph:

$$P_{prp} = \boxed{\texttt{?a1 publishes ?m1 . ?m1 refersTo ?m2 . ?a2 publishes ?m2}}$$

The set $\underline{m}_{prp}$ of all matches from P_{prp} to G_0 is:

$\underline{m}_{prp} : P_{prp} \Rightarrow G_0$ with $Tab(\underline{m}_{prp}) =$

?a1	?m1	?m2	?a2
auth2	mes3	mes1	auth1
auth3	mes4	mes1	auth1
auth3	mes4	mes2	auth1

Definition 2.8 (image of a graph by a set of functions)**.** The *image* of a graph X by a set of functions $\underline{f}$ from $\mathscr{V}(X)$ to $\mathscr{L}$, denoted $\underline{f}(X)$, is the graph union of the graphs $f(X)$ for every f in $\underline{f}$. Every set of functions $\underline{f} : \mathscr{V}(X) \Rightarrow \mathscr{L}$ can be extended in a unique way as a set of matches $\underline{f}^\sharp : X \Rightarrow \underline{f}(X)$.

Remark 2.9 (about RDF graphs)**.** RDF graphs [10] are graphs (as in Definition 2.1) without isolated nodes, where constants are either IRIs (Internationalized Resource Identifiers) or literals and where all predicates are IRIs and only objects can be literals. Blank nodes in RDF graphs are the same as variable nodes in our graphs. Thus an isomorphism of RDF graphs, as defined in [10], is an isomorphism of graphs as in Definition 2.3.

2.2 Operations on sets of matches

In this Section we introduce some operations on sets of matches which are used in Section 3 for defining the semantics of GrAL. The prominent one is the *merging* operation (Definition 2.10), which is a kind of generalized *joining* operation (see Definition 2.14). Other basic operations are the simple *restriction* and *extension* operations (Definitions 2.12 and 2.13). Then, these basic operations are combined in order to get some derived operations (Definition 2.14).

Definition 2.10 (Merge)**.** Let $\underline{m} : X \Rightarrow G$ be a set of matches and $\underline{p}_m : Y \Rightarrow H_m$ a family of sets of matches indexed by $m \in \underline{m}$, and let $H = \cup_{m\in\underline{m}} H_m$. The *merging* of $\underline{m}$ along the family $(\underline{p}_m)_{m\in\underline{m}}$ is the set of matches $m \bowtie p$ for every $m \in \underline{m}$ and every $p \in \underline{p}_m$ compatible with m:
$Merge(\underline{m}, (\underline{p}_m)_{m\in\underline{m}}) = \{m \bowtie p \mid m \in \underline{m} \wedge p \in \underline{p}_m \wedge m \sim p\} : X \cup Y \Rightarrow G \cup H$.

Let $\underline{q} = Merge(\underline{m}, (\underline{p}_m)_{m\in\underline{m}})$, then $\underline{q}$ is made of a match $m \bowtie p$ for each pair (m, p) with $m \in \underline{m}$ and $p \in \underline{p}_m$ compatible with m (so that for each m in $\underline{m}$ the number of $m \bowtie p$ in $\underline{q}$ is between 0 and $Card(\underline{p}_m)$). The match $m \bowtie p : X \cup Y \to G \cup H$ is such that $m \bowtie p(x) = m(x)$ when $x \in X$ and $m \bowtie p(y) = p(y)$ when $y \in Y$, which is unambiguous because of the compatibility condition.

Example 2.11. Here are some examples of merging, based on the sets of matches in Example 2.7.

- Let $\underline{p}'_{pl}$ be similar to $\underline{m}'_{pl}$, up to renaming the variables `?a1` and `?a2` as `?a2` and `?a1`, respectively, so that $\underline{p}'_{pl}$ is:

$$\underline{p}'_{pl} : P'_{pl} \Rightarrow G_0 \text{ with } Tab(\underline{p}'_{pl}) =$$

?a1	?a2
auth2	auth1
auth1	auth2
auth1	auth3
auth2	auth3

  For each match m in $\underline{m}'_{pl}$ let $\underline{p}_m = \underline{p}'_{pl}$, which does not depend on m. Then $Merge(\underline{m}'_{pl}, (\underline{p}_m)_{m\in\underline{m}'_{pl}})$ is denoted simply $Join(\underline{m}'_{pl}, \underline{p}'_{pl})$ (as in Definition 2.14 and Remark 2.15) and:

$$\underline{q}'_{pl} = Join(\underline{m}'_{pl}, \underline{p}'_{pl}) : P'_{pl} \Rightarrow G_0 \text{ with } Tab(\underline{q}'_{pl}) =$$

?a1	?a2
auth1	auth2
auth2	auth1

- Assume that there is some operation *concat* that builds a string from any given date and string. For each match m in $\underline{m}_{ps}$ let $\underline{p}_m = \{p_m\} : \{?dm\} \Rightarrow H_{ps}$ where $p_m(?dm) = concat(m(?d), m(?m))$ and

H_{ps} is any graph that contains all strings $concat(?d,?m)$ as nodes. Then:

$$\underline{q}_{ps} = Merge(\underline{m}_{ps}, (\underline{p}_m)_{m\in\underline{m}_{ps}}) : P_{ps} \cup \{?dm\} \Rightarrow G_0 \cup H_{ps}$$

with $Tab(\underline{q}_{ps}) =$

?a	?m	?d	?dm
auth1	mes1	date1	date1mes1
auth1	mes2	date2	date2mes2
auth2	mes3	date1	date1mes3
auth3	mes4	date4	date4mes4
auth3	mes5	date4	date4mes5

- For each match m in $\underline{m}_{ps}$ let $\underline{p}_m = \{p_m\} : \{?r\} \Rightarrow H_{var}$ where $p_m(?r)$ is some fresh variable $?r_m$ and H_{var} is any graph that contains all variables $?r_m$ as nodes. Then:

$$\underline{q}_{var} = Merge(\underline{m}_{ps}, (\underline{p}_m)_{m\in\underline{m}_{ps}}) : P_{ps} \cup \{?r\} \Rightarrow G_0 \cup H_{var}$$

with $Tab(\underline{q}_{var}) =$

?a	?m	?d	?r
auth1	mes1	date1	?r1
auth1	mes2	date2	?r2
auth2	mes3	date1	?r3
auth3	mes4	date4	?r4
auth3	mes5	date4	?r5

Definition 2.12 (Restrict)**.** Let $\underline{m} : X \Rightarrow G$ be a set of matches. For every graph Y contained in X and every graph H contained in G such that $\underline{m}(Y) \subseteq H$, the *restriction* $Restrict(\underline{m},Y,H) : Y \Rightarrow H$ is made of the restrictions of the matches in $\underline{m}$ as matches from Y to H. When $H = G$ the notation may be simplified: $Restrict(\underline{m},Y) = Restrict(\underline{m},Y,G) : Y \Rightarrow G$.

Definition 2.13 (Extend)**.** Let $\underline{m} : X \Rightarrow G$ be a set of matches. For every graph H containing G, the *extension* $Extend(\underline{m},H) : X \Rightarrow H$ is made of the extensions of the matches in $\underline{m}$ as matches from X to H.

New operations are obtained by combining the previous ones (assuming that *true* is a constant). Comments on Definition 2.14 are given in Remark 2.15. We will see in Section 3.2 that these derived operations provide the semantics of the operators of the language GrAL.

Definition 2.14 (derived operations)**.**

- For every sets of matches $\underline{m} : X \Rightarrow G$ and $\underline{p} : Y \Rightarrow H$, let $\underline{p}_m = \underline{p}$ for each $m \in \underline{m}$, then:
  $Join(\underline{m},\underline{p}) = Merge(\underline{m},(\underline{p}_m)_{m\in\underline{m}}) : X \cup Y \Rightarrow G \cup H$.
- For every set of matches $\underline{m} : X \Rightarrow G$, every family of constants $\underline{c} = (c_m)_{m\in\underline{m}}$ and every variable x, let $\underline{p}_m = \{p_m\}$ and $p_m(x) = c_m$ for each $m \in \underline{m}$, then:
  $Bind(\underline{m},\underline{c},x) = Merge(\underline{m},(\underline{p}_m)_{m\in\underline{m}}) : X \cup \{x\} \Rightarrow G \cup \underline{c}$.
- For every set of matches $\underline{m} : X \Rightarrow G$ and every family of constants $\underline{c} = (c_m)_{m\in\underline{m}}$, for some fresh variable x, let $\underline{true} = (true)_{m\in\underline{m}}$:
  $Filter(\underline{m},\underline{c}) = Restrict(Bind(Bind(\underline{m},\underline{c},x),\underline{true},x),X,G) : X \Rightarrow G$.
- For every set of matches $\underline{m} : X \Rightarrow G$ and every graph R, for every $m \in \underline{m}$ let $p_m : R \to p_m(R)$ be the match such that:
  $p_m(x) = m(x)$ if $x \in \mathcal{V}(R) \cap \mathcal{V}(X)$
  and $p_m(x) = var(x,m)$ is a fresh variable if $x \in \mathcal{V}(R) \setminus \mathcal{V}(X)$
  and let $\underline{p}_m = \{p_m\}$ and $\underline{p}(R) = \cup_{m\in\underline{m}}(p_m(R))$, then:
  $Construct(\underline{m},R) = Restrict(Merge(\underline{m},(\underline{p}_m)_{m\in\underline{m}}),R) : R \Rightarrow G \cup \underline{p}(R)$.

- For every sets of matches $\underline{m} : X \Rightarrow G$ and $\underline{p} : X \Rightarrow H$:
  $Union(\underline{m}, \underline{p}) = Extend(\underline{m}, G \cup H) \cup Extend(\underline{p}, G \cup H) : X \Rightarrow G \cup H$.

Remark 2.15. Let us analyse these definitions. Note that the definition of *Bind* and *Filter* rely on the fact that isolated nodes are allowed in graphs.

- Operation *Join* is *Merge* when the set of matches $\underline{p}_m$ does not depend on m, so that:
  $Join(\underline{m}, \underline{p}) = \{m \bowtie p \mid m \in \underline{m} \wedge p \in \underline{p} \wedge m \sim p\} : X \cup Y \Rightarrow G \cup H$.
  It follows that *Join* is commutative.

- Operation *Bind* is *Merge* when $\underline{p}_m$ has exactly one element p_m for each m, which is such that $p_m(x) = c_m$. There are two cases:
  - If $x \in \mathscr{V}(X)$ then this operation selects the matches m in $\underline{m}$ such that $m(x) = c_m$:
    $Bind(\underline{m}, \underline{c}, x) = \{m \mid m \in \underline{m} \wedge m(x) = c_m\} : X \Rightarrow G$.
  - If $x \notin \mathscr{V}(X)$ then this operation extends each match m in $\underline{m}$ by assigning the value c_m to the variable x. Let us denote the resulting match as $m \uplus (x \mapsto c_m)$, so that:
    $Bind(\underline{m}, \underline{c}, x) = \{m \uplus (x \mapsto c_m) \mid m \in \underline{m}\} : X \cup \{x\} \Rightarrow G \cup \{\underline{c}\}$.

- Operation *Filter* applies *Bind* twice, first when $x \notin \mathscr{V}(X)$ for extending each $m \in \underline{m}$ by assigning c_m to x, then since $x \in \mathscr{V}(X \cup \{x\})$ for selecting the matches m in $\underline{m}$ such that $c_m = true$. Now the value of the auxiliary variable x is always *true*, so that x can be dropped: this is the role of the last step which restricts the domain of the matches from $X \cup \{x\}$ to X and its range from $G \cup \{\underline{c}\}$ to G.

- The first step in operation *Construct* is *Merge* when $\underline{p}_m$ has exactly one element p_m for each m (as for *Bind*), which is determined by $p_m(x) = var(x, m)$ for each variable x in R that does not occur in X. Each $var(x, m)$ is a fresh variable, which means that it is distinct from the variables in G, X and R, and that the variables $var(x, m)$ are pairwise distinct. Note that the precise symbol used for denoting $var(x, m)$ does not matter. The second step in operation *Construct* restricts the domain of the matches from $X \cup R$ to R. Thus:
  $Construct(\underline{m}, R)$ is the set of matches from R to $G \cup \underline{p}(R)$
  determined by the functions $f_m : \mathscr{V}(R) \to \mathscr{L}$ (for each $m \in \underline{m}$) such that
  $f_m(x) = m(x)$ if $x \in \mathscr{V}(R) \cap \mathscr{V}(X)$ and $f_m(x) = var(x, m)$ if $x \in \mathscr{V}(R) \setminus \mathscr{V}(X)$.
  Thus, the graph $G \cup \underline{p}(R)$ is obtained by "gluing" one copy of G with $Card(\underline{m})$ copies of R in the right way. Note that the functions f_m are pairwise distinct when $\mathscr{V}(R)$ is not included in $\mathscr{V}(X)$, but it needs not be the case in general. Also, note that the domain R of $Construct(\underline{m}, R)$ may be quite different from the domain X of $\underline{m}$, whereas every other operation in Definition 2.14 either keeps or extends the domain of $\underline{m}$.

- Operation *Union* is simply the set-theoretic union of sets of matches which share the same domain (by assumption) and the same range (by extending the range if necessary). This operation differs from the previous ones in the sense that it is not defined by examining the matches in its arguments. Note that *Union* is commutative.

Proposition 2.16. *The sets of matches obtained by the operations previously defined in this Section have bounded cardinals, as follows.*

$Card(Merge(\underline{m},(\underline{p}_m)_{m\in\underline{m}})) \leq \sum_{m\in\underline{m}}(Card(\underline{p}_m))$
$Card(Restrict(\underline{m},X,G)) \leq Card(\underline{m})$
$Card(Extend(\underline{m},H)) = Card(\underline{m})$
$Card(Join(\underline{m},\underline{p})) \leq Card(\underline{m}) \times Card(\underline{p})$
$Card(Bind(\underline{m},\underline{c},x)) = Card(\underline{m})$
$Card(Filter(\underline{m},\underline{c})) \leq Card(\underline{m})$
$Card(Construct(\underline{m},R)) \leq Card(\underline{m})$
$Card(Union(\underline{m},\underline{p})) \leq Card(\underline{m}) + Card(\underline{p})$

The proof of Proposition 2.16 follows easily from the definitions.

3 The Graph Algebraic Query Language

In this Section we introduce the syntax and semantics of the Graph Algebraic Query Language GrAL. There are three syntactic categories in GrAL: *expressions*, *patterns* and *queries*. Expressions are considered in Section 3.1. Patterns are defined in Section 3.2, their semantics is presented as an evaluation function which maps every pattern P and graph G to a set of matches $[[P]]_G$. Queries are defined in Section 3.3, they are essentially specific kinds of patterns and their semantics is easily derived from the semantics of patterns, the main difference is that the execution of a query on a graph returns simply a graph instead of a set of matches.
To each expression e or pattern P is associated a set of variables called its *in-scope variables* and denoted $\mathcal{V}(e)$ or $\mathcal{V}(P)$, respectively. An expression e is *over* a pattern P if $\mathcal{V}(e) \subseteq \mathcal{V}(P)$. In this Section, as in Section 2, the set of *labels* $\mathcal{L}$ is the union of the disjoint sets $\mathcal{C}$ and $\mathcal{V}$, of *constants* and *variables* respectively. We assume that the set $\mathcal{C}$ of constants contains the numbers and strings and the boolean values *true* and *false*, as well as a symbol $\perp$ for errors.

3.1 Expressions

The expressions of GrAL are built from the labels using operators, which are classified as either basic operators (unary or binary) and aggregation operators (always unary). Remember that typing constraints are not considered in this paper. Typically, and not exclusively, the sets Op_1, Op_2 and Agg of *basic unary* operators, *basic binary* operators and *aggregation* operators can be:
$Op_1 = \{-, \text{NOT}\}$,
$Op_2 = \{+,-,\times,/,=,>,<,\text{AND},\text{OR}\}$,
$Agg = Agg_{elem} \cup \{agg \text{ DISTINCT} \mid agg \in Agg_{elem}\}$.
where $Agg_{elem} = \{\text{MAX},\text{MIN},\text{SUM},\text{AVG},\text{COUNT}\}$
A *group of expressions* is a non-empty finite list of expressions.

Definition 3.1 (syntax of expressions)**.** The *expressions* e of GrAL and their set of *in-scope* variables $\mathcal{V}(e)$ are defined recursively as follows:

- A constant $c \in \mathcal{C}$ is an expression with $\mathcal{V}(c) = \emptyset$.
- A variable $x \in \mathcal{V}$ is an expression with $\mathcal{V}(x) = \{x\}$.
- If e_1 is an expression and $op \in Op_1$ then $op\ e_1$ is an expression with $\mathcal{V}(op\ e_1) = \mathcal{V}(e_1)$.
- If e_1 and e_2 are expressions and $op \in Op_2$ then $e_1\ op\ e_2$ is an expression with $\mathcal{V}(e_1\ op\ e_2) = \mathcal{V}(e_1) \cup \mathcal{V}(e_2)$.

- If e_1 is an expression and $agg \in Agg$ then $agg(e_1)$ is an expression with $\mathcal{V}(agg(e_1)) = \mathcal{V}(e_1)$.
- If e_1 is an expression, $agg \in Agg$ and gp a group of expressions with all its variables distinct from the variables in e_1, then $agg(e_1 \text{ BY } gp)$ is an expression with $\mathcal{V}(agg(e_1 \text{ BY } gp)) = \mathcal{V}(e_1)$.

The *value* of an expression with respect to a set of matches $\underline{m}$ (Definition 3.2) is a family of constants $ev(\underline{m}, e) = (ev(\underline{m}, e)_m)_{m \in \underline{m}}$ indexed by the set $\underline{m}$. Each constant $ev(\underline{m}, e)_m$ depends on e and m and it may also depend on other matches in $\underline{m}$ when e involves aggregation operators. The *value* of a group of expressions $gp = (e_1, ..., e_k)$ with respect to $\underline{m}$ is the list $ev(\underline{m}, gp)_m = (ev(\underline{m}, e_1), ..., ev(\underline{m}, e_k))$. To each basic operator op is associated a function $[[op]]$ (or simply op) from constants to constants if op is unary and from pairs of constants to constants if op is binary. To each aggregation operator agg in Agg is associated a function $[[agg]]$ (or simply agg) from *multisets* of constants to constants. Note that each family of constants determines a multiset of constants: for instance a family $\underline{c} = (c_m)_{m \in \underline{m}}$ of constants indexed by the elements of a set of matches $\underline{m}$ determines the multiset of constants $\{|c_m \mid m \in \underline{m}|\}$, which is also denoted $\underline{c}$ when there is no ambiguity. Some aggregation operators agg in Agg_{elem} are such that $[[agg]](\underline{c})$ depends only on the set underlying the multiset $\underline{c}$, which means that $[[agg]](\underline{c})$ does not depend on the multiplicities in the multiset $\underline{c}$: for instance this is the case for MAX and MIN but not for SUM, AVG, COUNT. When $agg = agg_{elem}$ DISTINCT with agg_{elem} in Agg_{elem} then $[[agg]](\underline{c})$ is $[[agg_{elem}]]$ applied to the underlying set of $\underline{c}$. For instance, COUNT $(\underline{c})$ counts the number of elements of the multiset $\underline{c}$ with their multiplicies, while COUNT DISTINCT $(\underline{c})$ counts the number of distinct elements in $\underline{c}$.

Definition 3.2 (evaluation of expressions)**.** Let X be a graph, e an expression over X and $\underline{m} : X \Rightarrow Y$ a set of matches. The *value* of e with respect to $\underline{m}$ is the family of constants $ev(\underline{m}, e) = (ev(\underline{m}, e)_m)_{m \in \underline{m}}$ defined recursively as follows (with notations as in Definition 3.1):

- $ev(\underline{m}, c)_m = c$.
- $ev(\underline{m}, x)_m = m(x)$.
- $ev(\underline{m}, op\ e_1)_m = [[op]]\, ev(\underline{m}, e_1)_m$.
- $ev(\underline{m}, e_1\ op\ e_2)_m = ev(\underline{m}, e_1)_m\, [[op]]\, ev(\underline{m}, e_2)_m$.
- $ev(\underline{m}, agg(e_1))_m = [[agg]](ev(\underline{m}, e_1))$ (which is the same for every m in $\underline{m}$).
- $ev(\underline{m}, agg(e_1\ BY\ gp))_m = [[agg]](ev(\underline{m}|_{gp,m}, e_1))$ where $\underline{m}|_{gp,m}$ is the subset of $\underline{m}$ made of the matches m' in $\underline{m}$ such that $ev(\underline{m}, gp)_{m'} = ev(\underline{m}, gp)_m$ (which is the same for every m and m' in $\underline{m}$ such that $ev(\underline{m}, gp)_m = ev(\underline{m}, gp)_{m'}$).

Example 3.3. Let $\underline{m}_{pl}$ be as in Example 2.7. For every m in $\underline{m}_{pl}$ we have:

$$ev(\underline{m}_{pl}, \text{COUNT } (likes))_m = 5$$

whereas $ev(\underline{m}_{pl}, \text{COUNT } (likes \text{ BY } ?a1))_m$ depends on the match m in $\underline{m}_{pl}$:

$$ev(\underline{m}_{pl}, \text{COUNT } (likes \text{ BY } ?a1))_m = \begin{cases} 1 & \text{when } m(?a1) = \texttt{auth1} \\ 1 & \text{when } m(?a1) = \texttt{auth2} \\ 3 & \text{when } m(?a1) = \texttt{auth3} \end{cases}$$

Definition 3.4 (equivalence of expressions)**.** Two expressions over a graph X are *equivalent* if they have the same value with respect to every set of matches $\underline{m} : X \Rightarrow Y$.

3.2 Patterns

In Definition 3.5 the patterns of GrAL are built from graphs by using five operators: JOIN, BIND, FILTER, CONSTRUCT and UNION. In Definition 3.6 the semantics of patterns is given by an evaluation function. Some patterns have an associated graph called a *template*, such a pattern P may give rise to a query Q as explained in Section 3.3, then the result of query Q is built from the template of P.

Definition 3.5 (syntax of patterns)**.** The *patterns* P of GrAL, their set of *in-scope* variables $\mathscr{V}(P)$ and their *template* graph $\mathscr{T}(P)$ when it exists are defined recursively as follows.

- A graph is a pattern, called a *basic pattern*, and $\mathscr{V}(P)$ is the set of variables of the graph P.
- If P_1 and P_2 are patterns then P_1 JOIN P_2 is a pattern and $\mathscr{V}(P_1 \text{ JOIN } P_2) = \mathscr{V}(P_1) \cup \mathscr{V}(P_2)$.
- If P_1 is a pattern, e an expression over P_1 and x a variable then P_1 BIND e AS x is a pattern and $\mathscr{V}(P_1 \text{ BIND } e \text{ AS } x) = \mathscr{V}(P_1) \cup \{x\}$.
- If P_1 is a pattern and e an expression over P_1 then P_1 FILTER e is a pattern and $\mathscr{V}(P_1 \text{ FILTER } e) = \mathscr{V}(P_1)$.
- If P_1 is a pattern and R a graph then P_1 CONSTRUCT R,
  also written CONSTRUCT R WHERE P_1, is a pattern and $\mathscr{V}(P_1 \text{ CONSTRUCT } R) = \mathscr{V}(R)$.
  In addition this pattern has a template $\mathscr{T}(P_1 \text{ CONSTRUCT } R) = R$.
- If P_1 and P_2 are patterns with template and if $\mathscr{T}(P_1) = \mathscr{T}(P_2) = R$ then
  P_1 UNION P_2 is a pattern and $\mathscr{V}(P_1 \text{ UNION } P_2) = \mathscr{V}(R)$.
  In addition this pattern has a template $\mathscr{T}(P_1 \text{ UNION } P_2) = \mathscr{T}(P_1) = \mathscr{T}(P_2)$.

The *value* of a pattern over a graph is a set of matches, as defined now.

Definition 3.6 (evaluation of patterns)**.** The *value* of a pattern P of GrAL over a graph G is a set of matches $[[P]]_G : [P] \Rightarrow G^{(P)}$ from a graph $[P]$ that depends only on P to a graph $G^{(P)}$ that contains G. This value $[[P]]_G : [P] \Rightarrow G^{(P)}$ is defined inductively as follows (with notations as in Definition 3.1):

- If P is a basic pattern then $[[P]]_G = \mathit{Match}(P, G) : P \Rightarrow G$.
- $[[P_1 \text{ JOIN } P_2]]_G = \mathit{Join}([[P_1]]_G, [[P_2]]_{G^{(P_1)}}) : [P_1] \cup [P_2] \Rightarrow G^{(P_1)^{(P_2)}}$.
- $[[P_1 \text{ BIND } e \text{ AS } x]]_G = \mathit{Bind}([[P_1]]_G, \mathit{ev}([[P_1]]_G, e), x) : [P_1] \cup \{x\} \Rightarrow G^{(P_1)} \cup [[P_1]]_G(e)$.
- $[[P_1 \text{ FILTER } e]]_G = \mathit{Filter}([[P_1]]_G, \mathit{ev}([[P_1]]_G, e)) : [P_1] \Rightarrow G^{(P_1)}$.
- $[[P_1 \text{ CONSTRUCT } R]]_G = \mathit{Construct}([[P_1]]_G, R) : R \Rightarrow G^{(P_1)} \cup [[P_1]]_G(R)$.
- $[[P_1 \text{ UNION } P_2]]_G = \mathit{Union}([[P_1]]_G, [[P_2]]_{G^{(P_1)}}) : R \Rightarrow G^{(P_1)^{(P_2)}}$ where $R = \mathscr{T}(P_1) = \mathscr{T}(P_2)$.

Remark 3.7. Note that, syntactically, each operator OP builds a pattern P from a pattern P_1 and a parameter *param*, which is either a pattern P_2 (for JOIN and UNION), a pair (e, x) made of an expression and a variable (for BIND), an expression e (for FILTER) or a graph R (for CONSTRUCT). Semantically, for every non-basic pattern $P = P_1$ OP *param*, we denote $\underline{m}_1 : X_1 \Rightarrow G_1$ for $[[P_1]]_G : [P_1] \Rightarrow G^{(P_1)}$ and $\underline{m} : X \Rightarrow G'$ for $[[P]]_G : [P] \Rightarrow G^{(P)}$. In every case it is necessay to evaluate $\underline{m}_1$ before evaluating $\underline{m}$: for JOIN and UNION this is because pattern P_2 is evaluated on G_1, for BIND and FILTER because expression e is evaluated with respect to $\underline{m}_1$, and for CONSTRUCT because of the definition of *Construct*. According to Definition 3.6 given a pattern P and a graph G, the value $\underline{m} : X \Rightarrow G'$ of P is determined as follows:

- When P is a basic pattern then $X = P$, $G' = G$ and $\underline{m}$ is made of all matches from P to G.

- $P = P_1$ OP *param* then the semantics of P is easily derived from Definition 2.14 (see also Remark 2.15). However, note that the semantics of P_1 JOIN P_2 and P_1 UNION P_2 is not symmetric in P_1 and P_2 in general, unless $G^{(P_1)} = G$ and $G^{(P_2)} = G$, which occurs when P_1 and P_2 are basic patterns.
- The graph $G^{(P)}$ is built by adding to G "whatever is required" for the evaluation, in examples we often avoid its precise description.

Given a non-basic pattern $P = P_1$ OP *param*, the pattern P_1 is a *subpattern* of P, as well as P_2 when $P = P_1$ JOIN P_2 or $P = P_1$ UNION P_2. The semantics of patterns is defined in terms of the semantics of its subpatterns (and the semantics of its other arguments, if any). Thus, for instance, CONSTRUCT patterns can be nested at any depth.

Proposition 3.8. *For every pattern P, the set $\mathcal{V}(P)$ of in-scope variables of P is the same as the set $\mathcal{V}([P])$ of variables of the graph $[P]$.*

Example 3.9. In each item below we consider first some pattern P_i and some template R_i, then the pattern

$$C_i = P_i \text{ CONSTRUCT } R_i \text{ also written as } C_i = \text{CONSTRUCT } R_i \text{ WHERE } P_i \,.$$

We refer to Examples 2.7 and 3.3.

- ```
  C1 = CONSTRUCT { ?a1 cites ?a2 }
       WHERE { ?a1 publishes ?m1 . ?m1 refersTo ?m2 . ?a2 publishes ?m2 }
  ```
  Here, $C_1 = P_1$ CONSTRUCT R_1 where $P_1 = P_{prp}$, so that the value of P_1 over G_0 is $\underline{m}_{prp} : P_1 \Rightarrow G$. Note that $\mathcal{V}(R_1) \subseteq \mathcal{V}(P_1)$. Let $G_1 = G_0 \cup \{$`auth2 cites auth1 . auth3 cites auth1`$\}$, the value of C_1 over G_0 is:

$$[[C_1]]_{G_0} : R_1 \Rightarrow G_1 \text{ with } Tab([[C_1]]_{G_0}) =$$

?a1	?a2
auth2	auth1
auth3	auth1

- ```
  C2 = CONSTRUCT { ?n }
       WHERE {
         ?a likes ?m
         BIND COUNT(likes) AS ?n }
  ```
  Here, $C_2 = P_2$ CONSTRUCT R_2 where the template R_2 is the graph made of only one isolated node which is the variable $?n$. The graph $[P_2]$ is $\{$`?a likes ?m . ?n`$\}$. Let $G_2 = G_0 \cup \{5\}$, the value of C_2 over G_0 is:

$$[[P_2]]_{G_0} : [P_2] \Rightarrow G_2 \text{ with } Tab([[P_2]]_{G_0}) =$$

?a	?m	?n
auth1	mes3	5
auth1	mes4	5
auth1	mes5	5
auth2	mes1	5
auth2	mes4	5

  Then the graph $[C_2]$ is $\{$`?n`$\}$ and the value of C_2 over G_0 is:

$$[[C_2]]_{G_0} : [C_2] \Rightarrow G_2 \text{ with } Tab([[C_2]]_{G_0}) =$$

?n
5

-
```
C3 = CONSTRUCT { ?a1 nbOfLikes ?n }
     WHERE { ?a1 publishes ?m . ?a2 likes ?m
       FILTER (NOT(?a1=?a2))
       BIND COUNT(likes BY ?a1) AS ?n }
```

Here, $C_3 = P_3$ CONSTRUCT R_3 where $R_3 = \{$?a1 nbOfLikes ?n$\}$ is made of one triple and $[P_3] = \{$?a1 publishes ?m . ?a2 likes ?m . ?n$\}$. The evaluation of C_3 over G_0 starts from $\underline{m}_{pl}$. Let $G_3 = G_0 \cup \{$auth1 nbOfLikes 1 . auth2 nbOfLikes 1 . auth3 nbOfLikes 3$\}$, the value of C_3 over G_0 is:

$[[P_3]]_{G_0} : [P_3] \Rightarrow G_3$ with $Tab([[P_3]]_{G_0}) =$

?a1	?m	?a2	?n
auth1	mes1	auth2	1
auth2	mes3	auth1	1
auth3	mes4	auth1	3
auth3	mes4	auth2	3
auth3	mes5	auth1	3

Then the graph $[C_3]$ is $R_3 = \{$?a1 nbOfLikes ?n$\}$ and the value of C_3 over G_0 is:

$[[C_3]]_{G_0} : [C_3] \Rightarrow G_3$ with $Tab([[C_3]]_{G_0}) =$

?a1	?n
auth1	1
auth2	1
auth3	3

-
```
C4 = CONSTRUCT { ?a1 nbOfFriends ?n }
     WHERE
     {
       CONSTRUCT { ?a1 friend ?a2 }
       WHERE {
         ?a1 publishes ?m1 . ?a2 likes ?m1 .
         ?a2 publishes ?m2 . ?a1 likes ?m2
       }
       BIND COUNT (friend BY ?a1) AS ?n
     }
```

Here $C_4 = P_4$ CONSTRUCT R_4 where P_4 itself contains a subpattern $C'_4 = P'_4$ CONSTRUCT R'_4. The evaluation of the basic pattern P'_4 over G_0 gives

$[[P'_4]]_{G_0} : P'_4 \Rightarrow G_0$ with $Tab([[P'_4]]_{G_0}) =$

?a1	?m1	?a2	?m2
auth1	mes1	auth2	mes3
auth2	mes3	auth1	mes1

Then we get $[C'_4] = \{$?a1 friend ?a2$\}$ and:

$[[C'_4]]_{G_0} : [C'_4] \Rightarrow {G_0}^{(C'_4)}$ with $Tab([[C'_4]]_{G_0}) =$

?a1	?a2
auth1	auth2
auth2	auth1

Finally $[C_4] = \{$?a1 NbOfFriends ?n$\}$ and:

$[[C_4]]_{G_0} : [C_4] \Rightarrow {G_0}^{(C_4)}$ with $Tab([[C_4]]_{G_0}) =$

?a1	?n
auth1	1
auth2	1

- ```
  C5 = CONSTRUCT { ?r author ?a . ?r date ?d }
       WHERE { ?a publishes ?m . ?m stampedAt ?d }
  ```

  Here $C_5 = P_5$ CONSTRUCT R_5 with a variable $?r$ in R_5 that does not occur in P_5. The value of the basic pattern P_5 over G_0 is:

$$[[P_5]]_{G_0} : P_5 \Rightarrow G_0 \text{ with } Tab([[P_5]]_{G_0}) =$$

?a	?m	?d
auth1	mes1	date1
auth1	mes2	date2
auth2	mes3	date1
auth3	mes4	date4
auth3	mes5	date4

  Then $[C_5] = R_5$ and, since $[[P_5]]_{G_0}$ is made of 5 matches, the value $[[C_5]]_{G_0}$ is obtained by gluing 5 copies of R_5, with 5 fresh variables corresponding to different renamings of variable $?r$. Indeed, the variable $?r$ in R_5, which is not a variable of P_5, gives rise to one fresh variable for each match of P_5 in G_0. Thus:

$$[[C_5]]_{G_0} : R_5 \Rightarrow {G_0}^{(C_5)} \text{ with } Tab([[C_5]]_{G_0}) =$$

?r	?a	?d
?r1	auth1	date1
?r2	auth1	date2
?r3	auth2	date1
?r4	auth3	date4
?r5	auth3	date4

Definition 3.10 (equivalence of patterns)**.** Two patterns are *equivalent* if they have the same value over G for every graph G.

Proposition 3.11. *For every basic patterns P_1 and P_2, the basic pattern $P_1 \cup P_2$ is equivalent to P_1* JOIN *P_2 and to P_2* JOIN *P_1.*

3.3 Queries

A query in GrAL is essentially a pattern which has a template. The main difference between patterns and queries is that, while a pattern is interpreted as a function from graphs to sets of matches, a query is interpreted as a function from graphs to graphs. The operator for building queries from patterns is denoted GRAPH. According to Definition 3.6, the value of a pattern P with template R over a graph G is a set of matches $[[P]]_G : R \Rightarrow G^{(P)}$, and the semantics of patterns is defined recursively in terms of their values. Thus, patterns have a graph-to-set-of-matches semantics, while queries have a graph-to-graph semantics, as defined below, based on Definition 2.8 of the image of a graph by a set of functions.

Definition 3.12 (syntax of queries)**.** A *query* Q of GrAL is written GRAPH (P) where P is either a CONSTRUCT or a UNION pattern. Then *the pattern of* Q is P and the *template* $\mathscr{T}(Q)$ of Q is the template of P.

Definition 3.13 (result of queries)**.** The *result* of a query Q with pattern P and template R over a graph G is the subgraph of $G^{(P)}$ image of R by $[[P]]_G$, it is denoted $Result(Q,G)$.

Thus, when $Q =$ GRAPH $(P_1$ CONSTRUCT $R)$, the result of Q over G is the graph $Result(Q,G) = [[P_1]]_G(R)$ built by "gluing" the graphs $m(R)$ for $m \in [[P_1]]_G$, where $m(R)$ is a copy of R with each variable $x \in \mathscr{V}(R) \setminus \mathscr{V}(X)$ replaced by a fresh variable $var(x,m)$. And when $Q =$ GRAPH $(P_1$ UNION $P_2)$, the result of Q over G is the graph $Result(Q,G) = H_1 \cup H_2$ where $H_i = Result($GRAPH $(P_i), G)$ and the fresh variables occuring in H_1 are distinct from the ones in H_2.
  ```

Example 3.14. It is now easy to compute the result of the GrAL queries:

$$Q_i = \text{GRAPH } (C_i)$$

over G_0 when C_i is a pattern from Example 3.9. We know that the result of Q_i applied to G_0 is an instance of R_i when $\mathcal{V}(R_i) \subseteq \mathcal{V}(C_i)$, and that in general it is built by "gluing" together several instances of R_i (as for query Q_5 below).

- **Author citations.** Let us say that an author $a1$ *cites* an author $a2$ when $a1$ has published a message that refers to a message published by $a2$. In order to build the graph of author citations we use the query:
  $Q_1 = \text{GRAPH } (C_1)$.
  From $[[C_1]]_{G_0}$ we get the graph:

  $$\mathit{Result}(Q_1, G_0) = \{\ \texttt{auth2 cites auth1 . auth3 cites auth1}\ \}\ .$$

- **Number of likes.** Let us count the number of *likes* in the database. We can get this result by counting the number of triples with predicate `likes`, or equivalently the number of predicates `likes`. We use the query:
  $Q_2 = \text{GRAPH } (C_2)$.
  From $[[C_2]]_{G_0}$ we get the graph:

  $$\mathit{Result}(Q_2, G_0) = \{\ \texttt{5}\ \}$$

  This result is the number 5, which is considered in GrAL as a graph made of only one isolated node. Note that we would get a query equivalent to Q_2 by counting either the number of authors $?a$ who like a message (with multiplicity the number of messages liked by $?a$), or by counting the number of messages $?m$ which are liked by someone (with multiplicity the number of authors who like $?m$). This means that the line `BIND COUNT(likes) AS ?n` could be replaced either by `BIND COUNT(?a) AS ?n` or by `BIND COUNT(?m) AS ?n` .

- **Number of likes per author.** Let us now count the number of *likes per author* in the database, which means, for each author count the number of likes of messages published by this author, except for self-likes. We display the result as the graph made of the triples `?a1 nbOfLikes ?n` where $?n$ is the number of likes of author $?a1$, by using the query:
  $Q_3 = \text{GRAPH } (C_3)$.
  From $[[C_3]]_{G_0}$ we get the graph:

  $$\mathit{Result}(Q_3, G_0) = \{\ \texttt{auth1 nbOfLikes 1 . auth2 nbOfLikes 1 . auth3 nbOfLikes 3}\ \}\ .$$

- **Number of friends per author.** Let us count the number of friends of each author, where friendship is the symmetric relation between authors defined as follows: two authors are *friends* when each one likes a publication by the other (here self-friends are allowed). We use the query:
  $Q_4 = \text{GRAPH } (C_4)$.
  From $[[C_4]]_{G_0}$ we get the graph:

  $$\mathit{Result}(Q_4, G_0) = \{\ \texttt{auth1 nbOfFriends 1 . auth2 nbOfFriends 1}\ \}\ .$$

- **Generation of fresh variables.** Now let us build, for each author $?a$ and each message $?m$ published by $?a$ and stamped at date $?d$, a tree with a fresh variable as root and with two branches, one

named `author` towards ?a and the other one named `date` towards ?d. We use the query:
$Q_5 = \text{GRAPH } (C_5)$.
From $[[C_5]]_{G_0}$ we get the graph:

$$Result(Q_5, G_0) = T_1 \cup T_2 \cup T_3 \cup T_4 \cup T_5$$

where each T_i is the copy of R_5 corresponding to the i-th row in $Tab([[C_5]]_{G_0})$, so that:

$$\begin{aligned}
T_1 &= \{\ \texttt{?r1 author auth1 . ?r1 date date1}\ \} \\
T_2 &= \{\ \texttt{?r2 author auth1 . ?r2 date date2}\ \} \\
T_3 &= \{\ \texttt{?r3 author auth2 . ?r3 date date1}\ \} \\
T_4 &= \{\ \texttt{?r4 author auth3 . ?r4 date date4}\ \} \\
T_5 &= \{\ \texttt{?r5 author auth3 . ?r5 date date4}\ \}
\end{aligned}$$

In fact, query Q_5 "mimicks" the following SELECT query Q'_5:

```
SELECT ?a ?d
WHERE { ?a publishes ?m . ?m stampedAt ?d }
```

As explained in [4], the various copies of the variable ?r in the result of Q_5 act as identifiers for the rows in the table result of Q'_5 over G_0 (as for instance in SPARQL), which is obtained by dropping the column ?r from $Tab([[P_5]]_{G_0})$. Note that the table $Tab([[P_5]]_{G_0})$ has all its rows distinct by definition, whereas this becomes false when the column ?r is dropped.

Definition 3.15 (equivalence of queries)**.** Two queries are *equivalent* if they have the same template and the same result over every graph.

It follows that queries with equivalent patterns are equivalent, but this condition is not necessary.

Remark 3.16 (about SPARQL queries)**.** CONSTRUCT queries in SPARQL are similar to CONSTRUCT queries in GrAL: the variables in $\mathscr{V}(R) \setminus \mathscr{V}(X)$ in GrAL play the same role as the blank nodes in SPARQL. However the subCONSTRUCT patterns are specific to GrAL. There is no SELECT query in this core version of GrAL, however following [4] we may consider SELECT queries as kinds of CONSTRUCT queries.

4 Conclusion

We considered the problem of the evaluation of graph-to-graph queries, namely CONSTRUCT queries, possibly involving nested sub-queries. We proposed a new uniform evaluation semantics of such queries which rests on a recursive definition of the notion of patterns and a new definition of the considered graphs which are allowed to have isolated nodes. Hence, the evaluation of a pattern always yields a pair consisting of a graph and a set of matches (variable assignments). Notice that we did not tackle explicitly graph-to-table queries such as the well-known SELECT queries. We have shown recently in [4] that SELECT queries are particular case of CONSTRUCT queries. This stems from an easy encoding of tables as graphs. Thus, the proposed semantics can be extended immediately to SELECT queries involving Sub-SELECT queries.
The present work opens several perspectives including a generalization of the proposed semantics to other models of graphs such as *property graphs*. Such an extension needs to ensure the existence of the main operations of the proposed algebra such as the *Merge* operation. An operational semantics, based

on rewriting systems, which is faithful with the evaluation semantics proposed in this paper is under progress. Its underlying rewrite rules are inspired by the algebraic approach in [3].
Furthermore, the core language GrAL contains only simple patterns needed to illustrate our uniform semantics. Comparison with other expressions such as EXISTS(*pattern*) or patterns such as FROM(*query*) [2, 8] remains to be investigated.

References

[1] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter & Domagoj Vrgoc (2017): *Foundations of Modern Query Languages for Graph Databases*. *ACM Comput. Surv.* 50(5), pp. 68:1–68:40, doi:10.1145/3104031.

[2] Renzo Angles & Claudio Gutiérrez (2011): *Subqueries in SPARQL*. In Pablo Barceló & Val Tannen, editors: *Proceedings of the 5th Alberto Mendelzon International Workshop on Foundations of Data Management, Santiago, Chile, May 9-12, 2011*, *CEUR Workshop Proceedings* 749, CEUR-WS.org. Available at http://ceur-ws.org/Vol-749/paper19.pdf.

[3] Dominique Duval, Rachid Echahed & Frédéric Prost (2020): *An Algebraic Graph Transformation Approach for RDF and SPARQL*. In Berthold Hoffmann & Mark Minas, editors: *Proceedings of the Eleventh International Workshop on Graph Computation Models, GCM@STAF 2020, Online-Workshop, 24th June 2020*, *EPTCS* 330, pp. 55–70, doi:10.4204/EPTCS.330.4.

[4] Dominique Duval, Rachid Echahed & Frédéric Prost (2020): *All You Need Is CONSTRUCT*. *CoRR* abs/2010.00843. Available at https://arxiv.org/abs/2010.00843.

[5] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer & Andrés Taylor (2018): *Cypher: An Evolving Query Language for Property Graphs*. In Gautam Das, Christopher M. Jermaine & Philip A. Bernstein, editors: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, ACM, pp. 1433–1445, doi:10.1145/3183713.3190657.

[6] Mark Kaminski, Egor V. Kostylev & Bernardo Cuenca Grau (2017): *Query Nesting, Assignment, and Aggregation in SPARQL 1.1*. *ACM Trans. Database Syst.* 42(3), pp. 17:1–17:46, doi:10.1145/3083898.

[7] Egor V. Kostylev, Juan L. Reutter & Martín Ugarte (2015): *CONSTRUCT Queries in SPARQL*. In Marcelo Arenas & Martín Ugarte, editors: *18th International Conference on Database Theory, ICDT 2015, March 23-27, 2015, Brussels, Belgium*, *LIPIcs* 31, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 212–229, doi:10.4230/LIPIcs.ICDT.2015.212.

[8] Axel Polleres, Juan L. Reutter & Egor V. Kostylev (2016): *Nested Constructs vs. Sub-Selects in SPARQL*. In Reinhard Pichler & Altigran Soares da Silva, editors: *Proceedings of the 10th Alberto Mendelzon International Workshop on Foundations of Data Management, Panama City, Panama, May 8-10, 2016*, *CEUR Workshop Proceedings* 1644, CEUR-WS.org. Available at http://ceur-ws.org/Vol-1644/paper10.pdf.

[9] (2013): *SPARQL 1.1 Query Language*. W3C Recommendation. Available at https://www.w3.org/TR/sparql11-query/.

[10] (2014): *RDF 1.1 Concepts and Abstract Syntax*. W3C Recommendation. Available at https://www.w3.org/TR/rdf11-concepts/.

Statistical Model Checking of Common Attack Scenarios on Blockchain

Ivan Fedotov Anton Khritankov
Moscow Institute of Physics and Technology
Moscow, Russia
ivan.fedotov@phystech.edu anton.khritankov@phystech.edu

Blockchain technology has developed significantly over the last decade. One of the reasons for this is its sustainability architecture, which does not allow modification of the history of committed transactions. That means that developers should consider blockchain vulnerabilities and eliminate them before the deployment of the system. In this paper, we demonstrate a statistical model checking approach for the verification of blockchain systems on three real-world attack scenarios. We build and verify models of DNS attack, double-spending with memory pool flooding, and consensus delay scenario. After that, we analyze experimental results and propose solutions to avoid these kinds of attacks.

1 Introduction

Satoshi Nakamoto proposed blockchain technology as a distributed ledger of connected records that are linked using cryptography measures [19]. The development of blockchain systems can be challenging even for experts because of the distributed execution environment and the persistence of records. Known consensus vulnerabilities increase the cost of errors. For example, the popular cryptocurrency exchange and wallet Coincheck got a security incident in January 2018 and more than 500 million USD were stolen [12].

The model checking verification approach introduces methods to construct models that describe the possible system behavior in a mathematically precise way. The accurate modeling of systems often leads to the discovery of incompleteness in informal system specifications. One can write specifications in different formalisms: linear-temporal logic, computational tree logic, and extensions of them. Finite-state automata are the most expressed way to model systems. The model checking process consists of three parts [6]:

- **Modeling phase:** model the system; formalize the property to be checked.
- **Running phase:** run the model checker engine to check the validity of the specifications.
- **Analysis phase:** if the system satisfies the property, then check the next properties; otherwise, generate and analyze the counterexample, refine the model and repeat the entire procedure.

In this paper, we apply a statistical model checking approach to blockchain systems. We model three types of attacks that affect the major parties in blockchain systems. In the DNS attack, an adversary changes the DNS address for the connection to the network. The probability distribution function defines the address spoofing success. Memory pool flooding and consensus delay attack models implement the scenario of double-spending of the same asset. Size of the memory pool and delay time one can take from the historical data of the Bitcoin network. Based on the experimental analysis, we consider approaches to reduce the probability of success of these attacks. The results of such analysis one can use in the planning of industrial product development.

T. Kutsia (Ed.): Symbolic Computation
in Software Science (SCSS'21)
EPTCS 342, 2021, pp. 65–77, doi:10.4204/EPTCS.342.6

The rest of the paper is organized as follows. In section 2 we describe tools and approaches that will be used in model checking. Section 3 explains attack scenarios and their models and provides restrictions on them. We carry out experiments in section 4. In section 5 we evaluate experimental results and suggest solutions for the prevention of successful attacks. In section 6 we provide related studies on the modeling of smart contracts. Section 7 provides a discussion and the concluding remarks.

2 Background

In this section, we describe a tool and techniques that we use to construct models. We also illustrate the work of tools with a simple example. We use BIP (Behavior, Interaction, Priority) framework and a statistical model checker SBIP 2.0 [18] to model adversary's scenario on Blockchain. The stochastic real-time BIP formalism provides an ability to build models assembled from components presented as stochastic timed automata. Timed automaton is an extension of ordinary automaton by a finite set of clocks and clock constraints. One can think about stochastic timed automaton as a combination of timed automaton and Markov chain. Probability density functions can describe uncertainty in the model. A designer of the system can provide a specification of properties as a Metric Temporal Logic (MTL) formula [15]. BIP framework uses a statistical model checking approach [18] that answers two types of questions:

- **Qualitative:** is the probability for the stochastic system S to satisfy ϕ greater or equal to a certain threshold Θ? The approach to answering the qualitative question is based on hypothesis testing.
- **Quantitative:** what is the probability for S to satisfy ϕ? Given a precision δ and a risk parameter α: $\mathscr{P}(|p' - p| < \delta) \geq 1 - \alpha$ the algorithm computes a value for p'.

In BIP, systems consist of three parts: atomic components, component interaction, and priority of interactions. Below we describe each of these parts.

2.1 Atomic Components

Timed automaton specifies the behavior of atomic components. One can define an atomic component as the following elements:

- set of ports for synchronization with other components
- set of states of the component
- set of variables for local data
- set of transitions between states; the transition executes under certain boolean conditions.

Fig. 1 illustrates a simplified purchasing model [7], adapted for the BIP framework. The atomic component *Customer* has 3 states *c0*, *c1*, and *c2*, variables *balance*, *transfer*, *price* and *id*. Also, the *Customer* component has two ports *process* and *receive*, and three transitions: *process*, *receive* and *done*. Interaction *process* executes only when the local variable $balance > price$. The other atomic component explains the behavior of the seller with 2 states, *s0*, *s1*, internal variable balance and list of goods, and three transitions. The transition *process* executes only when the number of goods is greater than zero.

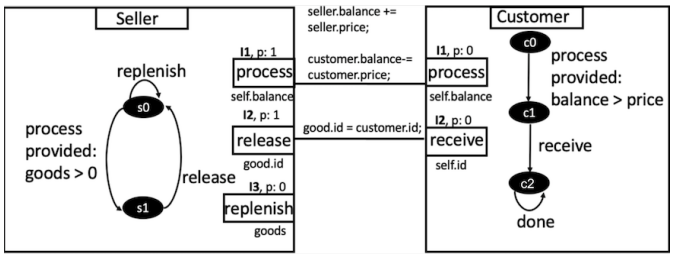


Figure 1: BIP model example

2.2 Connectors and Interactions

Atomic components can communicate with each other according to the logic in connectors. One can describe a connector as a sequence of ports that connect atomic components. SBIP 2.0 supports two types of interactions, timed and stochastic. Timed interactions take place only with time constraints that represent a lower and upper bound over clock valuations, as in timed automata. Stochastic interactions take place with a specific stochastic constraint, for example, a probability function.

Interaction *I1* at Fig. 1 connects atomic component *Customer* through the port *process*. This port implements the money transfer from the *Customer* to the *Seller* atomic component. Variable *balance* on interaction *I1* of the customer is reduced on the amount *price*, and the variable *balance* of the seller is incremented on the same amount. In interaction *I2* the asset assigns to the customer. Thus, the connector from example at Fig. 1 is a set of ports: $p_1|p_2$, where p_1 - port of atomic component *Customer* and p_2 - port of external atomic component *Seller*.

2.3 Priorities

Execution of priorities can proceed under certain conditions. If the condition holds, the priority of the considered connector is higher than another one. In the purchasing example from Fig. 1 a conflict can be between interactions *I2* and *I3* when the amount of good is non-zero. The interaction *process* executes first, as it has a higher priority.

2.4 Compound Components

Compound components are used for assembling a new component from the defined atomic components. Compound components include instances of atomic components and specify connectors between them. Fig. 1 illustrates a compound component that consists of two atomic components and two interactions.

3 Models Description

In this section, we describe models and attack scenarios. We study attacks from the recent survey [21] that possesses the most meaningful properties for blockchain systems:

- Affect peer-to-peer system or blockchain application.
- Led to the significant funds leak in the past.
- Involve several parts of the blockchain systems.
- Include non-deterministic interactions.

We present models for the DNS spoofing attack, double spending through the mempool flooding attack, and consensus delay attack. Each model describes a particular attack scenario. We focused on one probabilistic parameter in each model. It is either a probability distribution or a historical data set. Particular transition in the model depends on this probabilistic parameter. The code of the SBIP models is available in the repository [8].

3.1 DNS Attack

Blockchain applications use peer-to-peer network architecture for communication between the network nodes. When a new client joins a blockchain for the first time, she discovers active peers using Domain Name System (DNS) for IP address resolution. This is a DNS bootstrapping process.

The DNS mechanism is susceptible to cache poisoning, hijacking, tunneling, man-in-the-middle, and other types of attack [23]. A vulnerability in DNS led to the famous 2016 cyberattack [4].

We consider a DNS cache poisoning attack performed with a co-called birthday attack [23] on Berkeley Internet Name Domain (BIND) software. A BIND server can send multiple simultaneous recursive queries for the same IP address so an adversary can predict the next transaction id. After guessing the next transaction id, the adversary provides extra information in a DNS reply packet. Thus, by simultaneously generating a flow of queries to the server and an equal number of forged replies one can get a collision of transaction id and change the proper domain address to the fake one. The probability of collision is $P = 1 - (1 - \frac{1}{t})^{\frac{n*(n-1)}{2}}$ [23], where t is the total number of possible values in the master set, and n is the number of spoofed queries. The default number of possible values in the master set is 65535.

We took a birthday attack because the probability of collision is much higher than in conventional spoofing. The collision probability quickly increases up to 1.0 along for less than 1000 requests [23].

Let us describe an SBIP model of the DNS cache poisoning birthday attack. Fig. 2 illustrates the model. Here and on the other models, transitions with the prefix *l* present local interactions. The blue line indicates a connection between two atomic components. Connectors apply to interactions with the same name. For example, the adversary's atomic component can communicate with cache through external ports *requests*, *reply*, and *daemon*. The behavior of the atomic components is the following:

- Adversary. Provides a set of requests and replies. After each request, he tries to guess a transaction ID. A random event here is the *guess* transition, described by a collision probability function [23]. If the guess is successful, the adversary runs a daemon through the external interaction and goes to the final state $a2$.
- Cache. The cache server replies to DNS queries. In the case of collision, the interaction *daemon* works. The address in the DNS cache changes, and the user gets a spoofed address.
- User. Requests the DNS address from the cache server, connects to the network, and transfers funds. In case of a spoofed address, the transfer proceeds to the wrong recipient.
- Blockchain network. Awaits until the user connects to it and accomplishes transfer funds.
- Spoofed network. The same functionality as a blockchain network, but with the spoofed address.

The scenario of the attack is the following. The adversary makes requests to change the DNS address. At the same time, the user makes a query to the server. The request from the user and the adversary are temporary interactions that accomplish in the time period [0, 1000]. The upper bound of the period is a parameter, and one can change it. With the received address the user connects to the network and transfers the money.

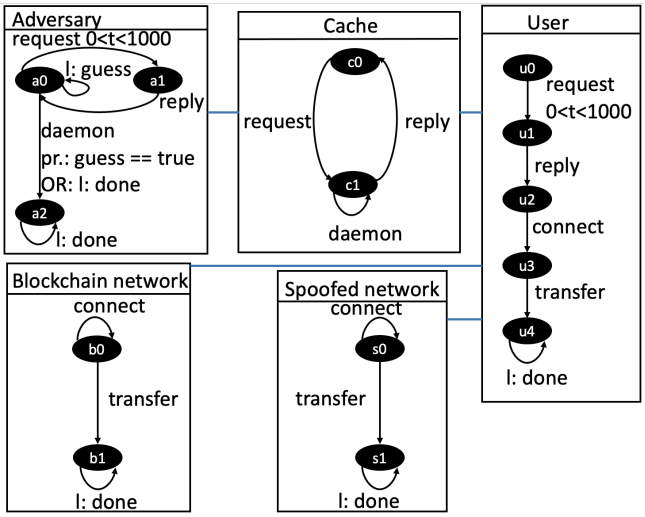


Figure 2: DNS spoofing model

One can bound the probability of the birthday attack by a negligible function μ by using cryptographic encryption techniques, for example, DNSSEC with NSEC5 records bounds [10]. NSEC5 is a resource record that can be used to detect certain attacks on secure DNS requests. We restricted the probability of violation by the function $\mu = R * q_s * 2^{-n}$ [10], where R - the set of domain names, that equals 65535, q_s - the number of adversary's queries, n - the length of the output of the hash function. Restriction applies on the transition *daemon* of the *Cache* atomic component.

3.2 Double Spending and Mempool Flooding

In a blockchain network, double-spending means applying the same transaction and its asset multiple times. Fig. 3 illustrates the process of double-spending. User A signs the transaction with a private key and sends it to user B. During the validation process, the recipient looks up the unspent transactions of the sender, verifies the sender's signature, and waits for the transaction to be mined into a valid block. Releasing the transaction to the network takes a certain time, which depends on the network throughput, the size of the memory pool of unconfirmed transactions (mempool), the consensus protocol, and the priority factor of the transaction.

In the case of fast transactions, the receiver can release the asset to the sender before the transaction gets committed into the blockchain network. In a while, user A can send the same transaction to user C. In this case, only those transaction is valid which gets into the blockchain first, but the asset might have been released twice. Double spending is one of the most widespread attacks on the blockchain platforms [20]. We performed a model checking for different types of double-spending attacks.

An adversary can cause a delay for successful double-spending with a mempool flooding attack. Mempool flooding is a kind of DDoS attack carried out at the memory pools of the cryptocurrencies [20]. Mempool has substantial properties such as timeout for transactions and default size limit. Relying on these configurations, users can estimate it to prioritize transactions. A potential adversary can estimate a delay in the mempool queue and increase this delay by a DDoS attack.

The survey [14] proposes two approaches to avoid double-spending. First, to set up the listening period before delivering the asset to the adversary. That gives time to double-check the spoofed transac-

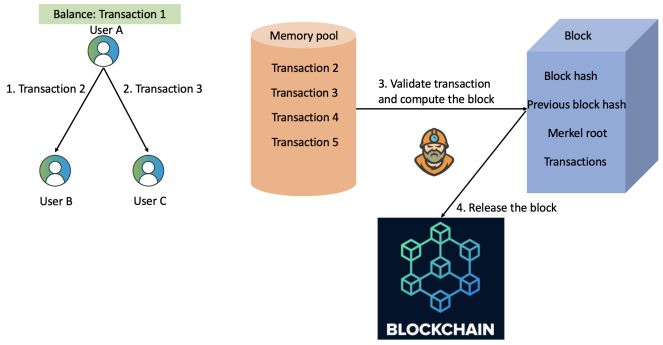


Figure 3: Double spending scenario

tion if the adversary already sends it to another user. But this does not consider the worst-case scenario when the adversary sends the second transaction just before accepting the first one to the blockchain. The second approach implies insertion to the network additional "observers" - nodes that would relay to the user all transactions that it receives. But observers entail additional upload to the network. Instead, we propose to set the time interval in which the blockchain participant can send one transaction to the network.

Model on Fig. 4 represents a simplified version of the mempool flooding scenario. We neglect the probability that one can send two transactions to the same miner by assuming that the blockchain network consists of a significant amount of participants. Also, in our model, there is no explicit mining fee for transactions. The historical data set considers the notion of the different fees for transactions implicitly. Below we explain atomic components.

- Adversary. Sends a transaction from the same parent's block to the different recipients. After that, the adversary waits for the release of the transaction's assets.
- Users A, B. Get the transaction from the sender, validate and locally verify it. Validation implies consistency with the blockchain's history and verification examines the sender's signature. After the verification phase, the user sends the transaction to the mempool and immediately releases the asset. The double-spending is successful if both assets were released to the sender.
- Miners C, D. Take transactions from users and allocate them in the local mempool until the block will be mined. Mining time is a random variable with distribution from the historical dataset [1].
- Proof-of-Work (PoW) Blockchain. We presented all other nodes as a *PoW Blockchain* atomic component. It takes the block from the miner, checks that the block does not contradict the history of committed transactions, and based on this either accepts or rejects the block.

The adversary sends a transaction to user A. After some time, the model defines a lower bound as a t' parameter, the adversary generates a transaction with the same history as the first one and sends it to user B. When the user receives the transaction, she validates its consistency with the blockchain's history. If the transaction is consistent, the user verifies the sender's signature and sends it to the mempool. The user releases a product to the sender before transaction confirmation.

3.3 Double Spending and Consensus Delay

Another way to enlarge the accepting time of the transaction is a consensus delay attack [5]. While the memory pool increases the delay of a block because of the transactions queue, consensus delay appears

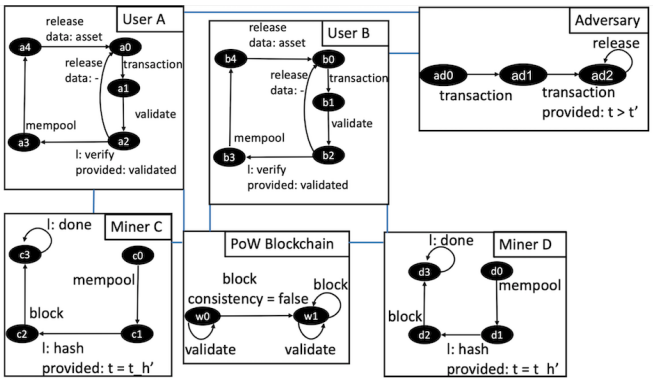


Figure 4: Mempool flooding model

because of the propagation time among the majority of users. When a new block is ready, the majority of blockchain's participants should confirm it. From the example in Fig. 3, only one transaction gets acceptance which is the first to propagate among more the half of the whole nodes.

Fig. 5 illustrates the consensus delay model with the Proof-of-Work consensus protocol. The behavior of the adversary and users are the same as in the mempool flooding model. When a node gets transactions from the user, it builds them into the block and propagates this block among other nodes. Propagation takes some time and causes the delay. *Ledger* atomic component represents other nodes of the network.

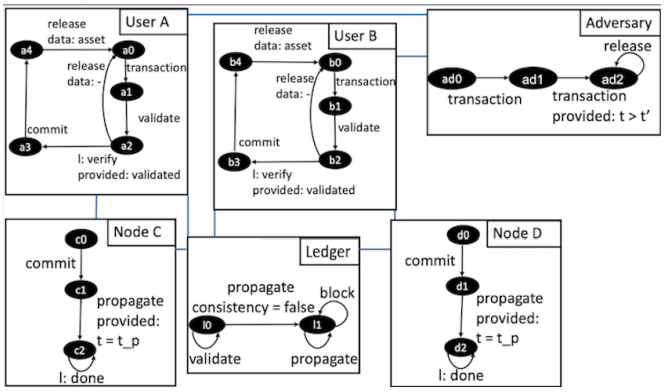


Figure 5: Consesus delay model

In the consensus delay model, the adversary sends two transactions at a time interval t'. Users validate and verify transactions and send them to other nodes to propagate them. If the first transaction already got commitment by the network, the second benign user rejected the transaction on the *validate* transition. We fixed the time interval t', in which the adversary can send the spoofed transaction.

4 Experimental Evaluation

We implemented three models with SBIP 2.0 framework and estimated the probability of satisfying the specification. In each model, we introduced a probabilistic parameter. In a DNS attack, it is a probability

of collision, in mempool flooding, it is the size of the memory pool. In the consensus delay model, the probabilistic parameter is a block propagation time. Each run can be either successful or not for an adversary, based on the probability distribution. The final ratio of successful attacks defines the probability of a successful adversary's scenario.

In the current section, we provide probability success estimated with the SBIP tool. The goal of experiments is to get the rate of the adversary's success and analyze the result to reduce the success probability of the adversary. We take time as a parameter of each experiment, as it is the most important factor that affects the result. Further, for each model, we propose a way to decrease the probability of adversary success. SBIP tool runs the model a certain number of times, the experiment parameters define this number. Parameters for the tool are the following: $\delta = 0.1, \alpha = 0.1$. If we take them less than 0.1, then the precision of the quantitative result of the experiment does not change significantly, but the evaluation time grows. In experiments we use statistical model checking algorithm [18], based on probability estimation method [9]. The code of models one can find in the repository [8].

Experiments imply the following assumptions. Adversary makes a DNS attack on BIND software with the corresponding probability distribution of the collision [23]. In the double-spending model, we consider a fast-transactions network. We also emphasize the reliability of measurements. If statistical data is not objective, i.e. there is a gap or multiple null values, then measurements are not reliable. If the experiment violates these assumptions, then the result is not correct.

The experiments were run on a 2,3 GHz Dual-Core Intel Core i5 CPU. The time and memory limits are 90 minutes and 4 GB, respectively.

4.1 DNS Attack

In this subsection, we provide the estimation of the adversary's success probability for the DNS model. We calculated a discrete set of values of the probability collision function. According to the MTL specification, the balance of the spoofed network eventually becomes nonzero: $F[0,x]spoofed.amount > 0$, where x is a parameter of the time in the range [100, 1000].

Fig. 6 depicts a probability versus time plot. Time denotes the number of milliseconds from the start of the scenario. If we add the probabilistic interaction, that restricts the collision success, the probability of transfer to the wrong network decreases significantly. We checked the violation probability with the different hash functions with different lengths: SHA-1, SHA-224, SHA-256, SHA-384, SHA-512 [22]. The success of the *daemon* interaction depends on the probability function $\mu = R * q_s * 2^{-n}$ [10], which we over-approximated by the uniform distribution.

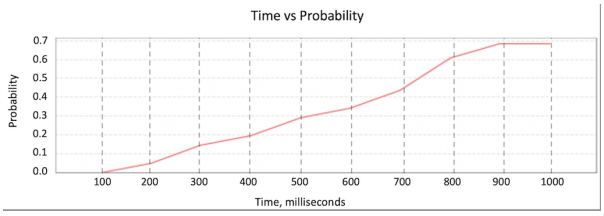


Figure 6: Dependence of the probability on the time for the DNS attack model

4.2 Double Spending and Mempool Flooding

Based on the Bitcoin data from 01/01/2016 till 01/05/2021 we provide the mining time [1] as a stochastic variable in our SBIP model. The stochastic variable t_h is assigned to the value from the dataset uniformly and randomly.

The experiment aims to get a minimum time interval t' in which the same user can send two transactions with the negligible probability of double-spending. The specification here represents the success of getting assets from both users: $F[0,10^{12}]adversary.asset == 2$. We took such an upper bound for the specification to over-approximate the mining time from the historical data set.

Fig. 7 illustrates the dependency of the double-spending probability on the time interval t'. We estimated the double-spending success probability with the time step in 10 seconds for t' between 500 and 650. Between 0 and 500, we measured with the time step in 50 seconds, as the probability changes more slowly. For the time interval, t' between 0 and 500 milliseconds, the probability of the double-spending is around 0.8. The following characteristics of the experiment explain this value. First, the value of the precision and risk parameters in the experiment is 0.1. Second, the adversary can send a transaction later than in t' seconds, as we provided the lower bound for the time interval.

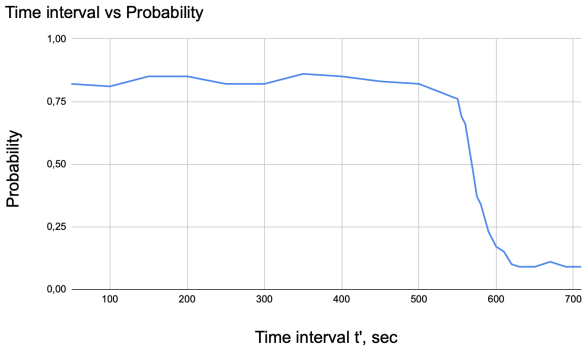


Figure 7: Dependence of the probability on the time interval t' for the mempool flooding model

4.3 Double Spending and Consensus Delay

Similar to the mempool double-spending scenario, we used the data [2] of the block propagation time to get a stochastic variable value. The goal of the experiment is to get the time interval t' between the adversary's transactions with which the double-spending probability is negligible. We took the historical data of propagation delay from 01/29/2016 till 01/05/2021 and assigned this data set to the random variable *t'* of the atomic component *Node*.

The specification checks that the adversary got the double-spending of the same transaction: $F[0,10^{12}]\ adversary.asset == 2$. We accomplished measurements with the time step equals to 2 seconds. We over-approximated the propagation time in the specification to include all possible delays. Fig. 8 illustrates the dependency of the time interval t' from the double-spending probability.

5 Analysis of Experimental Results

In this section, we estimate the results of experiments and evaluate proposed solutions to decrease the probability of a successful attack. A user can refine models with additional elements that specify her

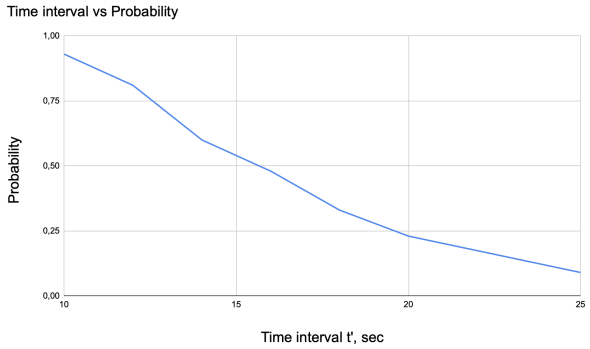


Figure 8: Dependence of the probability on the time interval t' for the consensus delay model

network. These elements can be a connection delay, network topology, bots for tracking DDoS attacks, and others. It is worth mentioning that a historical data set can include these elements implicitly. In this case, there is no need to change the model. Thus, one can use our models in their blockchain network.

5.1 DNS Attack

As we have seen from the experiment, the probability of the connection to the wrong network grows with the number of requests from the adversary. The probability function illustrated in Fig. 6 is similar to the collision rate function [23]. That means that the experimental evaluation corresponds to the theory. One can see from the experiment that for hash functions with an output length of more than 20 bits, the collision probability equals zero. Thus, using any listed hash functions [22] prevents the adversary's scenario.

5.2 Double Spending and Mempool Flooding

For the mempool flooding model, we analyzed the impact of the time interval between two transactions on the double-spending success. The increment of the time interval t', in which the adversary can send a transaction, decreases the double-spending probability. After 570 seconds, the probability of successful double-spending becomes less than 0.5. After 610 seconds, the probability becomes around 0.1 and remains in this value asymptotically. That follows from the setting of experimental parameters δ and α to 0.1 value. Also, the average time in the mempool, which is around 600 seconds, corresponds to the measured time interval, with which the double-spending probability becomes negligible. The double-spending probability should decrease significantly around average mining time, as after this point the first transaction most probably releases from the mempool. According to the dependency illustrated in Fig. 7, by setting the restriction for sending one transaction from the same user in 650 seconds one can reduce the probability of the double-spending success significantly.

Justification of the time restriction on transactions from the same user depends on the application area. That makes sense in the networks, for which confirmation and release asset of the first transaction is more crucial than the second transaction processing.

5.3 Double Spending and Consensus Delay

Changing of probability versus time function in the consensus delay scenario, is smoother than in the mempool case, as there is a higher deviation in the data set. Fig. 5 illustrates this dependency. Taking

into consideration the result of the mempool flooding experiment, we conclude that by restricting the time interval t' between two transactions from the same user in 650 seconds one can avoid both mempool flooding and consensus delay double-spending scenarios. To cover all factors that increase the double-spending success, one needs to make transaction fees and the mempool timeout as constants for all blockchain participants. The fixed transaction fee is possible to implement for the blockchain network [11]. One can fix the mempool timeout by assigning the same value for a timeout in memory pool configuration files for all blockchain participants and making these files immutable.

To sum up the experiment analysis, we specify security properties, that should hold for each system. In the case of a DNS attack, the system is safe if one uses any of the hash functions [22]. In the case of double-spending, one can put a time restriction for sending a transaction from one node. That bounds the successful attack probability by a negligible value and makes the system safe.

6 Related Work

The problem of the statistical model is fundamental and well-studied [16]. One can use the statistical model checking approach to verify blockchain systems [13]. To the best of our knowledge, only two studies [17] [3] provide a statistical model checking evaluation of blockchain systems. But these studies do not present experiments on real attack scenarios and discussion of model restrictions. In our setting, real attacks with probability conditions and restrictions of the models have been considered.

7 Conclusion

We modeled blockchain systems with historical data and probabilistic parameters. By running experiments on the historical data, one can predict the behavior of blockchain networks. We analyzed experimental results and proposed solutions to avoid the adversary's success and loss of money. Our models rely on real-world attack scenarios and cover all sides of the blockchain systems: miners, mining pools, exchanges, applications, users. Though we used open-source data from the Bitcoin network, one can adjust models for the private blockchain networks. Unlike in existing studies, our technique takes into account both temporal and stochastic interactions in blockchain systems.

As future work, we plan to expand the technique to automatically generate the SBIP models from the code of smart-contract. For some components of the blockchain system, such as memory pool or DNS cache it's difficult to build models automatically. But atomic components, that are responsible for the business logic in smart contracts, can be constructed in automatic mode. That extension reduces the manual work.

References

[1] (2021): *Average time to mine a block in minutes.* Available at `https://data.bitcoinity.org/bitcoin/block_time/5y?f=m10&t=l`.

[2] (2021): *Bitcoin Monitoring.* Available at `https://dsn.tm.kit.edu/bitcoin/`.

[3] Tesnim Abdellatif & Kei-Leo Brousmiche (2018): *Formal verification of smart contracts based on users and blockchain behaviors models*. In: *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, IEEE, pp. 1–5, doi:10.1109/NTMS.2018.8328737.

[4] Abhishta Abhishta, Roland van Rijswijk-Deij & Lambert JM Nieuwenhuis (2019): *Measuring the impact of a successful DDoS attack on the customer behaviour of managed DNS service providers*. *ACM SIGCOMM Computer Communication Review* 48(5), pp. 70–76, doi:10.1145/3310165.3310175.

[5] Gervais Arthur, Hubert Ritzdorf, Ghassan Karame & Srdjan Capkun (2015): *Tampering with the delivery of blocks and transactions in bitcoin*. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 692–705, doi:10.1145/2810103.2813655.

[6] Christel Baier & Joost-Pieter Katoen (2008): *Principles of model checking*. MIT press.

[7] Hsing-Chung Chen, Bambang Irawan, Chieh-Yang Shih, Cahya Damarjati, Zon-Yin Shae & Fengming Chang (2019): *A Smart Contract to Facilitate Goods Purchasing Based on Online Haggle*. In: *International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, Springer, pp. 618–628, doi:10.1007/978-3-030-22263-5_58.

[8] Ivan Fedotov & Anton Khritankov (2021): *SBIP models*. `https://github.com/1vanan/SBIP_models`.

[9] Thomas Hérault, Richard Lassaigne, Frédéric Magniette & Sylvain Peyronnet (2004): *Approximate probabilistic model checking*. In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*, Springer, pp. 73–84, doi:10.1007/978-3-319-06880-0_2.

[10] Amir Herzberg & Haya Shulman (2013): *DNSSEC: Security and availability challenges*. In: *2013 IEEE Conference on Communications and Network Security (CNS)*, IEEE, pp. 365–366, doi:10.1109/CNS.2013.6682730.

[11] Nicolas Houy (2014): *The economics of Bitcoin transaction fees*. *GATE WP* 1407, doi:10.2139/ssrn.2400519.

[12] Marie Huillet (2019): *Report: Record-Breaking Coincheck Hack Perpetrated by Virus Tied to Russian Hackers*. Available at `https://cointelegraph.com/news/report-record-breaking-coincheck-hack-perpetrated-by-virus-tied-to-russian-hackers`.

[13] Adnan Imeri, Nazim Agoulmine & Djamel Khadraoui (2020): *Smart Contract modeling and verification techniques: A survey*. In: *8th International Workshop on ADVANCEs in ICT Infrastructures and Services (ADVANCE 2020)*, pp. 1–8.

[14] Ghassan O Karame, Elli Androulaki & Srdjan Capkun (2012): *Double-spending fast payments in bitcoin*. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pp. 906–917, doi:10.1145/2382196.2382292.

[15] Ron Koymans (1990): *Specifying real-time properties with metric temporal logic*. *Real-Time Systems* 2(4), pp. 255–299, doi:10.1007/BF01995674.

[16] Axel Legay, Benoît Delahaye & Saddek Bensalem (2010): *Statistical model checking: An overview*. In: *International Conference on Runtime Verification*, Springer, pp. 122–135, doi:10.1007/978-3-642-16612-9_11.

[17] D.B. Maksimov, I.A. Yakimov & Kuznetsov A.S. (2020): *Statistical model checking for blockchain-based applications*. In: *IOP Conference Series: Materials Science and Engineering*, 734, IOP Publishing, p. 012152, doi:10.1088/1757-899X/734/1/012152.

[18] Braham Lotfi Mediouni, Ayoub Nouri, Marius Bozga, Mahieddine Dellabani, Axel Legay & Saddek Bensalem (2018): *SBIP 2.0: Statistical Model Checking Stochastic Real-Time Systems*. In: *International Symposium on Automated Technology for Verification and Analysis*, Springer, pp. 536–542, doi:10.1007/978-3-030-01090-4_33.

[19] Satoshi Nakamoto (2008): *Bitcoin: A peer-to-peer electronic cash system*. *Decentralized Business Review*, p. 21260.

[20] Muhammad Saad, Laurent Njilla, Charles A. Kamhoua, Kevin Kwiat & Aziz Mohaisen (2019): *Shocking Blockchain's Memory with Unconfirmed Transactions: New DDoS Attacks and Countermeasures*. *Blockchain for Distributed Systems Security*, p. 205, doi:10.1002/9781119519621.ch10.

[21] Muhammad Saad, Jeffrey Spaulding, Laurent Njilla, Charles Kamhoua, Sachin Shetty, DaeHun Nyang & Aziz Mohaisen (2019): *Exploring the attack surface of blockchain: A systematic overview*. *arXiv preprint arXiv:1904.03487*, doi:10.1109/COMST.2020.2975999.

[22] Rajeev Sobti & Ganesan Geetha (2012): *Cryptographic hash functions: a review*. *International Journal of Computer Science Issues (IJCSI)* 9(2), p. 461.

[23] Joe Stewart (2005): *DNS Cache Poisoning – The Next Generation*. Technical Report, Portland State University. Available at `https://www.ida.liu.se/~TDDD17/literature/dnscache.pdf`.

Learned Provability Likelihood for Tactical Search*

Thibault Gauthier
Czech Technical University in Prague, Prague, Czech Republic
email@thibaultgauthier.fr

We present a method to estimate the provability of a mathematical formula. We adapt the tactical theorem prover TacticToe to factor in these estimations. Experiments over the HOL4 library show an increase in the number of theorems re-proven by TacticToe thanks to this additional guidance. This amelioration in performance together with concurrent updates to the TacticToe framework lead to an improved user experience.

1 Introduction

We take inspiration from the instinct shown by mathematicians when attempting to prove a theorem. An important choice to make in their situation is whether to continue exploring a certain line of work or to switch to another approach entirely. The quality of this decision is influenced greatly by one's experience with other proof attempts. Our aim is to integrate such feedback to improve automation in interactive theorem provers (ITPs). In the majority of ITPs, most proofs are build using tactics in a goal-oriented manner. A tactic is a procedure that takes as input a goal g (a sequent in HOL4) and returns a list of goals whose conjunction implies g. We can classify HOL4 tactics into four categories. Solvers attempts to prove the goal using general or domain-specific knowledge (e.g. *metis_tac* [10] for first-order logic). Rewrite tactics modify subterms of a goal by applying rewrite rules constructed from proven equalities, Induction tactics (e.g. *Induct*) split the goal into multiple cases (typically a base case and an inductive case). Kernel-level tactics (e.g. *exist_tac*) allow for more refined control of the proof state. Given these tactics, it is possible to create an automated prover that searches for the proof of a starting goal by predicting suitable tactics for each intermediate goal. In the following, we refer to such automation as a tactic-based automated theorem prover (ATP). Multiple tactic-based ATPs have been developed in the course of the last five years and are now one of the most effective [8, 4] general proof automation available in an ITP. However, none of the tactic-based ATP so far take into account the provability of the goals produced by each tactic. Therefore, in this project, a tree neural network (TNN) is taught a function estimating the provabilty of goals in HOL4 [14]. This *value* function produces feedback signals called rewards that further guide the search algorithm of the tactic-based ATP TacticToe [8].

Related Works The most successful related tactic-based ATPs are Tactician [4] for Coq [3], Pamper [12] for Isabelle/HOL [15], and HOList [2] for HOL Light [9]. The Tactician is very user-friendly. In particular, it is the only one that can record tactic calls on the fly. It uses the k-nearest neighbor algorithm for tactic selection as TacticToe does. However, it does not predict argument theorems independently of tactics. In Pamper, the policy predictors are decision trees trained on top of human-engineered features. In HOList, the prediction effort is concentrated on learning the policy for a few selected tactics and their arguments (theorems) using deep reinforcement learning. A related field of research is machine learning

*Supported by the Czech Science Foundation project 20-06390Y

T. Kutsia (Ed.): Symbolic Computation
in Software Science (SCSS'21)
EPTCS 342, 2021, pp. 78–85, doi:10.4204/EPTCS.342.7

for first-order ATPs. The ENIGMA [5] system for E prover gathers positive and negative clauses from successful proof attempts. We use a similar technique to collect our training examples. The ATP LeanCoP [11] has been trained via a reinforcement learning loop using boosted random forest predictors. Its proof search relies on the same variant of Monte Carlo Tree Search (MCTS) [13] as TacticToe. Nevertheless, it ignores the goal selection issue and thus could benefit directly from the MCTS adaptions proposed in this paper.

2 Monte Carlo Tree Search with Tactics

We integrate the learned provability estimator into the proof search of TacticToe. Here is a brief summary of how the MCTS algorithm operates in the context of tactic-based theorem proving. The algorithm starts with a list of goals (typically a singleton) to be proven in a root node. In the selection phase, it chooses a goal branch in the current output node and a tactic branch (and possibly an argument branch) leading to the selection of an output node containing the list of goals produced by the tactic. This process is repeated until a leaf is reached. In the extension phase, the tactic t selected in the leaf is applied to its parent goal. In the case of a successful tactic application, an output leaf is created containing the list of goals produced by t. During the backup phase, a feedback signal is propagated from the newly created leaf to the root. The gathered node rewards influence the next selection phase. The three phases are repeated until the algorithm finds a proof for each of the root goals, times out, or saturates. In the following, we present the existing MCTS algorithm for TacticToe and describe ways to improve the quality of the feedback mechanism. Figure 1 illustrates the effect of one iteration of the improved MCTS loop on the search tree for our running example.

Selection Phase Given a list of tactic $(t_r)_{1\leq r\leq n}$ with parent node g, we can compute their PUCT [1] (Polynomial Upper Confidence Trees) score as follows:

$$PUCT(t_r) = AverageRewards(t_r) + c \times Policy(t_r) \times \frac{\sqrt{Visits(t_r)}}{Visits(g)}$$

The tactic with the highest PUCT score is selected. The policy *Policy* is given by a nearest neighbor predictor trained from supervised data consisting of goal-tactic pairs [8]. The value of $Policy(t_j)$ is experimentally chosen to be 0.5^{n+1} where n is the number of open tactic branches with parent node g and higher nearest neighbor score. At the start, the search is principally guided towards nodes with higher policy. As the number of iterations increases, the search tends to explore goals with higher reward averages more often. The exploration coefficient c decides how fast this transition happens. We choose c to be 2.0 since this is a suitable value for guiding a goal-oriented first-order ATP [11].

In this version of TacticToe, the theorems predicted by its nearest neighbor algorithm are split. Given a list of predicted arguments $x_1,\ldots,x_n$ and a tactic t (except *metis_tac*) that expects a list of theorems as argument, the tactic calls $t[x_1],\ldots,t[x_n]$ are constructed instead of $t[x_1,\ldots,x_n]$. This adds another branching layer to the tree that functions exactly like the tactic selection layer. To simplify our explanations in the rest of this paper, the branching occurring during argument selection is considered to be part of tactic selection.

In previous developments of TacticToe and in other tactic-based ATPs, the selected goal is always the first unproven goal of the output node. One issue is that the rewards of an output node are exactly the rewards of its first goal until it is proven. To factor the influence of other goals in the output node rewards, each unproven goal branch now receives almost the same number of visits. This is achieved by choosing at each iteration one of the least visited unproven goal branches.

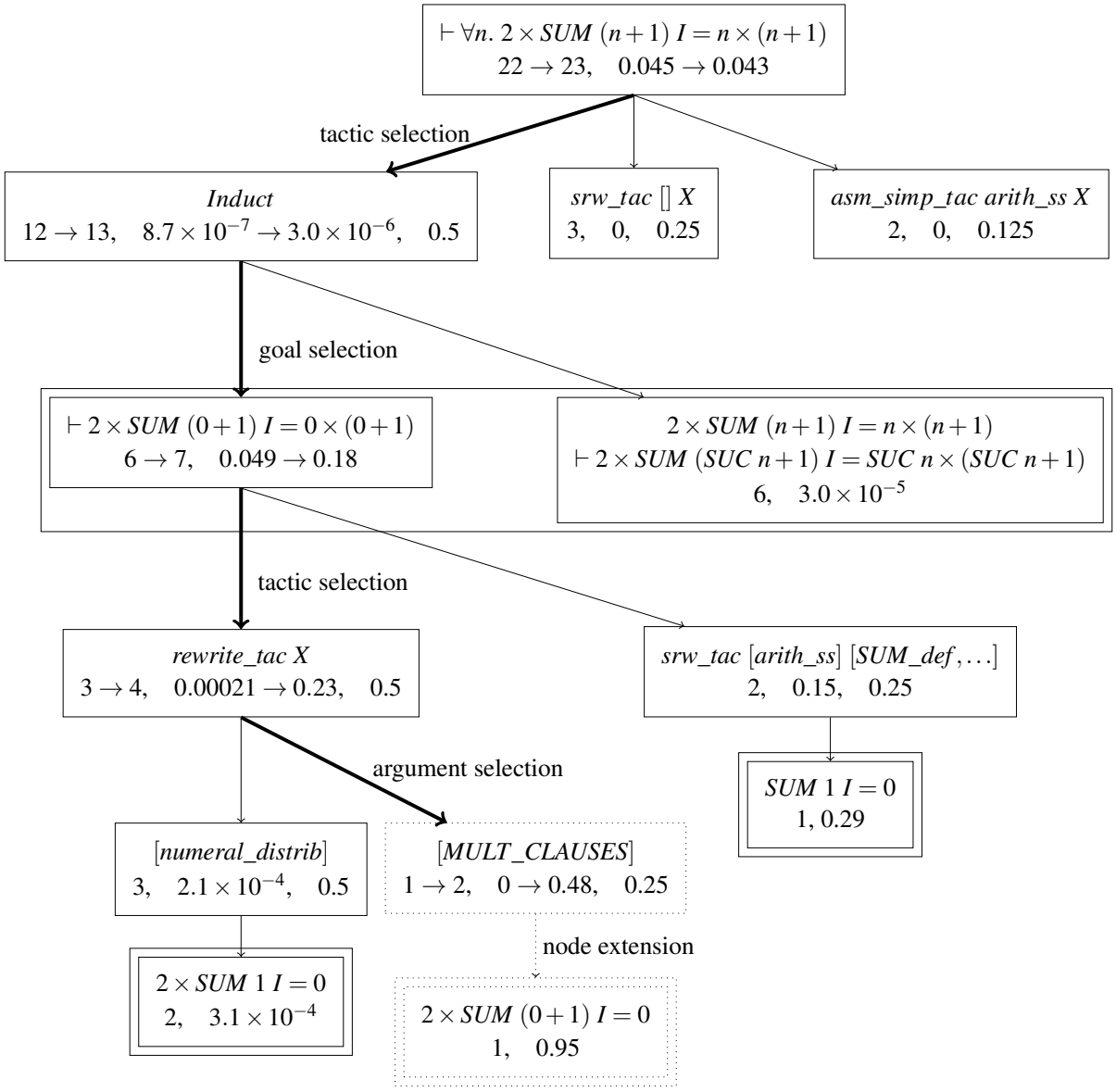


Figure 1: Iteration 22 of the value-guided MCTS loop on the goal $\vdash \forall n.\ 2 \times SUM\ (n+1)\ I = n \times (n+1)$. The term $SUM\ n\ f$ stands for $\sum_{x=0}^{x<n} f(x)$ and I is the identity function. Each node contains in the following order: the number of visits with a possible update, the average of the rewards and the prior policy for tactics (and arguments). The presence of an arrow after these numbers indicates a backup update. The selection path is made bold and created nodes are dotted. Saturated tactic (and argument) nodes and the subtree of the inductive case are omitted. Tactics may contain the placeholder X [8] to indicate that an argument has to be provided. To avoid dividing by 0 in the PUCT formula, all tactic (and argument) nodes are initialized with one visit and a reward of 0. A proof is found after 427 iterations.

Backup Phase There are three possible outcomes of the extension phase. If the tactic t applied proves the parent goal g, t receives a reward of 1. If t fails or induces a loop then t receives a reward of 0. Otherwise, a new output node is created and the value network is called on each of the goals. The reward of the newly created leaf is computed by multiplying the inferred values and backed up unchanged to t. Each tactic selection layer propagates the reward of the selected tactic unchanged to its parent goal. To capture the influence of multiple explored goal branches in the goal selection layer, we back up the rewards through this layer in the following manner. Given a list of goals $(g_r)_{1\leq r\leq n}$ composing an output node p and a selected goal g_i, the reward for p is given by:

$$Reward_k(p) = Reward_k(g_i) \times \prod_{1\leq r\leq n \,\wedge\, r\neq i} AverageRewards(g_r)$$

If a goal g_r is proven, the value of $AverageRewards(g_r)$ in the formula is overridden and set to 1. This formula multiplies the reward of the goal g_i at iteration k of the search loop with the existing reward average *AverageRewards* for the other goals in p. Thus the feedback of different branches is merged although only one branch is explored at each iteration of the loop. Overall, p receives a reward that is a lower bound estimation for its provability as it assumes independence of its goals.

3 Learning Provability

We explain here how a TNN can be trained to estimate the provability of a goal. These estimates are to be used as rewards in our improved MCTS algorithm. We choose a TNN as our machine learning model because it performs well on arithmetic and propositional formulas [7] as well as on Diophantine equations and combinators [6]. In our TNN, each HOL4 operator of arity a has a neural network associated with it modeling a function from $\mathbb{R}^{a\times d}$ to $\mathbb{R}^d$, where d is a globally fixed embedding size. When $a = 0$, the associated neural network is a trainable embedding (vector in $\mathbb{R}^d$). Networks are composed in a manner that reflects the tree structure of a given goal g. They gradually construct embeddings for sub-trees of g. A head network models a function from $\mathbb{R}^d$ to $\mathbb{R}$ targeting a provability estimation for g from an embedding for g. Figure 2 shows part of the computation graph produced by a TNN on the running example.

Until now, the value function of TacticToe was uniform. It gave a reward of 1 independently of the provability of the goal produced. Thus, the search was guided solely by a policy trained from human-written proof scripts. The lack of negative examples in these proofs makes this dataset unsuitable for training the value function. That is why we extract training examples from TacticToe searches instead. To do so, we collect search trees from the successful proof attempts. The goal nodes extracted from these trees create our training examples with proven goals labeled positively and other goals negatively. In order to speed up the training process, we keep at most one example for each goal with a preference for the positive ones. We also remove examples with large goals (≥ 80 operators) and select the 600 most visited negative examples per proof attempt. From this dataset, we train a TNN with embedding size ($d = 16$) and one layer per operator. The training is scheduled to take 100 epochs at a learning rate of 0.08 and the batch size is increased regularly according to the sequence $[16, 24, 32, 48, 64]$. These parameters were optimized by experimenting with a fifth of the HOL4 library. There, the parameters used in [7] were gradually mutated to yield higher accuracy on HOL4 goals.

4 Results

We experiment on the HOL4 library included in the HOL4 repository [1]. This library contains 168 theories about mathematical and computer science concepts such as lists, trees, probabilities, measures and

[1] `https://github.com/HOL-Theorem-Prover/HOL`

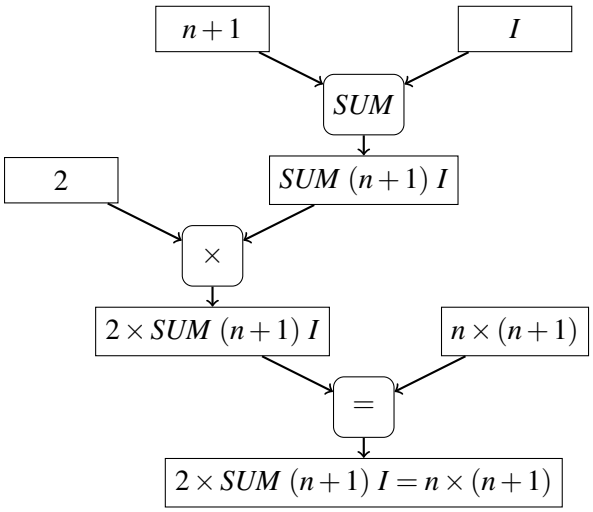


Figure 2: Computation of the embedding of $2 \times SUM\ (n+1)\ I = n \times (n+1)$ by a TNN. To simplify the diagram, the computation of the embeddings of 1, 2, $n+1$, $n \times (n+1)$ are not shown. Rectangles represent embeddings and squares represent neural networks.

.

integration. During the development of HOL4 library, the proof of a theorem is usually found as an argument of one of the following functions: `store_thm`,`maybe_thm`, `Store_thm`,`asm_store_thm`,`prove` or `TAC_PROOF`. We introduce a hook in these functions in order to evaluate TacticToe on their argument goal as the library is built. In this way, only tactics, theorems and simplification sets created before a goal g is proven are available to TacticToe when attempting to prove g. The code for this experiment is available at this commit[2]. Replication instructions are given in the file src/tactictoe/EVALUATION.

After an evaluation of TacticToe with a 30 seconds timeout over 15948 theorems of the library resulting in 8812 successful searches, 188409 examples (34021 positives and 154388 negatives) are collected. Correcting for the imbalance between the positive and negative examples by oversampling positive examples did not seem to improve the overall performance when experimenting with a small part of the library. Therefore, no balancing method is applied in this full-scale experiment. The efficiency of the learning phase is then assessed by splitting the examples into a training set (90%) and a test set (10%). Following the training of the TNN, we measure an accuracy of 97.5% on the training set and 84.7% on the test set. For the final evaluation, the TNN is retrained on all examples.

The performance of value-guided TacticToe is compared with its default version in Table 1. All changes to the proof search algorithm proposed in this paper are included in both versions. The only difference between the two versions is whether a uniform or trained value function gives the reward signal. Overall, the results show a small increase in the number of theorems re-proven. Calls to the TNN increase the node creation time from 3.7% to 13.3% of the total search time. Such a small footprint could not have been achieved without a Standard ML implementation of TNNs [7]. Thus, the positive impact of learning is not severely dampened by the neural network overhead.

4.1 Proofs

Among the 239 theorems solely proven by TacticToe$_{TNN}$, 30 of them belong to the theory real_topology. Two such theorems with their tactical proof are analyzed here. Compared to the previous version of

[2] 13fbff7b94bb1a5b51881a5aeee63ffc44d3be1

	TacticToe	TacticToe$_{TNN}$	Combined
Standard library (15948 theorems)	8812 (138)	8913 (239)	9051

Table 1: Number of theorems in the HOL4 standard library re-proven within 30 seconds. The number of theorems re-proven by one strategy but not by the other is shown in parentheses.

TacticToe, the proofs are now automatically printed with the preferred style of many HOL4 users. Tactics are written in lowercase when possible and the tacticals >> (one goal), >| (multiple goals) compose tactics in the final proof script.

The theorem $\vdash \forall a.\ \mathit{diameter}\ \{a\} = 0$ is re-proven in 28.2 seconds. It states that the diameter of any singleton is equal to 0. The diameter of a set of reals s is by definition 0 if s is empty and otherwise is the supremum $\{abs(x-y) \mid x \in s \wedge y \in s\}$. The proof starts by rewriting the goal with the definition of *diameter*, continues by distinguishing whether s is empty or not and proves each case by calling a first-order solver and a simplification tactic with an appropriate lemma.

```
rewrite_tac [diameter] >> REPEAT strip_tac >>
COND_CASES_TAC >| [metis_tac [], srw_tac [] [REAL_SUP_UNIQUE]]
```

The theorem $\vdash \forall f\ s.\ (\mathit{linear}\ f \wedge \mathit{subspace}\ s) \Rightarrow \mathit{subspace}\ (\mathit{IMAGE}\ f\ s)$ is re-proven in 16.7 seconds. It states that the image of a subspace by a linear function is a subspace. A subspace s is a subset of $\mathbb{R}$ that satisfies three conditions $0 \in s$, $(x \in s \wedge y \in s) \Rightarrow x+y \in s$, and $\forall c \in \mathbb{R}.\ x \in s \Rightarrow c \times x \in s$. The proof consists of rewriting the goal with this definition and then solving each of the three cases by calling *metis_tac* with suitable premises.

```
srw_tac [] [subspace] >|
  [metis_tac [REAL_MUL_LZERO,linear],
   first_assum (X_CHOOSE_TAC ``B`` o MATCH_MP (LINEAR_BOUNDED)) >>
     pop_assum (mp_tac o Q.SPEC `x`) >> metis_tac [LINEAR_ADD],
   metis_tac [LINEAR_CMUL]]
```

5 Usage

To make TacticToe more attractive to new users, tactic-goal pairs from the HOL4 library are prerecorded. Therefore, the users can now use TacticToe right after building HOL4 without spending multiple hours recording the data themselves (see HOL/src/tactictoe/README). Multiple new functions relying on the TNN trained in this paper are implemented and available at the commit mentioned in Section 4 . To run these functions, one first need to download the file `tnn_for_tactictoe` from `http://grid01.ciirc.cvut.cz/~thibault/` to a desired location `path_to_foo`. The function `ttt_tnn` runs TacticToe $_{TNN}$ on a chosen goal with the advice of a TNN imported by `mlTreeNeuralNetwork.read_tnn`. The following commands demonstrate how to execute `ttt_tnn` in an interactive session:

```
load "tacticToe"; open tacticToe;
val tnn = mlTreeNeuralNetwork.read_tnn "path_to_foo";
load "sum_numTheory"; open sum_numTheory; open arithmeticTheory;
set_timeout 60.0;
ttt_tnn tnn ([],``!n. 2 * SUM (n+1) I = n * (n+1) ``);
```

The function `confidence_tnn` returns the provability estimate of a goal as computed by the TNN in 1 to 10 milliseconds. This might encourage the user to call TacticToe $_{TNN}$ if this value becomes high on an open goal during a manual proof attempt.

The function `suggest` creates partial proofs from failed proof attempts. By limiting the numbers of iterations of the MCTS loop to 22 (`tttSearch.looplimit := SOME 22;`), a call to `ttt_tnn` now fails on the running example but the search tree is stored in the reference `searchtree_glob`. Executing `suggest ();` at this point returns the most promising partial proof tree in `searchtree_glob`. To construct the tree, this function traverses down the tree and selects the first proved tactic branch or the most visited open tactic branch if no branch has been proved. A tactic built from this proof tree can then be applied to the original goal and the user might continue working on the produced open goals. The resulting partial proof suggested from our failed attempt is:

```
Induct >| [rewrite_tac [numeralTheory.numeral_distrib] >> all_tac,
           srw_tac [ARITH_ss] [MULT_CLAUSES,LEFT_ADD_DISTRIB] >> all_tac]
```

The depth of this advice may be controlled by updating the reference `tttSearch.suggest_depth`. The function `suggest` can also be used in combination with the original `ttt` function but the advice will be of lesser value as the most promising proof branches are fixed by the policy and therefore the most promising proof tree is gradually extended as the search tree grows. In contrast, the decision on which path is most promising might be revised when running `ttt_tnn` with higher timeouts.

6 Conclusion

Training a TNN on intermediate goals generated by TacticToe leads to the creation of a goal provability estimator with good accuracy. Since a goal can be split into multiple goals during tactic-based searches, we propose improvements to the selection and backup phases of the underlying MCTS algorithm in order to get more accurate feedback from the learned estimator. Experiments on the full HOL4 library measure an increase in the success rate of TacticToe and demonstrate the scalability of our approach. This resulted in the creation of new tools for HOL4 users that leverage the advice given by the estimator. In particular, TacticToe is now able to suggest how to start a proof even when it fails to find one.

References

[1] David Auger, Adrien Couëtoux & Olivier Teytaud (2013): *Continuous Upper Confidence Trees with Polynomial Exploration - Consistency*. In: *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2013, Prague, Czech Republic, September 23-27, 2013, Proceedings, Part I*, pp. 194–209, doi:10.1007/978-3-642-40988-2_13.

[2] Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy & Stewart Wilcox (2019): *HOList: An Environment for Machine Learning of Higher Order Logic Theorem Proving*. In Kamalika Chaudhuri & Ruslan Salakhutdinov, editors: *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, *Proceedings of Machine Learning Research* 97, PMLR, pp. 454–463. Available at `http://proceedings.mlr.press/v97/bansal19a.html`.

[3] Yves Bertot (2008): *A Short Presentation of Coq*. In Otmane Ait Mohamed, César Muñoz & Sofiène Tahar, editors: *Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, *LNCS* 5170, Springer, pp. 12–16, doi:10.1007/978-3-540-71067-7_3.

[4] Lasse Blaauwbroek, Josef Urban & Herman Geuvers (2020): *The Tactician - A Seamless, Interactive Tactic Learner and Prover for Coq*. In Christoph Benzmüller & Bruce R. Miller, editors: *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings*, *LNCS* 12236, Springer, pp. 271–277, doi:10.1007/978-3-030-53518-6_17.

[5] Karel Chvalovský, Jan Jakubuv, Martin Suda & Josef Urban (2019): *ENIGMA-NG: Efficient Neural and Gradient-Boosted Inference Guidance for E*. In: *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, pp. 197–215, doi:10.1007/978-3-030-29436-6_12.

[6] Thibault Gauthier (2020): *Deep Reinforcement Learning for Synthesizing Functions in Higher-Order Logic*. In Elvira Albert & Laura Kovács, editors: *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020*, *EPiC Series in Computing* 73, EasyChair, pp. 230–248, doi:10.29007/7jmg.

[7] Thibault Gauthier (2020): *Tree Neural Networks in HOL4*. In Christoph Benzmüller & Bruce R. Miller, editors: *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings*, *LNCS* 12236, Springer, pp. 278–283, doi:10.1007/978-3-030-53518-6_18.

[8] Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar & Michael Norrish (2021): *TacticToe: Learning to Prove with Tactics*. *J. Autom. Reason.* 65(2), pp. 257–286, doi:10.1007/s10817-020-09580-x.

[9] John Harrison (2009): *HOL Light: An Overview*. In Stefan Berghofer, Tobias Nipkow, Christian Urban & Makarius Wenzel, editors: *Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, *LNCS* 5674, Springer, pp. 60–66, doi:10.1007/978-3-642-03359-9_4.

[10] Joe Hurd (2005): *System Description: The Metis Proof Tactic*. In Carsten Schuermann Christoph Benzmueller, John Harrison, editor: *Workshop on Empirically Successful Automated Reasoning in Higher-Order Logic (ESHOL)*, pp. 103–104. Available at `https://arxiv.org/pdf/cs/0601042`.

[11] Cezary Kaliszyk, Josef Urban, Henryk Michalewski & Miroslav Olsák (2018): *Reinforcement Learning of Theorem Proving*. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi & Roman Garnett, editors: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pp. 8836–8847. Available at `https://proceedings.neurips.cc/paper/2018/hash/55acf8539596d25624059980986aaa78-Abstract.html`.

[12] Yutaka Nagashima & Yilun He (2018): *PaMpeR: proof method recommendation system for Isabelle/HOL*. In Marianne Huchard, Christian Kästner & Gordon Fraser, editors: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, ACM, pp. 362–372, doi:10.1145/3238147.3238210.

[13] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel & Demis Hassabis (2017): *Mastering the game of Go without human knowledge*. *Nature* 550, pp. 354–359, doi:10.1038/nature24270.

[14] Konrad Slind & Michael Norrish (2008): *A Brief Overview of HOL4*. In Otmane Aït Mohamed, César A. Muñoz & Sofiène Tahar, editors: *Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, *LNCS* 5170, Springer, pp. 28–32, doi:10.1007/978-3-540-71067-7_6.

[15] Makarius Wenzel, Lawrence C. Paulson & Tobias Nipkow (2008): *The Isabelle Framework*. In Otmane Aït Mohamed, César A. Muñoz & Sofiène Tahar, editors: *Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, *LNCS* 5170, Springer, pp. 33–38, doi:10.1007/978-3-540-71067-7_7.

Congruence Closure Modulo Permutation Equations

Dohan Kim and Christopher Lynch
Clarkson University, Potsdam, NY, USA
{dohkim,clynch}@clarkson.edu

We present a framework for constructing congruence closure modulo permutation equations, which extends the *abstract congruence closure* [7] framework for handling permutation function symbols. Our framework also handles certain interpreted function symbols satisfying each of the following properties: idempotency (I), nilpotency (N), unit (U), $I \cup U$, or $N \cup U$. Moreover, it yields convergent rewrite systems corresponding to ground equations containing permutation function symbols. We show that congruence closure modulo a given finite set of permutation equations can be constructed in polynomial time using equational inference rules, allowing us to provide a polynomial time decision procedure for the word problem for a finite set of ground equations with a fixed set of permutation function symbols.

1 Introduction

Congruence closure procedures [12, 18, 19] have been researched for several decades, and play important roles in software/hardware verification (see [9, 19, 20]) and satisfiability modulo theories (SMT) solvers [8, 10]. They provide fast decision procedures for determining whether a ground equation is an (equational) consequence of a given set of ground equations. (The fastest known congruence closure algorithm runs in $O(n \log n)$ [15].)

In [7, 14], some approaches to constructing the congruence closure of ground equations using completion methods were considered. These approaches capture the efficient techniques from standard term rewriting for congruence closure procedures. In particular, the *abstract congruence closure* approach in [7] (cf. Kapur's approach in [14]) constructs a reduced convergent ground rewrite system R_S for a finite set of ground equations S, which consists of either rewrite rules of the form $a \to c$ or $f(c_1, \ldots, c_n) \to c$ or $c \to d$ for fresh constants $c_1, \ldots, c_n, c, d$. Furthermore, R_S is a conservative extension of the equational theory induced by S (i.e. the congruence closure $CC(S)$) on ground terms, and two ground terms are congruent in $CC(S)$ iff they have the same normal form w.r.t. R_S. Note that this approach does not require a total termination ordering on ground terms.

Congruence closure procedures were extended to congruence closure procedures modulo theories in order to handle interpreted function symbols in the signature [3, 6, 15]. The notion of congruence closure modulo associative and commutative (AC) theories was discussed in [6, 16], and the notion of conditional congruence closure with uninterpreted and some interpreted function symbols was considered in [15].

Meanwhile, an equation is a *permutation equation* [1] if it is of the form $f(x_1, \ldots, x_n) \approx f(x_{\pi(1)}, \ldots, x_{\pi(n)})$, where π is a permutation on the set $\{1, \ldots, n\}$. Commutativity is the simplest case of permutation equations. Permutation equations are difficult to handle in equational reasoning without using the modulo approach. For example, an ordered completion procedure for *ordered rewriting* [5] produces every equation of the form $f(x_1, x_2, \ldots, x_n) \approx f(x_{\rho(1)}, x_{\rho(2)}, \ldots, x_{\rho(n)})$ (up to variable renaming) from two permutation equations $f(x_1, x_2, \ldots, x_n) \approx f(x_2, x_1, x_3, \ldots, x_n)$ and $f(x_1, x_2, \ldots, x_n) \approx f(x_2, x_3, \ldots, x_n, x_1)$, where ρ is a permutation in the symmetric group S_n of cardinality $n!$. (Recall that the symmetric group S_n can be generated by two cycles $(1\,2)$ and $(1\,2 \cdots n)$.) Therefore, it is natural to view permutation

T. Kutsia (Ed.): Symbolic Computation
in Software Science (SCSS'21)
EPTCS 342, 2021, pp. 86–98, doi:10.4204/EPTCS.342.8

equations as "structural axioms" (defining a congruence relation on terms) rather than viewing them as "simplifiers" (defining a reduction relation on terms) [5].

In this paper, we present a framework for generating congruence closure modulo a finite set of permutation equations. To our knowledge, it has not been discussed in the literature, and a polynomial time decision procedure for the word problem for a finite set of ground equations with a fixed set of permutation function symbols has not yet been known.

Our framework is based on the notion of abstract congruence closure that is particularly useful for term representation and checking E-equality between two flat terms for a given set of permutation equations E, which does not require an E-compatible ordering (cf. [17]). In addition, it also handles function symbols satisfying each of the following properties: idempotency (I), nilpotency (N), unit (U), $I \cup U$, or $N \cup U$. (If a function symbol is a permutation function symbol satisfying one of the above properties, then it should be a commutative function symbol.)

We show that congruence closure modulo a given finite set of permutation equations (with or without the function symbols satisfying the above properties) can be constructed in polynomial time, which provides a polynomial time decision procedure for the word problem for a finite set of ground equations with a fixed set of permutation function symbols (appearing in E).

2 Preliminaries

We use the standard terminology and definitions of term rewriting [4, 11], congruence closure [7, 12, 19], and permutation groups [13]. We also use some terminology and the results of permutation equations found in [1, 2].

We denote by $\mathscr{T}(\mathscr{F}, \mathscr{X})$ the set of terms over a finite set of function symbols $\mathscr{F}$ and a denumerable set of variables $\mathscr{X}$. We denote by $T(\mathscr{F})$ the set of ground terms over $\mathscr{F}$. We assume that each function symbol in $\mathscr{F}$ has a fixed arity.

An *equation* is an expression $s \approx t$, where s and t are (first-order) terms built from $\mathscr{F}$ and $\mathscr{X}$. A *ground equation* (resp. *ground term*) is an equation (resp. a term) which does not contain any variable.

We write $s[u]$ if u is a subterm of s and denote by $s[t]_p$ the term that is obtained from s by replacing the subterm at position p of s by t.

An *equivalence* is a reflexive, transitive, and symmetric binary relation. An equivalence $\sim$ on terms is a *congruence* if $s \sim t$ implies $u[s]_p \sim u[t]_p$ for all terms s, t, u and positions p.

An *equational theory* is a set of equations. We denote by $\approx_E$ (called the *equational theory* induced by E) the least congruence on $T(\mathscr{F}, \mathscr{X})$ that is stable under substitutions and contains a set of equations E. If $s \approx_E t$ for two terms s and t, then s and t are *E-equivalent*.

Given a finite set $S = \{a_i \approx b_i \mid 1 \le i \le m\}$ of ground equations where $a_i, b_i \in T(\mathscr{F})$, the *congruence closure* $CC(S)$ [3, 15] is the smallest subset of $T(\mathscr{F}) \times T(\mathscr{F})$ that contains S and is closed under the following rules: (i) $S \subseteq CC(S)$, (ii) for every $a \in T(\mathscr{F})$, $a \approx a \in CC(S)$ (*reflexivity*), (iii) if $a \approx b \in CC(S)$, then $b \approx a \in CC(S)$ (*symmetry*), (iv) if $a \approx b$ and $b \approx c \in CC(S)$, then $a \approx c \in CC(S)$ (*transitivity*), and (v) if $f \in \mathscr{F}$ is an n-ary function symbol ($n > 0$) and $a_1 \approx b_1, \ldots, a_n \approx b_n \in CC(S)$, then $f(a_1, \ldots, a_n) \approx f(b_1, \ldots, b_n) \in CC(S)$ (*monotonicity*). Note that $CC(S)$ is also the equational theory induced by S.

A (strict) ordering $\succ$ on terms is an irreflexive and transitive relation on $T(\mathscr{F}, \mathscr{X})$.

Given a rewrite system R and a set of equations E, the rewrite relation $\rightarrow_{R,E}$ on $T(\mathscr{F}, \mathscr{X})$ is defined by $s \rightarrow_{R,E} t$ if there is a non-variable position p in s, a rewrite rule $l \rightarrow r \in R$, and a substitution σ such that $s|_p \approx_E l\sigma$ and $t = s[r\sigma]_p$. The transitive and reflexive closure of $\rightarrow_{R,E}$ is denoted by $\stackrel{*}{\rightarrow}_{R,E}$. We say that a term t is an *R,E-normal form* if there is no term t' such that $t \rightarrow_{R,E} t'$.

The rewrite relation $\to_{R/E}$ on $T(\mathscr{F},\mathscr{X})$ is defined by $s \to_{R/E} t$ if there are terms u and v such that $s \approx_E u$, $u \to_R v$, and $v \approx_E t$. We simply say the rewrite relation $\to_{R/E}$ (resp. $\to_{R,E}$) on $T(\mathscr{F},\mathscr{X})$ as the rewrite relation R/E (resp. R,E).

The rewrite relation R,E is *Church-Rosser modulo* E if for all terms s and t with $s \overset{*}{\leftrightarrow}_{R\cup E} t$, there are terms u and v such that $s \overset{*}{\to}_{R,E} u \overset{*}{\leftrightarrow}_E v \overset{*}{\leftarrow}_{R,E} t$. The rewrite relation R,E is *convergent modulo* E if R,E is Church-Rosser modulo E and R/E is well-founded.

The *depth* of a term t is defined as $depth(t) = 0$ if t is a variable or a constant and $depth(f(s_1,\ldots,s_n)) = 1 + \max\{depth(s_i) \mid 1 \le i \le n\}$. A term t is *flat* if its depth is 0 or 1.

An equation of the form $f(x_1,\ldots,x_n) = f(x_{\rho(1)},\ldots,x_{\rho(n)})$ is a *permutation equation* [1] if ρ is a permutation on $\{1,\ldots,n\}$. We use variable naming in such a way that the left-hand side of each equation in a set of permutation equations with the same function symbol has the same name of variables $x_1,\ldots,x_k$ from left to right. (In this paper, we assume that the set of function symbols $\mathscr{F}$ in $T(\mathscr{F},\mathscr{X})$ is finite and each function symbol in $\mathscr{F}$ has a fixed arity.)

We denote by $\mathscr{F}_E$ the set of all function symbols occurring in a finite set of permutation equations E.

If $e := f(x_1,\ldots,x_n) \approx f(x_{\rho(1)},\ldots,x_{\rho(n)})$ is a permutation equation, then ρ is the permutation of this equation. We denote by $\pi[e]$ the permutation of e. For example, ρ is the permutation of the permutation equation $e' := f(x_1,x_2,x_3,x_4) \approx f(x_1,x_3,x_2,x_4)$ (i.e. $\pi[e'] = \rho$) with $\rho(1) = 1, \rho(2) = 3$, $\rho(3) = 2$, and $\rho(4) = 4$. Let E be a set of permutation equations with the same top function symbol. Then $\Pi[E]$ is defined as $\Pi[E] := \{\pi[e] \mid e \in E\}$. The permutation group generated by $\Pi[E]$ is denoted by $<\Pi[E]>$.

Theorem 1. *(see Theorem 1.4 in [2]) Let E be a set of permutation equations and let e be a permutation equation such that every equation in $E \cup \{e\}$ has the same (top) function symbol. Then $E \models e$ if and only if $\pi[e] \in <\Pi[E]>$.*

Let $i_1, i_2, \ldots, i_r\,(r \le n)$ be distinct elements of $I_n = \{1,2,\ldots,n\}$. Then $(i_1\, i_2 \cdots i_r)$, called *a cycle of length* r, is defined as the permutation that maps $i_1 \mapsto i_2$, $i_2 \mapsto i_3$,..., $i_{r-1} \mapsto i_r$ and $i_r \mapsto i_1$, and every other element of I_n maps onto itself. The symmetric group S_n of cardinality $n!$ can be generated by two cycles (12) and $(12\cdots n)$.

Example 1. Let $E = \{f(x_1,x_2,x_3,x_4,x_5) \approx f(x_2,x_1,x_3,x_4,x_5), f(x_1,x_2,x_3,x_4,x_5) \approx f(x_2,x_3,x_4,x_5,x_1)\}$. Then $\Pi[E]$ consists of two cycles $\{(12),(12345)\}$. Since two cycles (12) and (12345) generate the symmetric group S_5, we see that $<\Pi[E]>$ is S_5. Therefore, $f(x_1,\ldots,x_5) \approx_E f(x_{\tau(1)},\ldots,x_{\tau(5)})$ for any permutation $\tau \in S_5$ by Theorem 1.

Let E be a finite set of permutation equations. Then E can be uniquely decomposed as $\bigcup_{i=1}^{n} E_i$ such that (i) each E_i is a finite set of permutation equations, and (ii) E_j and E_k with $j \neq k$ are disjoint such that if $s_j \approx t_j \in E_j$ and $s_k \approx t_k \in E_k$, then s_j and s_k do not have the same top symbol (and are not variants of each other). Since we assume that each function symbol has a fixed arity, each distinct function symbol occurring in E corresponds to a distinct E_i in E. We denote by $Eq(f)$ the corresponding equational theory with terms headed by such a function symbol f. Now, we may apply Theorem 1 for each $Eq(f)$ in E with $f \in \mathscr{F}_E$.

3 Congruence closure modulo permutation equations

Definition 2. Let K be a set of constants disjoint from $\mathscr{F}$.

(i) A *D-rule* (w.r.t. $\mathscr{F}$ and K) is a rewrite rule of the form $f(c_1,\ldots,c_n) \to c$, where $c_1,\ldots,c_n,c$ are constants in K and $f \in \mathscr{F}$ is an n-ary function symbol.

(ii) A *C-rule* (w.r.t. K) is a rule $c \to d$, where c and d are constants in K.

In Definition 2(i), note that $f \in \mathscr{F}$ can also be a 0-ary function symbol (i.e. a constant).

Example 2. *Let* $E = \{f(x_1,x_2) \approx f(x_2,x_1), g(x_1,x_2,x_3) \approx g(x_2,x_1,x_3)\}$. *If* $\mathscr{F} = \{a,b,h,f,g\}$ *with* $\mathscr{F}_E = \{f,g\}$ *and* $P = \{f(b,g(b,a,a)) \approx h(a)\}$, *then* $D_0 = \{a \to c_0, b \to c_1, g(c_1,c_0,c_0) \to c_2, f(c_1,c_2) \to c_3, h(c_0) \to c_4\}$ *is a possible set of D-rules over* $\mathscr{F}$, *and we have* $K = \{c_0,c_1,c_2,c_3,c_4\}$. *Using* D_0, *we can simplify the original equations in P, which gives the set of C rules, i.e.,* $C_0 = \{c_3 \to c_4\}$ *where* $c_3 \succ c_4$.

Definition 3. Let E be a finite set of permutation equations and K be a set of constants disjoint from $\mathscr{F}$. A ground rewrite system $R = D \cup C$ is a *congruence closure modulo* E (w.r.t. $\mathscr{F}$ and K) if the following conditions are met:

(i) D is a set of D-rules and C is a set of C-rules, and for each constant $c \in K$, there exists at least one ground term $t \in \mathscr{T}(\mathscr{F})$ such that $t \overset{*}{\leftrightarrow}_{R,E} c$.

(ii) R,E is a ground convergent (modulo E) rewrite system over $\mathscr{T}(\mathscr{F} \cup K)$.

In addition, given a set of ground equations P over $\mathscr{T}(\mathscr{F} \cup K)$, R is said to be a *congruence closure modulo* E (w.r.t. $\mathscr{F}$ and K) *for* P if for all ground terms s and t over $\mathscr{T}(\mathscr{F})$, $s \overset{*}{\leftrightarrow}_{P \cup E} t$ iff there are ground terms u and v over $\mathscr{T}(\mathscr{F} \cup K)$ such that $s \overset{*}{\rightarrow}_{R,E} u \overset{*}{\leftrightarrow}_E v \overset{*}{\leftarrow}_{R,E} t$.

In the following, by *B-rules with the interpreted function symbol* $g \in \mathscr{F}$, we mean either the idempotency rule (I): $\{g(x,x) \to x\}$ or the nilpotency rule (N): $\{g(x,x) \to 0\}$ or the unit rule (U): $\{g(x,0) \to x, g(0,x) \to x\}$ or $I \cup U$ or $N \cup U$.

Definition 4. Let E be a finite set of permutation equations and K be a set of constants disjoint from $\mathscr{F}$. A ground rewrite system $R = D \cup C$ is a *congruence closure modulo* $E \cup B$ (w.r.t. $\mathscr{F}$ and K) if the following conditions are met:

(i) B is a set of B-rules with the interpreted function symbol $g \in \mathscr{F}$.[1]

(ii) D is a set of D-rules and C is a set of C-rules, and for each constant $c \in K$, there exists at least one ground term $t \in \mathscr{T}(\mathscr{F})$ such that $t \overset{*}{\leftrightarrow}_{R,E} c$.

(iii) $R \cup B, E$ is a convergent (modulo E) rewrite system over $\mathscr{T}(\mathscr{F} \cup K, \mathscr{X})$.[2]

In addition, given a set of ground equations P over $\mathscr{T}(\mathscr{F} \cup K)$, R is said to be a *congruence closure modulo* $E \cup B$ (w.r.t. $\mathscr{F}$ and K) *for* P if for all ground terms s and t over $\mathscr{T}(\mathscr{F})$, $s \overset{*}{\leftrightarrow}_{P \cup B \cup E} t$ iff there are ground terms u and v over $\mathscr{T}(\mathscr{F} \cup K)$ such that $s \overset{*}{\rightarrow}_{R \cup B,E} u \overset{*}{\leftrightarrow}_E v \overset{*}{\leftarrow}_{R \cup B,E} t$.

Note that B or E can be empty in Definition 4. If B is empty, then it is the same as Definition 3. Also, condition (ii) in Definition 4 states that each constant c in K represents some term in $\mathscr{T}(\mathscr{F})$ w.r.t. R,E, meaning that K contains no superfluous constants (cf. [7]).

Definition 5. We denote by W the infinite set of constants $\{c_0,c_1,\ldots\}$ such that W is disjoint from $\mathscr{F}$, and denote by K a finite subset chosen from W. We define orderings $\succ_K$ on K, and $\succ$ and $\succ_{lpo}$ on $\mathscr{T}(\mathscr{F} \cup K)$ as follows:

(i) $c_i \succ_K c_j$ if $i < j$ for all $c_i, c_j \in K$.

(ii) $c \succ d$ if $c \succ_K d$, and $t \succ c$ if $t \to c$ is a D-rule.

[1] If $g \in \mathscr{F}_E$, then g is a commutative function symbol, i.e., $g(x_1,x_2) \approx g(x_2,x_1) \in E$.

[2] In this paper, $R \cup B, E$ (resp. $R \cup B/E$) denotes $(R \cup B), E$ (resp. $(R \cup B)/E$).

(iii) $\succ_{lpo}$ is a lexicographic path ordering on $\mathscr{T}(\mathscr{F}\cup K)$, which can be defined from the following assumptions:
(iii.1) $c \succ_{lpo} d$ if $c \succ_K d$,
(iii.2) $t \succ_{lpo} c$ if t is any term headed by a function symbol f in $\mathscr{F}$ and c is any constant in K, and
(iii.3) there is a total precedence on symbols in $\mathscr{F}$.

Observe that $\succ_{lpo}$ extends $\succ$, and is total on $\mathscr{T}(\mathscr{F}\cup K)$. (If the precedence on $\mathscr{F}\cup K$ is total, then the associated lexicographic path ordering $\succ_{lpo}$ is total on $\mathscr{T}(\mathscr{F}\cup K)$ (see [11]).) We emphasize that a partial ordering $\succ$ on $\mathscr{T}(\mathscr{F}\cup K)$ suffices for inference rules in Figure 1.

Figure 1 shows the inference rules for congruence closure modulo permutation equations, which extends the inference rules for the abstract congruence closure framework in [7]. We have the additional inference rule called the REWRITE rule in Figure 1. Also, we use the E-equality $\approx_E$ instead of the equality $\approx$ for the DEDUCE and DELETE inference rules. We write $(K,P,R)\vdash(K',P',R')$ to indicate that (K',P',R') can be obtained from (K,P,R) by application of an inference rule in Figure 1, where K denotes a set of new constants (see Definition 5), P a set of equations, and R a set of rewrite rules consisting of C-rules and D-rules. Also, in Figure 1, B denotes a set of B-rules. A *derivation* is a sequence of states $(K_0,P_0,R_0)\vdash(K_1,P_1,R_1)\vdash\cdots$.

Lemma 6. *If $(K,P,R)\vdash(K',P',R')$, then for all u and v in $\mathscr{T}(\mathscr{F}\cup K)$, we have $u \overset{*}{\leftrightarrow}_{E\cup B\cup P'\cup R'} v$ if and only if $u \overset{*}{\leftrightarrow}_{E\cup B\cup P\cup R} v$.*

Proof. We consider each application of an inference rule τ for $(K,P,R)\vdash(K',P',R')$. If τ is EXTEND, SIMPLIFY, ORIENT, DELETE, COLLAPSE, or COMPOSE, then the conclusion can be verified similarly to [5, 7].

If τ is REWRITE, then we let $P=\bar{P}$, $R=\bar{R}\cup\{l'\to r'\}$, $R'=\bar{R}$, $P'=\bar{P}\cup\{r\sigma\approx r'\}$, and $K=K'$. Since $(K\cup P\cup R)-(K'\cup P'\cup R')=\{l'\to r'\}$, we need to show that $l' \overset{*}{\leftrightarrow}_{E\cup B\cup P'\cup R'} r'$. As $l'=l\sigma\to_B r\sigma\leftrightarrow_{P'} r'$, we have $l' \overset{*}{\leftrightarrow}_{E\cup B\cup P'\cup R'} r'$. Conversely, since $(K'\cup P'\cup R')-(K\cup P\cup R)=\{r\sigma\approx r'\}$, we need to show that $r\sigma \overset{*}{\leftrightarrow}_{E\cup B\cup P\cup R} r'$. As $r\sigma\leftarrow_B l\sigma=l'\to_R r'$, we have $r\sigma \overset{*}{\leftrightarrow}_{E\cup B\cup P\cup R} r'$.

If τ is DEDUCE, then let $R=\bar{R}\cup\{s\to c, t\to d\}$, $P'=P\cup\{c\approx d\}$, $R'=\bar{R}\cup\{t\to d\}$, and $K=K'$, where $s\approx_E t$. Since $(K\cup P\cup R)-(K'\cup P'\cup R')=\{s\to c\}$, we need to show that $s \overset{*}{\leftrightarrow}_{E\cup B\cup P'\cup R'} c$. As $s \overset{*}{\leftrightarrow}_E t\to_{R'} d\leftrightarrow_{P'} c$, we have $s \overset{*}{\leftrightarrow}_{E\cup B\cup P'\cup R'} c$. Conversely, since $(K'\cup P'\cup R')-(K\cup P\cup R)=\{c\approx d\}$, we need to show that $c \overset{*}{\leftrightarrow}_{E\cup B\cup P\cup R} d$. As $c\leftarrow_R s \overset{*}{\leftrightarrow}_E t\to_R d$, we have $c \overset{*}{\leftrightarrow}_{E\cup B\cup P\cup R} d$. □

Definition 7. (i) A derivation is said to be *fair* if any inference rule that is continuously enabled is applied eventually.
(ii) By a *fair μ-derivation*, we mean that the EXTEND and SIMPLIFY rule are applied eagerly in a fair derivation.

Theorem 8. *Let $(K_0,P_0,R_0)\vdash(K_1,P_1,R_1)\vdash\cdots$ be a fair μ-derivation such that P_0 is a finite set of ground equations with $K_0=\emptyset$ and $R_0=\emptyset$.*
(i) Each fair μ-derivation starting from the initial state (K_0,P_0,R_0) is finite.
(ii) If (K_n,P_n,R_n) is a final state (i.e. no inference rule can be applied to (K_n,P_n,R_n)) of a fair μ-derivation starting from the initial state (K_0,P_0,R_0), then $R_n\cup B,E$ is convergent modulo E, and R_n is a congruence closure modulo $E\cup B$ for P_0.

Proof. Since $(K_0,P_0,R_0)\vdash(K_1,P_1,R_1)\vdash\cdots$ is a fair μ-derivation, this derivation can be written as $(K_0,P_0,R_0)\vdash^*(K_m,P_m,R_m)\vdash(K_{m+1},E_{m+1},R_{m+1})\vdash\cdots$, where the derivation $(K_m,P_m,R_m)\vdash(K_{m+1},E_{m+1},R_{m+1})\vdash\cdots$ does not involve any application of the EXTEND rule, so we have the set $K_m=K_{m+1}=\cdots$.

EXTEND: $$\frac{(K, P[t], R)}{(K \cup \{c\}, P[c], R \cup \{t \to c\})}$$

if $t \to c$ is a D-rule, $c \in W - K$, and t occurs in some equation in P.

SIMPLIFY: $$\frac{(K, P[t], R \cup \{t \to c\})}{(K, P[c], R \cup \{t \to c\})}$$

if t occurs in some equation in P.[3]

REWRITE: $$\frac{(K, P, R \cup \{l' \to r'\})}{(K, P \cup \{r\sigma \approx r'\}, R)}$$

if $l' = l\sigma$, where $l \to r \in B$.

ORIENT: $$\frac{(K, P \cup \{s \approx t\}, R)}{(K, P, R \cup \{s \to t\})}$$

if $s \succ t$, and $s \to t$ is a D-rule or a C-rule.

DEDUCE: $$\frac{(K, P, R \cup \{s \to c\,, t \to d\})}{(K, P \cup \{c \approx d\}, R \cup \{t \to d\})}$$

if $s \approx_E t$.

DELETE: $$\frac{(K, P \cup \{s \approx t\}, R)}{(K, P, R)}$$

if $s \approx_E t$.

COMPOSE: $$\frac{(K, P, R \cup \{t \to c\,, c \to d\})}{(K, P, R \cup \{t \to d\,, c \to d\})}$$

COLLAPSE: $$\frac{(K, P, R \cup \{t[c] \to c'\,, c \to d\})}{(K, P, R \cup \{t[d] \to c'\,, c \to d\})}$$

if c is a proper subterm of t and $c \to d$ is a C-rule.

Figure 1: Inference rules for congruence closure modulo permutation equations

For (i), we provide a more concrete result in the following Lemma 9.

For (ii), let (K_n, P_n, R_n) be a final state of a fair μ-derivation starting from the state (K_0, P_0, R_0). (Note that each fair μ-derivation starting from the initial state (K_0, P_0, R_0) is finite by (i), so we have some final state.) Observe that $P_m, P_{m+1}, \ldots$ either contains only C-equations or is empty, and $\succ$ can orient those C equations, so $P_n = \emptyset$. Since $l \succ_{lpo} r$ for all rules $l \to r \in R_n \cup B$, we see that $R_n \cup B/E$ is terminating.

Also, since $R_n \cup B$ is non-overlapping w.r.t. the rewrite system $R_n \cup B, E$ (i.e. there is no non-trivial critical pair between rules in $R_n \cup B$), $R_n \cup B, E$ is Church-Rosser modulo E by the critical pair lemma [5]. Thus, $R_n \cup B, E$ is convergent modulo E.

Finally, we show that for each constant $c \in K$, there exists at least one ground term $t \in \mathscr{T}(\mathscr{F})$ such that $t \overset{*}{\leftrightarrow}_{R_n,E} c$ by induction. Let c be a constant in K and $f(c_1, \ldots, c_k) \to c$ be the corresponding extension rule for c when c was added. By induction hypothesis, we have $s_i \overset{*}{\leftrightarrow}_{R_n,E} c_i$, and thus $f(s_1, \ldots, s_k) \overset{*}{\leftrightarrow}_{R_n,E} f(c_1, \ldots, c_k) \to_{\cup_i R_i} c$. By Lemma 6, we also have $f(s_1, \ldots, s_k) \overset{*}{\leftrightarrow}_{R_n \cup P_n B \cup E} c$, and thus $f(s_1, \ldots, s_k) \overset{*}{\leftrightarrow}_{R_n \cup B \cup E} c$ because $P_n = \emptyset$. As $R_n \cup B, E$ is convergent modulo E and no more REWRITE rule can be applied to $f(s_1, \ldots, s_k)$ by fairness of the derivation, we have $f(s_1, \ldots, s_k) \overset{*}{\leftrightarrow}_{R_n,E} c$. Thus R_n is a congruence closure modulo $E \cup B$ for P_0. □

In the following lemma, recall that function symbols in $\mathscr{F}$ include 0-ary function symbols in $\mathscr{F}$, i.e., constants in $\mathscr{F}$.

Lemma 9. *Let $(K_0, P_0, R_0) \vdash (K_1, P_1, R_1) \vdash \cdots$ be a fair μ-derivation such that P_0 is a finite set of ground equations with $K_0 = \emptyset$ and $R_0 = \emptyset$. Then its derivation length is bounded by $O(n^2)$, where n is the sum of the sizes (number of symbols) of the left-hand and right-hand sides of equations in P_0.*

Proof. We show that the number of applications of each rule in Figure 1 in a fair μ-derivation is bounded above by $O(n^2)$, where n is the sum of the sizes (number of symbols in $\mathscr{F}$) of the left-hand and right-hand sides of equations in P_0. Since $(K_0, P_0, R_0) \vdash (K_1, P_1, R_1) \vdash \cdots$ is a fair μ-derivation, we may write this derivation as

$$(K_0, P_0, R_0) \vdash^* (K_m, P_m, R_m) \vdash (K_{m+1}, E_{m+1}, R_{m+1}) \vdash \cdots,$$

where the derivation $(K_m, P_m, R_m) \vdash (K_{m+1}, E_{m+1}, R_{m+1}) \vdash \cdots$ does not involve any application of the EXTEND rule, and thus we have the finite set $K_m = K_{m+1} = \cdots$.

(i) The total number of the EXTEND inference steps is bounded by $O(n)$. This is because the total number of $\mathscr{F}$-symbols in the second component of the state is not increasing for each transition step[4] and each EXTEND inference step decreases this number by one.

(ii) A derivation step by the SIMPLIFY, REWRITE, DEDUCE, COMPOSE, or COLLAPSE rule either reduces the number of function symbols of $\mathscr{F}$ in $R_i \cup P_i$ or rewrites some constant. The length of a rewriting sequence $c_1 \to c_2 \to \cdots$ is bounded by $|K_m|$. (Here, $|K_m|$ is $O(n)$ because the total number of the EXTEND inference steps is bounded by $O(n)$ as discussed in (i).) Also, the total number of symbols in $P_i \cup R_i$ is bounded by $O(n + |K_m|)$,[5] which is also $O(n)$. This means that the total number of the SIMPLIFY, DEDUCE, COLLAPSE, or COMPOSE inference steps is bounded

[4] The only exception is the case where the REWRITE inference step using the nilpotency rule introduces constant 0 in the second component of the state, where 0 does not occur in P_0. But this requires at most one additional EXTEND inference step.

[5] The total number of symbols in $P_i \cup R_i$ for each transition step does not increase except by an EXTEND inference step, where an EXTEND inference step may increase this number by two.

by $O(n^2)$. (Note that rewriting constants takes $O(n^2)$ because there are at most $O(n)$ constants, and the length of a rewriting sequence for each constant is bounded by $O(n)$.)

(iii) The total number of the DELETE inference steps is bounded by $O(n+|K_m|)$ (i.e. $O(n)$) because the total number of symbols in $P_i \cup R_i$ is bounded by $O(n+|K_m|)$.

(iv) The total number of the ORIENT inference steps is bounded by the total number of EXTEND, SIMPLIFY, DEDUCE, COLLAPSE, and COMPOSE inference steps, which is $O(n^2)$. Note that each ORIENT inference step neither increases the number of function symbols nor the number of constants.

Thus, the derivation length of any fair μ-derivation starting from (K_0, P_0, R_0) is bounded by $O(n^2)$. □

Given a finite (fixed) set of permutation equations E and two terms $s = f(s_1, \ldots, s_k)$ and $t = f(t_1, \ldots, t_k)$ with $f \in \mathscr{F}_E$, we can determine whether $s \approx_E t$ in $O(n^2)$ time (measured in $n = |s| + |t|$) using an additional data structure (i.e. a table) that can be constructed in polynomial time [1]. If s and t are both flat, then we can determine whether $s \approx_E t$ in $O(n)$ time using the following procedure with a table that can be constructed in polynomial time (see [1]).

Equality-Test(s, t)
Input: $s = f(c_1, \ldots, c_i)$ and $t = g(d_1, \ldots, d_j)$, where s and t are both flat.
Output: If $s \approx_E t$, then return true. Otherwise, return false.

1. Determine whether s and t are headed by the same function symbol (i.e. $f = g$ and thus $i = j$). If not, then return false. If it is true, then consider the following:
2. Determine whether $f \in \mathscr{F}_E$. If not, then s and t are compared by syntactic equality, and return true if they are syntactically equal. Otherwise, if $f \in \mathscr{F}_E$, then consider the following:
3. Determine whether $s \approx_E t$ using the *TestEq* procedure in [1].

It is easy to see that steps 1 and 2 of the *Equality-Test*(s, t) procedure take at most $O(n)$ time. For step 3, which corresponds to the case $f = g$ and $f \in \mathscr{F}_E$, it takes $O(n)$ time for comparing two multisets.

If they are equal, then s and t are further compared in constant time using the *TestEq* procedure in [1] with a table that can be constructed in polynomial time. (Note that the arity of all $f \in \mathscr{F}_E$ and the size of the data structure (i.e. table) is bounded by a constant independent of the size of the input terms.) Therefore, the *Equality-Test*(s, t) procedure takes $O(n)$ time using a table that can be constructed in polynomial time. In what follows, we denote by $Table(Eq(f))$ this table for each $f \in \mathscr{F}_E$ for a finite (fixed) set of permutation equations E.

Theorem 10. *Given the table $Table(Eq(f))$ for each $f \in \mathscr{F}_E$, a congruence closure modulo $E \cup B$ for a finite set of ground equations P can be computed in $O(n^3)$ time, where n is the sum of the sizes (number of symbols) of the left and right sides of equations in P.*

Proof. We first construct a fair μ-derivation $(K_0, P_0, R_0) \vdash (K_1, P_1, R_1) \vdash \cdots$ such that P_0 is a finite set of ground equations with $K_0 = \emptyset$ and $R_0 = \emptyset$.

It is easy to see that each EXTEND, SIMPLIFY, ORIENT, COMPOSE, and COLLAPSE inference step in the derivation takes $O(n)$ time.

Each REWRITE inference step in the derivation takes $O(n)$ time because we only need to consider for rules $g(x,x) \to x$, $g(x,x) \to 0$, $g(x,0) \to x$, and $g(0,x) \to x$ for some interpreted function symbol $g \in \mathscr{F}$.

Each DEDUCE inference step in the derivation takes $O(n)$ time for checking E-equality (see the

Equality-Test(s, t) procedure) between two left-hand side terms s and t in R_i. Similarly, each DELETE inference step in the derivation takes $O(n)$ time for checking E-equality.

By Lemma 9, we know that the derivation length of a fair μ-derivation is bounded by $O(n^2)$. Since each inference step in the derivation takes $O(n)$ time, a congruence closure modulo $E \cup B$ for a finite set of ground equations P can be computed in $O(n^3)$ time. □

Corollary 11. *The word problem for a finite set of ground equations P with a fixed set of permutation function symbols is decidable in polynomial time.*

Proof. We can decide whether $s \approx^?_E t$ for two ground terms s and t using a congruence closure modulo E for P. By Theorem 10, we can compute a congruence closure modulo E for P in polynomial time by constructing and using the table $Table(Eq(f))$ for each $f \in \mathscr{F}_E$. Let R be a congruence closure modulo E for P. We obtain each normal form of s and t using R. We first rewrite each constant symbol in $\mathscr{F}$ of s and t to a new constant symbol in K obtained from constructing R, which takes $O(m)$ time where $m = |s| + |t|$. Each rewrite step either reduces the size of a term or rewrites a constant in K to another constant in K. The length of a rewriting sequence $c_1 \to c_2 \to \cdots$ is bounded by $|K|$ (i.e. $O(n)$), where n is the sum of the sizes of the left-hand and right-hand sides of equations in P. We may also infer that the sum of the sizes of the left-hand and right-hand sides of the rewrite rules in R is $O(n+|K|)$, which is $O(n)$. Each rewrite step takes at most $O(n^2)$ time using R and the *Equality-Test* procedure. By combining these steps together, we can decide whether $s \approx^?_E t$ for two ground terms s and t using their normal forms in polynomial time. □

The above corollary also holds if some function symbols (not necessarily permutation function symbols) satisfies the properties, such as idempotency (I), nilpotency (N), unit (U), $I \cup U$, or $N \cup U$.

4 Example of congruence closure modulo $E \cup B$

Let B be the set of the equation for an idempotency function symbol g, i.e., $B = \{g(x,x) \to x\}$ and let E be the following set of permutation equations:

$$\begin{aligned} E = \{ & f(x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8) \approx f(x_2,x_1,x_3,x_4,x_5,x_6,x_7,x_8), \\ & f(x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8) \approx f(x_2,x_3,x_4,x_1,x_5,x_6,x_7,x_8), \\ & f(x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8) \approx f(x_1,x_2,x_3,x_4,x_6,x_5,x_7,x_8), \\ & f(x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8) \approx f(x_1,x_2,x_3,x_4,x_5,x_6,x_8,x_7)\}. \end{aligned}$$

In this example, we may view each variable x_i as a switch in a specially designed electric board, where each variable will be assigned to either constant T (representing "on") or constant F (representing "off"). Each ground term $f(c_1,c_2,c_3,c_4,c_5,c_6,c_7,c_8)$ with $c_i = T$ or F represents a certain state of this electric board. There is a special transformation button in this electric board, which may transform one state to another state of the electic board. This transformation button is represented by a function with symbol $h \notin \mathscr{F}_E$. The problem is to determine if a certain state in the electric board (represented by a term) generates a fault state (represented by term $\perp$). We see that $\prod[E] = \{(12),(1234),(56),(78)\}$, which means that $f(x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8) \approx_E f(x_{\rho(1)},x_{\rho(2)},x_{\rho(3)},x_{\rho(4)},x_5,x_6,x_7,x_8)$ for any permutation ρ on the set $\{1,2,3,4\}$, $f(x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8) \approx_E f(x_1,x_2,x_3,x_4,x_{\pi(5)},x_{\pi(6)},x_7,x_8)$ for any permutation π on the set $\{5,6\}$, and $f(x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8) \approx_E f(x_1,x_2,x_3,x_4,x_5,x_6,x_{\tau(7)},x_{\tau(8)})$ for any permutation τ

on the set $\{7,8\}$ (see Thereom 1). Therefore, eight switches in the board are partitioned into three components, i.e. $\{x_1,x_2,x_3,x_4\}$, $\{x_5,x_6\}$ and $\{x_7,x_8\}$, where the order of "switch on" or "switch off" does not matter in each component. For example, $f(T,T,F,F,T,F,T,F) \approx_E f(F,F,T,T,F,T,T,F)$. Meanwhile, g is an idempotent function symbol, which serves as a comparator for fault states. For example, if $g(\bot, f(F,F,F,T,T,T,T,F))$, then it is $\bot$ if $f(F,F,F,T,T,T,T,F)$ is $\bot$. Now we start with the following set of ground equations:

1. $f(T,T,T,T,T,T,T,T) \approx \bot$
2. $h(f(F,F,F,F,F,F,F,F)) \approx f(F,T,F,T,F,T,F,T)$
3. $f(T,F,F,F,F,F,F,T) \approx g(\bot, h(f(T,T,T,T,F,T,F,T)))$
4. $h(f(T,F,T,F,T,F,T,F)) \approx f(F,F,F,F,T,T,T,T)$
5. $f(F,F,F,F,T,T,T,T) \approx f(T,T,T,T,F,F,F,F)$
6. $h(f(T,T,T,T,F,F,F,F)) \approx f(T,T,T,T,T,F,T,F)$
7. $h(f(T,T,T,T,F,T,F,T)) \approx f(T,T,T,T,T,T,T,T)$

We show that, for example, each of $h^4(f(F,F,F,F,F,F,F,F))$ and $f(T,F,F,F,F,F,F,T)$ is a fault state. (For notational brevity, by $h^i(t)$, we mean the function symbol h is applied to term $h^{i-1}(t)$ with $h^0(t)$ denoting t.) The initial state is (K_0,P_0,R_0), where $K_0 = R_0 = \emptyset$ and P_0 consists of the above equations $1-7$. We apply a fair μ-derivation starting with (K_0,P_0,R_0) and some intermediate and repetitive steps are omitted for clarity. In the following, each rewrite rule is an element of some R_i and each equation is an element of some P_j. We assume that $c_i \succ c_j$ if $i < j$.

$1(a)$.	$T \to c_1, F \to c_2, \bot \to c_3$	EXTEND and SIMPLIFY for 1
$1(b)$.	$f(c_1,c_1,c_1,c_1,c_1,c_1,c_1,c_1) \to c_4$	
$1(c)$.	$c_4 \approx c_3$	
$2(a)$.	$f(c_2,c_2,c_2,c_2,c_2,c_2,c_2,c_2) \to c_5$	EXTEND and SIMPLIFY for 2
$2(b)$.	$h(c_5) \to c_6$	
$2(c)$.	$f(c_2,c_1,c_2,c_1,c_2,c_1,c_2,c_1) \to c_7$	
$2(d)$.	$c_6 \approx c_7$	
$3(a)$.	$f(c_1,c_2,c_2,c_2,c_2,c_2,c_2,c_1) \to c_8$	EXTEND and SIMPLIFY for 3
$3(b)$.	$f(c_1,c_1,c_1,c_1,c_2,c_1,c_2,c_1) \to c_9$	
$3(c)$.	$h(c_9) \to c_{10}$	
$3(d)$.	$g(c_3,c_{10}) \to c_{11}$	
$3(e)$.	$c_8 \approx c_{11}$	
$4(a)$.	$f(c_1,c_2,c_1,c_2,c_1,c_2,c_1,c_2) \to c_{12}$	EXTEND and SIMPLIFY for 4
$4(b)$.	$h(c_{12}) \to c_{13}$	
$4(c)$.	$f(c_2,c_2,c_2,c_2,c_1,c_1,c_1,c_1) \to c_{14}$	
$4(d)$.	$c_{13} \approx c_{14}$	
$5(a)$.	$f(c_2,c_2,c_2,c_2,c_1,c_1,c_1,c_1) \to c_{15}$	EXTEND and SIMPLIFY for 5
$5(b)$.	$f(c_1,c_1,c_1,c_1,c_2,c_2,c_2,c_2) \to c_{16}$	
$5(c)$.	$c_{15} \approx c_{16}$	
$6(a)$.	$f(c_1,c_1,c_1,c_1,c_2,c_2,c_2,c_2) \to c_{17}$	EXTEND and SIMPLIFY for 6
$6(b)$.	$h(c_{17}) \to c_{18}$	
$6(c)$.	$f(c_1,c_1,c_1,c_1,c_1,c_2,c_1,c_2) \to c_{19}$	
$6(d)$.	$c_{18} \approx c_{19}$	
$7(a)$.	$f(c_1,c_1,c_1,c_1,c_2,c_1,c_2,c_1) \to c_{20}$	EXTEND and SIMPLIFY for 7

$7(b)$.	$h(c_{20}) \to c_{21}$	
$7(c)$.	$f(c_1,c_1,c_1,c_1,c_1,c_1,c_1,c_1) \to c_{22}$	
$7(d)$.	$c_{21} \approx c_{22}$	
$8(a)$.	$c_7 \approx c_{12}$ (Rule 2(c) is now removed.)	DEDUCE with 2(c) and 4(a)
$8(b)$.	$c_{14} \approx c_{15}$ (Rule 4(c) is now removed.)	DEDUCE with 4(c) and 5(a)
$8(c)$.	$c_{16} \approx c_{17}$ (Rule 5(b) is now removed.)	DEDUCE with 5(b) and 6(a)
$8(d)$.	$c_9 \approx c_{20}$ (Rule 3(b) is now removed.)	DEDUCE with 3(b) and 7(a)
$8(e)$.	$c_{19} \approx c_{20}$ (Rule 6(c) is now removed.)	DEDUCE with 6(c) and 7(a)
$8(f)$.	$c_4 \approx c_{22}$ (Rule 1(b) is now removed.)	DEDUCE with 1(b) and 7(c)

We next orient equations into C-rules and apply other inference rules. The set of C-rules is $C = \{c_3 \to c_4, c_6 \to c_7, c_8 \to c_{11}, c_{13} \to c_{14}, c_{15} \to c_{16}, c_{18} \to c_{19}, c_{21} \to c_{22}, c_7 \to c_{12}, c_{14} \to c_{15}, c_{16} \to c_{17}, c_9 \to c_{20}, c_{19} \to c_{20}, c_4 \to c_{22}\}$. Using DEDUCE, COMPOSE, and ORIENT inference steps, it becomes $C' = \{c_3 \to c_{22}, c_6 \to c_{12}, c_8 \to c_{11}, c_{13} \to c_{17}, c_{15} \to c_{17}, c_{18} \to c_{20}, c_{21} \to c_{22}, c_7 \to c_{12}, c_{14} \to c_{17}, c_{16} \to c_{17}, c_9 \to c_{20}, c_{19} \to c_{20}, c_4 \to c_{22}\}$. The REWRITE inference step 9(d) is available after the following inference steps $9(a)$, $9(b)$, and $9(c)$:

$9(a)$.	$h(c_{20}) \to c_{10}$	COLLAPSE 3(c) with $c_9 \to c_{20}$
$9(b)$.	$c_{10} \to c_{22}$	DEDUCE with 7(b) and 9(a), ORIENT, COMPOSE
$9(c)$.	$g(c_{22},c_{22}) \to c_{11}$	COLLAPSE 3(d) with $c_3 \to c_{22}$ and $c_{10} \to c_{22}$
$9(d)$.	$c_{11} \to c_{22}$	REWRITE 9(c), ORIENT

In the above, Rule $3(c)$ is removed after $9(a)$, Rule $9(a)$ is removed after $9(b)$, Rule $3(d)$ is removed after $9(c)$, and Rule $9(c)$ is removed after $9(d)$. We may obtain a congruence closure $R_n = C_n \cup D_n$ modulo $E \cup B$ for P_0 for some n with some additional inference steps, where $C_n = \{c_3 \to c_{22}, c_4 \to c_{22}, c_6 \to c_{12}, c_7 \to c_{12}, c_8 \to c_{11}, c_9 \to c_{20}, c_{10} \to c_{22}, c_{11} \to c_{22}, c_{13} \to c_{17}, c_{14} \to c_{17}, c_{15} \to c_{17}, c_{16} \to c_{17}, c_{18} \to c_{20}, c_{21} \to c_{22}\}$ and D_n consists of the following set of rules:

D1: $T \to c_1$	D2: $F \to c_2$
D3: $\bot \to c_{22}$	D4: $f(c_2,c_2,c_2,c_2,c_2,c_2,c_2,c_2) \to c_5$
D5: $h(c_5) \to c_{12}$	D6: $f(c_1,c_2,c_2,c_2,c_2,c_2,c_2,c_1) \to c_{22}$
D7: $f(c_1,c_2,c_1,c_2,c_1,c_2,c_1,c_2) \to c_{12}$	D8: $h(c_{12}) \to c_{17}$
D9: $f(c_2,c_2,c_2,c_2,c_1,c_1,c_1,c_1) \to c_{17}$	D10: $f(c_1,c_1,c_1,c_1,c_2,c_2,c_2,c_2) \to c_{17}$
D11: $h(c_{17}) \to c_{20}$	D12: $f(c_1,c_1,c_1,c_1,c_2,c_1,c_2,c_1) \to c_{20}$
D13: $h(c_{20}) \to c_{22}$	D14: $f(c_1,c_1,c_1,c_1,c_1,c_1,c_1,c_1) \to c_{22}$

Now we determine whether $h^4(f(F,F,F,F,F,F,F,F))$ is a fault state: i.e., $h^4(f(F,F,F,F,F,F,F,F)) \approx^?_{R_n \cup B \cup E} \bot$. Since $h^4(f(F,F,F,F,F,F,F,F)) \xrightarrow{*}_{R_n,E} h^4(f(c_2,c_2,c_2,c_2,c_2,c_2,c_2,c_2)) \to_{R_n,E} h^4(c_5) \to_{R_n,E} h^3(c_{12}) \to_{R_n,E} h^2(c_{17}) \to_{R_n,E} h(c_{20}) \to_{R_n,E} c_{22}$ and $\bot \to_{R_n,E} c_{22}$, it is a fault state. Similarly, we can determine whether $f(T,F,F,F,F,F,F,T)$ is a fault state. Since $f(T,F,F,F,F,F,F,T) \xrightarrow{*}_{R_n,E} f(c_1,c_2,c_2,c_2,c_2,c_2,c_2,c_1) \to_{R_n,E} c_{11} \to_{R_n,E} c_{22}$ and $\bot \to_{R_n,E} c_{22}$, it is a fault state. Meanwhile, $h^2(f(F,F,T,T,F,T,F,T))$ is not a fault state, i.e., $h^2(f(F,F,T,T,F,T,F,T)) \not\approx_{R_n \cup B \cup E} \bot$. Since $h^2(f(F,F,T,T,F,T,F,T)) \xrightarrow{*}_{R_n \cup B,E} h^2(f(c_2,c_2,c_1,c_1,c_2,c_1,c_2,c_1)) \to_{R_n \cup B,E} h^2(c_{12}) \to_{R_n \cup B,E} h(c_{17}) \to_{R_n \cup B,E} c_{20}$ and $\bot \to_{R_n \cup B,E} c_{22}$, it is not a fault state.

5 Conclusion

We have presented a framework for constructing congruence closure modulo a finite set of permutation equations E, extending the abstract congruence closure framework for handling permutation function symbols with or without the interpreted function symbols (not necessarily permutation function symbols) satisfying each of the following properties: idempotency (I), nilpotency (N), unit (U), $I \cup U$, or $N \cup U$. We have provided a polynomial time decision procedure for the word problem for a finite set of ground equations with a fixed set of permutation function symbols by constructing congruence closure modulo E.

Although congruence closure procedures have been widely used in software/hardware verfication and satisfiability modulo theories (SMT) solvers, congruence closure procedures with built-in permutations have not been well studied. We believe that our framework for constructing congruence closure modulo permutation equations has practical significance to software/hardware verfication and SMT solvers involving built-in permutations, where built-in permutations are represented by a finite set of permutation equations containing permutation function symbols.

References

[1] Jürgen Avenhaus (2004): *Efficient Algorithms for Computing Modulo Permutation Theories*. In David Basin & Michaël Rusinowitch, editors: *Automated Reasoning - Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4–8*, Springer, Berlin, Heidelberg, pp. 415–429, doi:10.1007/978-3-540-25984-8_31.

[2] Jürgen Avenhaus & David A. Plaisted (2001): *General Algorithms for Permutations in Equational Inference*. *Journal of Automated Reasoning* 26(3), pp. 223–268, doi:10.1023/A:1006439522342.

[3] Franz Baader & Deepak Kapur (2020): *Deciding the Word Problem for Ground Identities with Commutative and Extensional Symbols*. In Nicolas Peltier & Viorica Sofronie-Stokkermans, editors: *Automated Reasoning*, Springer International Publishing, Cham, pp. 163–180, doi:10.1007/978-3-030-51074-9_10.

[4] Franz Baader & Tobias Nipkow (1998): *Term Rewriting and All That*. Cambridge University Press, Cambridge, UK, doi:10.1017/CBO9781139172752.

[5] Leo Bachmair (1991): *Canonical Equational Proofs*. Birkhäuser, Boston, doi:10.1007/978-1-4684-7118-2.

[6] Leo Bachmair, IV Ramakrishnan, Ashish Tiwari & Laurent Vigneron (2000): *Congruence Closure Modulo Associativity and Commutativity*. In Hélène Kirchner & Christophe Ringeissen, editors: *Frontiers of Combining Systems*, Springer, Berlin, Heidelberg, pp. 245–259, doi:10.1007/10720084_16.

[7] Leo Bachmair, Ashish Tiwari & Laurent Vigneron (2003): *Abstract congruence closure*. *Journal of Automated Reasoning* 31(2), pp. 129–168, doi:10.1023/B:JARS.0000009518.26415.49.

[8] Clark Barrett & Cesare Tinelli (2018): *Satisfiability Modulo Theories*, pp. 305–343. Springer International Publishing, Cham, doi:10.1007/978-3-319-10575-8_11.

[9] David Cyrluk, Sreeranga Rajan, Natarajan Shankar & Mandayam K. Srivas (1995): *Effective theorem proving for hardware verification*. In Ramayya Kumar & Thomas Kropf, editors: *Theorem Provers in Circuit Design*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 203–222, doi:10.1007/3-540-59047-1_50.

[10] Leonardo De Moura & Nikolaj Bjørner (2011): *Satisfiability modulo theories: introduction and applications*. *Communications of the ACM* 54(9), pp. 69–77, doi:10.1145/1995376.1995394.

[11] Nachum Dershowitz & David A. Plaisted (2001): *Rewriting*. In: *Handbook of Automated Reasoning*, chapter 9, Volume I, Elsevier, Amsterdam, pp. 535 – 610, doi:10.1016/b978-044450813-3/50011-4.

[12] Peter J. Downey, Ravi Sethi & Robert Endre Tarjan (1980): *Variations on the Common Subexpression Problem*. *J. ACM* 27(4), p. 758–771, doi:10.1145/322217.322228.

[13] Thomas W. Hungerford (1980): *Algebra*. Springer, New York, NY, doi:10.1007/978-1-4612-6101-8.

[14] Deepak Kapur (1997): *Shostak's Congruence Closure as Completion*. In Hubert Comon, editor: *Rewriting Techniques and Applications*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 23–37, doi:10.1007/3-540-62950-5_59.

[15] Deepak Kapur (2019): *Conditional Congruence Closure over Uninterpreted and Interpreted Symbols*. *Journal of Systems Science and Complexity* 32, pp. 317–355, doi:10.1007/s11424-019-8377-8.

[16] Deepak Kapur (2021): *A Modular Associative Commutative (AC) Congruence Closure Algorithm*. In: *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, Buenos Aires, Argentina (Virtual Conference), July 17–24*, 195, LIPIcs, pp. 15:1–15:21, doi:10.4230/LIPIcs.FSCD.2021.15.

[17] Dohan Kim & Christopher Lynch (2021): *An RPO-based ordering modulo permutation equations and its applications to rewrite systems*. In: *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, Buenos Aires, Argentina (Virtual Conference), July 17–24*, 195, LIPIcs, pp. 19:1–19:17, doi:10.4230/LIPIcs.FSCD.2021.19.

[18] Dexter Kozen (1977): *Complexity of Finitely Presented Algebras*. In John E. Hopcroft, Emily P. Friedman & Michael A. Harrison, editors: *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, ACM, pp. 164–177, doi:10.1145/800105.803406.

[19] Greg Nelson & Derek C. Oppen (1980): *Fast Decision Procedures Based on Congruence Closure*. *J. ACM* 27(2), p. 356–364, doi:10.1145/322186.322198.

[20] Vilhelm Sjöberg & Stephanie Weirich (2015): *Programming up to Congruence*. *SIGPLAN Not.* 50(1), p. 369–382, doi:10.1145/2676726.2676974.

First-Order Logic in Finite Domains: Where Semantic Evaluation Competes with SMT Solving

Wolfgang Schreiner*
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University Linz, Austria
Wolfgang.Schreiner@risc.jku.at

Franz-Xaver Reichl†
Algorithms and Complexity Group
TU Wien, Austria
freichl@ac.tuwien.ac.at

In this paper, we compare two alternative mechanisms for deciding the validity of first-order formulas over finite domains supported by the mathematical model checker RISCAL: first, the original approach of "semantic evaluation" (based on an implementation of the denotational semantics of the RISCAL language) and, second, the later approach of SMT solving (based on satisfiability preserving translations of RISCAL formulas to SMT-LIB formulas as inputs for SMT solvers). After a short presentation of the two approaches and a discussion of their fundamental pros and cons, we quantitatively evaluate them, both by a set of artificial benchmarks and by a set of benchmarks taken from real-life applications of RISCAL; for this, we apply the state-of-the-art SMT solvers Boolector, CVC4, Yices, and Z3. Our benchmarks demonstrate that (while SMT solving generally vastly outperforms semantic evaluation), the various SMT solvers exhibit great performance differences. More important, we identify classes of formulas where semantic evaluation is able to compete with (or even outperform) satisfiability solving, outlining some room for improvements in the translation of RISCAL formulas to SMT-LIB formulas as well as in the current SMT technology.

1 Introduction

The aim of the RISCAL system [16] is to support the analysis of theories and algorithms over discrete domains, as they arise in computer science, discrete mathematics, logic, and algebra. For this purpose, RISCAL provides an expressive specification language based on a strongly typed variant of first-order logic in which theories can be formulated and algorithms can be specified; nevertheless the validity of all formulas and the correctness of all algorithms is decidable. This is because all RISCAL types have finite sizes which are configurable by model parameters. Therefore, a RISCAL model actually represents an infinite set of finite models; before verifying the validity of a theorem over the infinite model set by a deductive proof in some theorem proving environment, we can check its validity over selected finite instances of the set by model checking in RISCAL. The system has been mainly developed for educational purposes [15] but it has also been applied in research [19].

The basic mechanism of RISCAL for deciding the validity of formulas and the correctness of algorithms is "semantic evaluation", which is based on a constructive implementation of the denotational semantics of all kinds of syntactic phrases allowed by the language. However, since 2020, the system also provides an alternative (and potentially much more efficient) decision mechanism based on SMT (satisfiability modulo theories) solving, implemented by the second author [14, 17]. In this approach, the decision of a RISCAL formula is performed via a translation to a formula in the SMT-LIB language [3] and the application of some external SMT solver (currently, the SMT solvers Boolector, CVC4, Yices,

*Supported by the JKU Linz LIT Project LOGTECHEDU and by the Aktion Österreich-Slowakei Project 2019-10-15-003.
†Supported by the Austrian Science Fund (FWF) under grant W1255.

T. Kutsia (Ed.): Symbolic Computation
in Software Science (SCSS'21)
EPTCS 342, 2021, pp. 99–113, doi:10.4204/EPTCS.342.9

and Z3 are supported). Indeed, this has achieved great performance improvements [14] and allowed to constructively work with theories that were out of reach of semantic evaluation.

Actually, SMT solvers have served for a long time as backends of various program verification tools, for real languages such as Java [1] as well as for algorithmic languages such as Dafny [10] where the verification backend Boogie [2] generates SMT-LIB conditions that are discharged by the SMT solver Z3. Furthermore, SAT/SMT solvers are applied for the analysis of system modeling languages such as Alloy , Event-B, and VDM. Last but not least, they are employed as backends for interactive provers, e.g., in Isabelle's "sledgehammer" component [4] and in Coq's SMTCoq plugin [7].

As for the translation of higher-level specification languages into the languages of satisfiability solvers, [9] describes the techniques used in the Alloy Analyzer to transform formulas from first-order relational logic; [21] discusses improvements implemented in the SAT-based relational model finder Kodkod. On top of Kodkod, the counterexample generator Nitpick [5] generates finite countermodels of Isabelle formulas by a translation to relational logic. In [8], it is briefly sketched how Alloy constraints have been (manually) translated into the language of the SMT solver Yices, which shows drastic speedups when tautologies are decided. [11] sketches the encoding of VDM proof obligations as SMT problems proved with the SMT solver Z3. In somewhat more detail, [6] discusses the implementation of a SMT plugin for the Event-B platform Rodin and experimentally compares this plugin with those for other provers.

However, as beneficial SMT solving in general is, in our own work of deciding RISCAL formulas we also have regularly encountered cases where the performance of the SMT-based decision is comparatively poor, sometimes even beaten by semantic evaluation. While some potential reasons have already been outlined in [14], a more systematic analysis and evaluation has been lacking so far. Also the work reported in the scientific literature is a bit unsatisfactory in this respect: while the relative merits of SMT solvers are regularly evaluated in the SMT-COMP competition series [20], it is harder to find comparisons with alternative decision mechanisms such as the one with the Vampire prover presented in [13].

In this paper, we provide a detailed comparison of the built-in decision mechanism of RISCAL by semantic evaluation with the corresponding decisions by SMT solving. In particular, we identify classes of situations where the performance of SMT-based decisions is relatively low, i.e., where indeed "semantic evaluation competes with SMT solving", as a starting point for potential improvements in the SMT-LIB translation of RISCAL and in SMT technology in general. While our insights are clearly limited to the particular strategy applied for translating RISCAL formulas to SMT-LIB, our work shows that SMT is not a panacea in all kinds of reasoning problems but has to be applied with some caveats.

Closest to our work is [12], where the untyped first-order logic of Lamport's language TLA$^+$ is translated to SMT-LIB conditions that are discharged by the SMT solvers CVC4 and Z3; however the results have not been experimentally compared with the built-in TLC model checker. As another difference, the TLA$^+$ translation heavily relies on a nonconstructive encoding of non-integer values by uninterpreted sorts and functions with corresponding background axioms. In contrast to this, the RISCAL translation generates formulas over bit vectors with uninterpreted sort and function symbols, which minimizes the use of uninterpreted functions by a constructive encoding of all types as bit vectors.

The remainder of this paper is organized as follows: In Section 2, we outline the decision mechanisms applied in RISCAL. In Section 3, we present the artificial benchmarks which we use to compare both mechanisms. In Section 4, we extend these investigations to a selected set of benchmarks taken from real-life applications of RISCAL. In Section 5, we present our conclusions derived from these investigations and outline possible strands of further research in RISCAL and SMT technology. Appendix A includes detailed illustrations of the benchmark results. More details can be found in the technical report [18] on which this paper is based. Due to space restrictions, we have to refer the reader to [16] for an overview on the RISCAL language which is elucidated by a tutorial and reference manual and various publications.

2 Deciding First-Order Formulas

In the following, we briefly describe the two alternative mechanisms that RISCAL implements for deciding first-order formulas: internal semantic evaluation and the application of external SMT solvers.

Semantic Evaluation The built-in decision mechanism of RISCAL is based on the translation of every syntactic phrase of the RISCAL language into an executable representation of its denotational semantics. This representation is a Java "lambda expression" that in essence maps an assignment for the free variables of the phrase to the value denoted by its semantics, i.e., the truth value of a formula or the updated variable assignment resulting from the execution of a command [17]. In the case of first-order logic formulas, the most interesting part of the translation is that of a universally quantified formula $\forall x : D.\ F$ and that of an existentially quantified formula $\exists x : D.\ F$, respectively; these translations are semi-formally sketched below (here $[\![F]\!]$ denotes the body of a function whose execution yields the truth value of F):

```
⟦∀x: D. F⟧ :=
  e := enumerate(D)
  loop
    if empty(e) then return true
    x := next(e); e := rest(e)
    if ¬call(⟦F⟧,x) then return false
```

```
⟦∃x: D. F⟧ :=
  e := enumerate(D)
  loop
    if empty(e) then return false
    x := next(e); e := rest(e)
    if call(⟦F⟧,x) then return true
```

The core of the translation is a loop that enumerates every element of the domain D of the quantified variable x and evaluates the body of the quantified formula with x bound to that element, until the truth value of the body determines the overall result. As an optimization, RISCAL actually implements the enumeration of D in a mostly "lazy" fashion such that it is not necessary to simultaneously keep all elements in memory; the generation stops when the first element has been produced that allows to decide the formula. Consequently, the "worst case" is exhibited by a *true universal formula* or a *false existential formula*: here we have to generate all elements, before we can decide that the universal formula is true or the existential formula is false.

Furthermore, RISCAL supports expressions that do not denote unique values, for example the term (choose $x : D$ with $F[x]$) that denotes any value x of the domain D that satisfies the formula $F[x]$. RISCAL implements such a term in its "nondeterministic" evaluation mode [17] by the computation of a (lazily evaluated) *stream* of such values. Like for quantified formulas, the core of this translation is a loop that enumerates every element of D; the translation yields each element that satisfies the body formula as a value of the expression (i.e., this value is appended to a stream of values denoted by the term). A formula that depends on such terms correspondingly denotes a stream of truth values; the formula is only considered as valid if this stream only consists of instances of truth value "true". Not necessarily unique choices arise in many mathematical definitions and algorithms ("choose any element e of set S"). Furthermore, applications of such expressions may emerge from the modular verification of user-defined operations; here not the definition of an operation but its contract is considered. For instance, an application $f(a)$ of a function f specified as

$$\mathsf{fun}\ f(x : D) :\ D\ \mathsf{ensures}\ F[x, result]$$

(where *result*, is a special variable that denotes the result of the function) can be replaced by the expression (choose $result : D.\ F[a, result]$).

SMT Solving The problem of deciding the validity of a formula F, denoted as $valid[\![F]\!]$, can be reduced to the problem of deciding the satisfiability of the negation of F, denoted as $sat[\![\neg F]\!]$, by applying the equivalence. $valid[\![F]\!] \equiv \neg sat[\![\neg F]\!]$. RISCAL implements a translation to formulas in the SMT-LIB format [3]; thus we can decide the validity of the RISCAL formula F by letting an external SMT solver decide the satisfiability of the SMT-LIB version of $\neg F$. As a background theory we have chosen the theory of *fixed-size bit vectors*: since every RISCAL domain is finite, every element of a domain with n elements can be represented by a vector of $\lceil \log n \rceil$ bits; furthermore the set of bit vector operations is expressive enough to allow a proper encoding of the various RISCAL operations. However, bit vectors alone are not enough: the treatment of quantifiers and choose expressions (discussed below) requires functions which are not explicitly characterized by definitions but only implicitly by axioms; therefore we demand from the theory also support for *uninterpreted functions*. Furthermore, the main SMT-LIB logic that provides bit vectors and uninterpreted function is the logic QF_UFBV of "unquantified formulas over bit vectors with uninterpreted sort and function symbols" which is supported, e.g., by the well known SMT solvers Boolector, CVC4, Yices, and Z3. We therefore have to translate a RISCAL formula with quantifiers into a corresponding quantifier-free SMT-LIB formula (since some SMT solvers actually support as a non-standard extension also quantified bit vector formulas with uninterpreted functions, we will in the benchmarks later also experiment with the preservation of quantifiers).

The problem of eliminating quantifiers is addressed by the following equivalences which semi-formally sketch how quantifiers can be removed from RISCAL formulas (the role of function f is explained below):

$$\begin{aligned} valid[\![\exists x : D.\ F[x]]\!] &\equiv \neg sat[\![\neg \exists x : D.\ F[x]]\!] \\ &\equiv \neg sat[\![\forall x : D.\ \neg F[x]]\!] \equiv \neg sat[\![\neg F[e_1] \wedge \ldots \wedge \neg F[e_n]]\!] \\ valid[\![\forall x : D.\ F[x]]\!] &\equiv \neg sat[\![\neg \forall x : D.\ F[x]]\!] \\ &\equiv \neg sat[\![\exists x : D.\ \neg F[x]]\!] \equiv \neg sat[\![\neg F[f(x_1, \ldots, x_n)]]\!] \end{aligned}$$

We assume that before the translation is applied all formulas have been transformed into *negation normal form*, i.e., all applications of the negation symbol have been *pushed inside* down to the level of atomic formulas. Thus, above occurrences of quantified formulas are *positive*, i.e., they do not appear in the context of negation. Then the decision of $valid[\![\exists x : D.\ F[x]]\!]$ boils down to the decision of the satisfiability of the universally quantified formula $\forall x : D.\ \neg F[x]$. Now, if D consists of n values denoted by terms $e_1, \ldots, e_n$, we can expand the quantified formula to an equivalent conjunction $\neg F[e_1] \wedge \ldots \wedge \neg F[e_n]$. On the other hand, the decision of $valid[\![\forall x : D.\ F[x]]\!]$ boils down to the decision of the satisfiability of the existentially quantified formula $\exists x : D.\ \neg F[x]$. Analogously to the previous case, we could in principle also expand this formula, namely to a disjunction $\neg F[e_1] \vee \ldots \vee \neg F[e_n]$. However, for this kind of decision we generally prefer another option that avoids the blow-up of the formula. Let us assume that the existentially quantified formula appears in the context of n universally quantified variables $x_1, \ldots, x_m$, i.e., the problem of deciding the satisfiability of $\exists x : D.\ \neg F[x]$ actually occurs in the course of deciding the satisfiability of a global formula of the shape $\forall x_1 : D_1.\ \ldots \forall x_m : D_m.\ \ldots \exists x : D.\ \neg F[x]$. Then we introduce an m-ary function symbol f that does not appear anywhere else in the global formula; the denoted function can therefore have an arbitrary interpretation (we call such a function a *Skolem function*). Finally, we replace $\exists x : D.\ \neg F[x]$ by $\neg F[f(x_1, \ldots, x_m)]$. In the special case $m = 0$, i.e., if there is no outer universally quantified variable, f becomes a *Skolem constant* and the formula becomes $\neg F[f]$. Although the resulting formula is not logically equivalent to the original one, it is *equi-satisfiable*, i.e., it is satisfiable if and only if the original formula is (if we may choose for all values $x_1, \ldots, x_m$ a value for x that makes $F[x]$ true, then from these choices we may construct the Skolem function f and vice versa). Since the translation preserves satisfiability, the equivalence stated above holds.

From the above translation, deciding the validity of an existentially quantified formula may blow-up the formula to a size that is exponential in the depth of the nesting of existential quantifiers; this may also increase the complexity of the decision. Furthermore, problems may arise even with the decision of the validity of a universally quantified formula, which entails deciding the satisfiability of a formula $\neg F[f(x_1,\ldots,x_m)]$ with Skolem function f (please note that $x_1,\ldots,x_m$ represent concrete values, the translated formula does not have any free variables). In the translation to the SMT-LIB theory QF_UFBV, the range of f is not anymore the original RISCAL domain D of the existentially quantified variable, but some bit vector type B whose values encode the values of D. Since not every bit vector in B necessarily represents an element from D, we have to constrain the range of f by a predicate p_D that holds for a bit vector $b \in B$ if and only if b actually represents an element from D. We achieve this by adding to the translation an axiom

$$\bigwedge_{d_1,\ldots,d_m} p_D(f(t(d_1),\ldots,t(d_m)))$$

where $t(d)$ represents an expression that denotes the bit vector associated to the RISCAL value d. The size of this conjunction is proportional to the number of possible combinations of values $d_1,\ldots,d_m$ for the arguments of f; this may blow up the SMT-LIB translation considerably and overcome the benefits of applying Skolemization rather than expansion. Thus the translation can be configured to apply a heuristic: if the number of conjuncts in the Skolemization axiom is significantly larger than the number of conjuncts derived from expanding the original formula, the translation forsakes Skolemization in favor of expansion.

While expressions denoting unique values can be directly encoded by bit vectors operations, an (choose $y\colon D.\ F[x,y]$) with free variable $x\colon D$ gives in the SMT-LIB translation rise to a new function $f:\ D \to D$ with axiom $\forall x\colon D.\ F[x,f(x)]$. Similarly, in modular verification, every RISCAL operation (function, predicate, procedure) specified by a contract gives rise to an SMT-LIB function with a corresponding axiomatization. As explained above, such axiomatizations by universally quantified formulas yield large SMT-LIB expansions and potentially costly SMT decisions (in addition to the user-defined axiomatization, such functions have also to be constrained by the type representation axioms explained above). However, in certain contexts we may replace applications of such axiomatized functions. For instance, take the formula $\forall a.\ (\ldots f(a)\ldots)$ where application $f(a)$ occurs positively (unnegated) in a context $(\ldots\ f(a)\ldots)$ that does not embed $f(a)$ in another quantifier and assume that function $f:\ D \to D$ has been axiomatized as described above. Then above formula can be transformed to the equi-satisfiable formula $\forall a,b.\ (F[a,b] \Rightarrow \ldots b\ldots)$. Thus, we have replaced the application $f(a)$ of axiomatized function f by a fresh universally quantified variable b with assumption $F[a,b]$. This means that the original axiomatization (which applied formula F to arbitrary values x from D) has been specialized to the instances that are actually relevant. RISCAL optionally implements in the SMT-LIB translation a generalized form of this transformation under the name "eliminate choices", because it is directly applied to choose expressions given by the user and to choose expressions generated from applications of implicitly defined functions. In combination with the option "inline definitions", also choose expressions indirectly arising from the definitions of operations may be inlined. While this expands the size of the core formula to be decided, it removes general axiomatizations and may thus be beneficial all in all.

Comparison As discussed above, deciding by SMT solving formulas with quantifiers or choose expressions can become problematic, because the SMT-LIB translation may yield vastly expanded formulas (arising from existential formulas, quantified constraints of Skolem functions, and axioms of uninterpreted functions emerging from choose expressions or modular verification). To which extent this affects the actual performance of the decision process can be investigated only by actual benchmarks.

3 Artificial Benchmarks

Basic Setup We start by investigating the "base behavior" of the two decision approaches. For this, we use the following two predicates:

$$\textit{cycle4-valid} \equiv \neg(x_1 < x_2 \wedge x_2 < x_3 \wedge x_3 < x_4 \wedge x_4 < x_1)$$
$$\textit{cycle4-sat1} \equiv \neg(x_1 < x_2 \wedge x_2 < x_3 \wedge x_3 < x_4 \wedge x_4 < x_1 + 4)$$

Both predicates have free occurrences of four integer variables x_1, x_2, x_3, x_4. Predicate *cycle4-valid* states that these variables cannot form a "less-than cycle"; this predicate is valid and its negation is unsatisfiable. On the other side, predicate *cycle4-sat1* is satisfiable but not valid, as is its negation. However, while *cycle4-sat1* has many satisfying assignments, its negation has only few; thus *cycle4-sat1* represents a "mostly valid" formula, while its negation denotes a "mostly unsatisfiable" one. Both predicates only depend on the atomic predicate $<$ (the second one also on the constant addition $+4$). The predicates do not require any complex calculations or decisions in order to most clearly exhibit the effect of various forms of quantification structures on the decision process. Here we investigate the eight quantification patterns $\exists^4\forall^0$, $\exists^3\forall^1$,$\exists^2\forall^2$, $\exists^1\forall^3$, $\forall^4\exists^0$, $\forall^3\exists^1$, $\forall^2\exists^2$, $\forall^1\exists^3$ where Q^i represents the i-fold repetition of quantifier Q and the variables are quantified in the order x_1, x_2, x_3, x_4. Thus, e.g., the combination of quantification pattern $\exists^3\forall^1$ with predicate *cycle4-valid* represents the following formula:

$$\exists x_1 : D, x_2 : D, x_3 : D.\ \forall x_4 : D.\ \neg(x_1 < x_2 \wedge x_2 < x_3 \wedge x_3 < x_4 \wedge x_4 < x_1)$$

Above quantifier patterns consider the cases of purely existential formulas ($\exists^4\forall^0$), purely universal formulas ($\forall^4\exists^0$), as well as existential formulas with universal bodies ($\exists^i\forall^j$) and universal formulas with existential bodies ($\forall^j\exists^i$) where the number of corresponding quantifiers represent different sizes of the respective quantification ranges. As for the domain D of the variables, we focus on $D := \mathbb{N}[2^N - 1]$ for some $N \in \mathbb{N}$, i.e., each of the 4 variables holds some natural number up to maximum $2^N - 1$; the total value space thus consists of 2^{4N} elements. In the following benchmarks, we choose $N := 6$, i.e., a value space of size 2^{24}. For a formula of shape $\exists^i\forall^j$ or $\forall^j\exists^i$, this value space is partitioned according to the numbers i and j of existentially and universally quantified variables, respectively. The "existential search space" has size 2^{iN}, which leads in the QF_UFBV translation to the generation of 2^{iN} clauses. The "universal search space" has size 2^{jN}, which leads in QF_UFBV to j Skolem constants (if the universal quantifiers are outermost) or j Skolem functions of arity i (if the universal quantifiers are innermost); the domain of each Skolem constant or function is a bit vector of length N with 2^N possible values.

Experimental Results The four diagrams in Figure 1 plot the decision times for the quantified formulas with (valid) predicate *cycle4-valid* and its (unsatisfiable) negation and for the satisfiable (mostly valid) predicate *cycle4-sat1* and its also satisfiable (but mostly unsatisfiable) negation. The labels of the horizontal axis denote the applied quantification pattern (labels e*i*a*j* and a*i*e*j* denote patterns $\exists^i\forall^j$ and $\forall^i\exists^j$, respectively). The vertical axis denotes the decision time in ms, within the interval $[1, 60000]$ (please note the logarithmic scale). All decision procedures were forcefully terminated after 1 minute; thus, if a plot point is at the top line of the diagram, this actually indicates "timeout" or "no result" (a timeout is also indicated, if the software ran out of memory or produced any other kind of error). All measurements were performed on a virtual GNU/Linux machine with a CPU of type i7-2670QM@2.20GHz using 8 GB RAM. In case of the SMT solvers, only the time for the actual decision (not including the time for translating the RISCAL formula to an SMT-LIB formula) was considered.

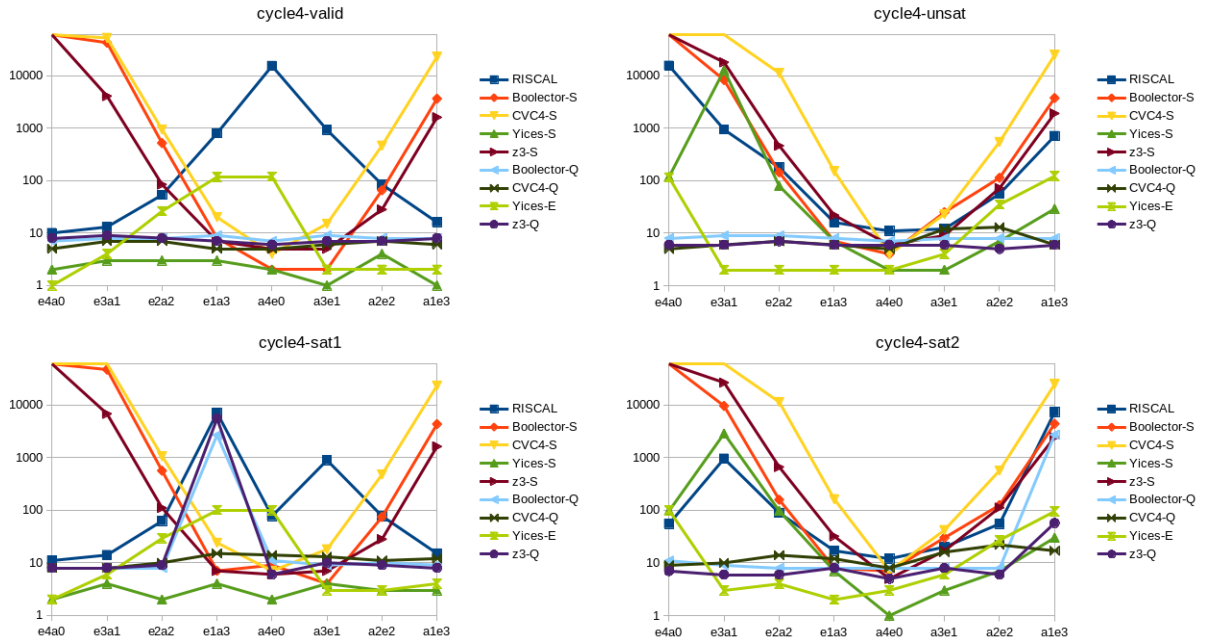


Figure 1: Artificial Benchmarks: Base Behavior

The various labeled curves give the times for the decision mechanisms that we have benchmarked using RISCAL 3.8.5 and the SMT solvers Boolector 3.1.0, CVC4 1.7, Yices 2.6.1, and Z3 4.8.7. Here tag RISCAL represents the built-in semantic evaluation mechanism of RISCAL. Tags Boolector-S, CVC4-S, Yices-S, Z3-S represent the application of the various SMT solvers to the generated QF_UFBV formula where quantifiers are removed by Skolemization (in case of the originally universal quantifiers) or expansion (in case of the existential quantifiers). Tags Boolector-Q, CVC4-Q, Z4-Q represent the application of the SMT solvers where, however, in the generated formula all quantifiers are preserved (Boolector, CVC4, and Z3 also support quantification). Yices-E represents the application of Yices where also the (originally) universal quantifiers have been removed by expansion rather than by Skolemization. Thus every SMT solver is benchmarked twice, with two different mechanisms for dealing with quantifiers: by eliminating them as described in Section 2 to yield a formula in the standard logic QF_UFBV of SMT-LIB, or by applying the non-standard quantification support of the various SMT solvers. Only in the case of Yices (which has only a limited support for quantification), we apply the alternative of expanding also (originally) universal quantifiers.

An inspection of the diagrams shows that, when eliminating quantifiers by Skolemization or expansion, Yices is mostly the fastest among the benchmarked SMT solvers; the other solvers are able to compete with Yices only if quantifiers are preserved in the formulas. The semantic evaluation mechanism is mostly outperformed by Yices and also by the other solvers. However, the other solvers are superior only if the quantifiers are preserved in the formulas, or if we consider the cases in the middle of the left diagrams (few or no existential quantifiers and mainly valid base predicates). When quantifiers are eliminated, the semantic evaluation mechanism of RISCAL outperforms Boolector, CVC4, and Yices at the boundaries of the left diagrams; also in the right diagrams (mostly unsatisfiable base predicates), the performance of semantic evaluation at least matches that of the solvers.

Furthermore, the semantic evaluation mechanism of RISCAL exhibits comparatively good performance in diagram *cycle4-valid* for the quantification pattern $\exists^i\forall^j$. Since the outermost quantifier is existential, only a single value for its variable has to be found that makes the formula true; since the base predicate is valid, already the first choice is successful. The more existential quantifiers follow, i.e., the bigger i is, the bigger the advantage is. If all quantifiers are existential (case $\exists^4\forall^0$), the first attempted choice for all variables already leads to a decision of the formula. However, if more and more variables get universally quantified, the more and more work has to be performed to validate the existential choice. The worst situation arises, if all variables are universally quantified (case $\forall^4\exists^0$); here the full variable space has to be investigated to determine the validity of the formula. However, the more of the inner variables get existentially quantified, the quicker the decision for each value of a universally quantified variable becomes. If only the outermost variable is universally quantified (case $\forall^1\exists^3$), only the space of the outermost variable has to be fully investigated. This explains the shape of the RISCAL curve which grows from the fully existentially quantified formula of type $\exists^4\forall^0$ until it reaches a sharp peak at the fully universally quantified formula of type $\forall^4\exists^0$; then the curve goes down again towards the formula pattern $\forall^1\exists^3$. On the other hand, plot *cycle4-unsat* illustrates the dual behavior for the negated (unsatisfiable) version of the predicate. To show that the fully existentially quantified formula $\exists^4\forall^0$ is false, the whole value space has to be investigated, while for the fully universally quantified formula $\forall^4\exists^0$ the first encountered value combination represents a counterexample to the truth of the formula; for a growing number of inner existential quantifiers, again more value combinations have to be investigated, though. Finally, the two plots for the mostly satisfiable predicate *cycle4-sat1* and its mostly unsatisfiable negation *cycle4-sat2* are similar to the plots for the valid and unsatisfiable cases, except that there is no more a pronounced "peak" (maximum or minimum) for the fully universally quantified pattern $\forall^4\exists^0$: the investigation of the value space can stop when the first counterexample is found, but not necessarily the first value combination encountered immediately represents such a counterexample.

More results are given in Appendix A which illustrates in Figure 4 and Figure 5 corresponding benchmarks with more complex predicates involving operations such as non-linear arithmetic, arithmetic quantification, set and array operations. These benchmarks reveal various situations when the semantic evaluation mechanism of RISCAL is able to compete with or even outperform some of the SMT solvers; for a more detailed interpretation of these and other benchmarks, see [18].

So far, we have only considered formulas with built-in operations (functions and predicates). Now we are going to also consider operations specified by contracts such as the following two functions:

$$\begin{aligned}&\mathsf{fun}\ f(x_1,x_2,x_3,x_4)\ \mathsf{ensures}\\&\quad \mathsf{if}\ x_1<x_2\wedge x_2<x_3\wedge x_3<x_4\wedge x_4<x_1\ \mathsf{then}\ \mathit{result}=0\ \mathsf{else}\ \mathit{result}=1;\\&\mathsf{fun}\ g(x_1,x_2,x_3,x_4)\ \mathsf{ensures}\\&\quad \mathsf{if}\ x_1=x_2\wedge x_3=x_4\ \mathsf{then}\ \mathit{result}=0\ \mathsf{else}\ \mathit{result}=1;\end{aligned}$$

Here $f(x_1,x_2,x_3,x_4)$ is 1 for all x_1,x_2,x_3,x_4 and $g(x_1,x_2,x_3,x_4)$ may be 1 or 0, depending on the values of x_1,x_2,x_3,x_4. We will consider quantified formulas with the quantification patterns used in the previous sections, using four base predicates that test the equality of above functions with values 1 or 0, yielding again one valid, one unsatisfiable, and two satisfiable situations.

Figure 2 displays the decision times for model parameter $N := 5$ (we now use value 5 rather than 6 to compensate the additional quantifier of the function axiom). Here the evaluation mechanism of RISCAL is generally faster than the SMT solvers (indeed Boolector and CVC4 do mostly not deliver any answers within the given time bound); the major exception is Z3 if quantifiers are preserved, which is faster than RISCAL for valid and unsatisfiable formulas. Yices also produces results but is mostly much slower than

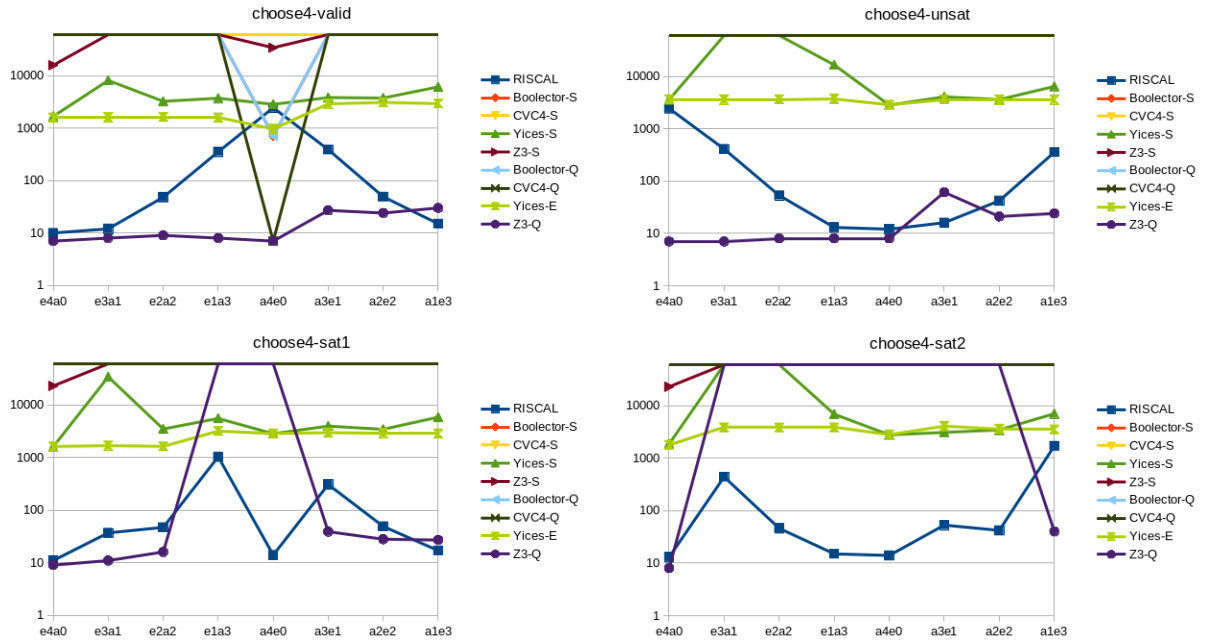


Figure 2: Formulas with Functions Specified by Contracts

RISCAL. Boolector does not support uninterpreted functions if quantifiers are preserved, thus also the benchmark set "Boolector-Q" expands (as "Boolector-S" does) universal and existential quantifiers to conjunctions and disjunctions, respectively.

The results demonstrate that uninterpreted functions characterized by axioms rather than definitions pose a major problem for most SMT solvers; in the presence of such functions often the nondeterministic evaluation mechanism of RISCAL is superior. In an attempt to mitigate this problem a bit, we have applied the previously mentioned options "eliminate choices" and "inline definitions" to eliminate certain applications of contract-specified operations by embedding the postconditions into the enclosing formulas; this does not change the satisfiability of the formula, if the application occurs in a non-negated purely universally quantified context. In our experiments, this is the case (only) for the quantifier pattern a4e0 ($\forall^4\exists^0$) which indeed shows substantial speedups (only) with the application of Yices; the experimental results given above have been derived with these options.

4 Real-Life Benchmarks

The artificial benchmarks investigated so far do not demonstrate whether/how often the observed effects indeed emerge in "real-life" examples. To shed some light on this issue, we have also collected from real RISCAL models a selection of formulas whose validity is to be decided. These formulas mainly represent conditions to validate the specifications of problems, verify the correctness of algorithms, or also theorems over the domains of consideration. In the overwhelming majority of cases, the decision of such conditions by SMT solving vastly outperforms the decision by semantic evaluation. However, for the purpose of this paper, we have explicitly selected a sample where this is not the case, i.e., where semantic evaluation is competitive with SMT solving.

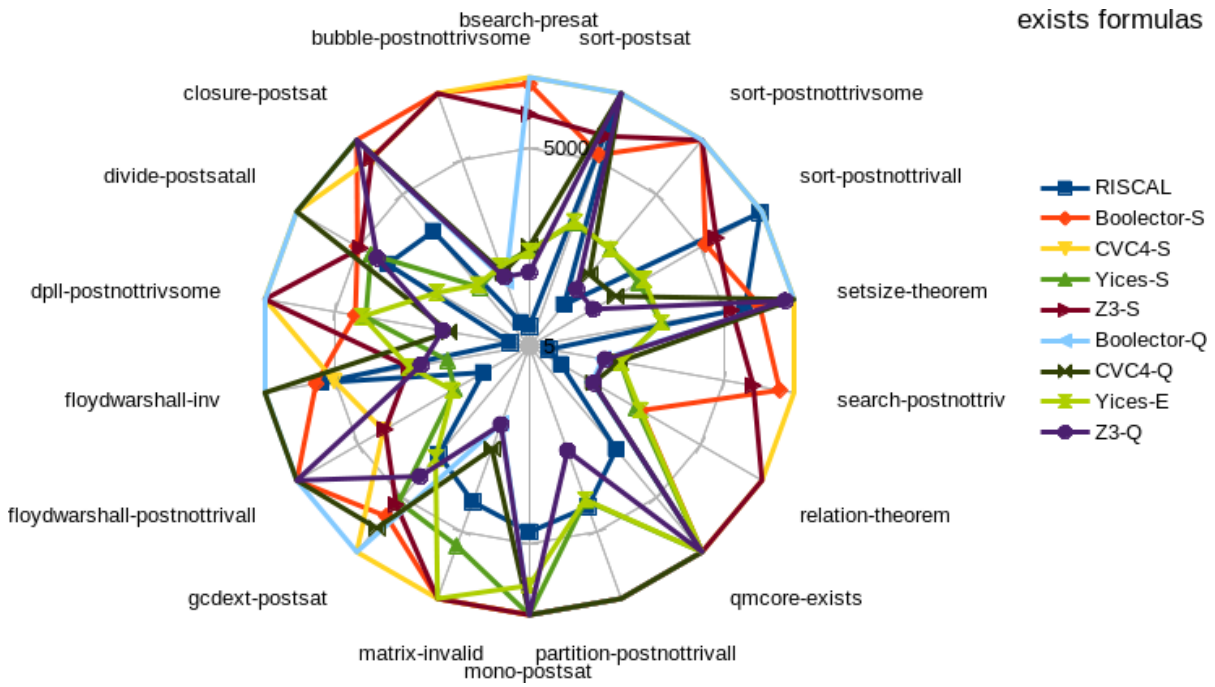


Figure 3: Real-Life Benchmarks

The diagram displayed in Figure 3 presents the results of benchmarking the decision of some of these formulas (see Appendix A or [18] for a link to the sources). The benchmarks are visualized as a circular "net" chart where each radial represents one formula whose validity is to be decided, the center represents some minimal time and the outermost rim of the net represents the 60 seconds timeout limit. Therefore, the closer the lines connecting the benchmark points of a particular decision mechanism are to the center of the net, the faster the decision mechanism is; a line along the outermost rim indicates timeout situations. As in the artificial benchmarks, the values along the radials are plotted in logarithmic magnitudes. In particular, the diagram illustrates benchmarks for formulas that have substantial "existential" content; many stem from validating procedure contracts by checking the satisfiability of the postconditions for all inputs that satisfy the preconditions; here the semantic evaluation of RISCAL often outperforms the various SMT solvers. The best SMT results are achieved by Z3 when preserving quantifiers (Z3-Q) and Yices with either Skolemization or expansion of the (original) universal quantifiers (Yices-S and Yices-E).

Appendix A lists in Figure 6 more benchmarks where the left column illustrates the execution of the benchmarks with smaller values for the model parameters, while the right column illustrates executions with larger values; all executions were again terminated after 60 seconds. The top row gives the benchmarks already shown in Figure 3, but now also with larger model parameters. The second row illustrates benchmarks for formulas with choose expressions; here in the SMT decisions the option "choose elimination" transformation was *not* applied. The evaluation mechanism of RISCAL again outperforms many of the SMT solvers; among these typically Yices performs best. The third row illustrates benchmarks for the same formulas as in the second row but with the options "choose elimination" and "inline definitions" turned on. This shows a clear improvement in many examples; now various SMT solvers clearly beat the semantic evaluation mechanism of RISCAL. The last row illustrates benchmarks for formulas that do not obviously fall into above categories but where nevertheless the semantic evaluation mechanism is competitive, from the structure of the formulas and/or the complexity of the underlying operations.

5 Conclusions

Generally the decision of first-order formulas over finite domains by external SMT solvers via a translation into the SMT-LIB logic QF_UFBV (unquantified formulas over bit vectors with uninterpreted sort and function symbols) and applying external SMT solvers vastly outperforms the semantic evaluation mechanism built-in into RISCAL. In this paper, however, we have also identified cases where this is not necessarily the case, mainly because the SMT-LIB translation leads to a large number of clauses in the generated conjunctive normal form.

One case are theorems with substantial "existential" content, i.e., positive occurrences of existential quantifiers with large quantification ranges: the resulting formulas may be so big that their decision by SMT solving may be outperformed by semantic evaluation. Another case are theorems with uninterpreted functions that are axiomatized by universally quantified formulas with large quantification ranges; also these axioms lead to the generation of SMT-LIB formulas with a huge number of clauses that slow down the execution of SMT solvers. This problem may be partially mitigated, if applications of such functions occur in a pure universal context; an optimization technique may replace the function application by universally quantified variables that are constrained by an appropriate instance of this axiom.

Furthermore, also for universally quantified theorems the problem arises that the Skolem functions generated from their negated counterparts have to be constrained by axioms that describe which bit vector values indeed denote valid RISCAL values. RISCAL therefore implements an SMT-LIB option that applies a heuristic to decide whether it is cheaper to expand the quantified formula rather than to generate a Skolem function. Another problem may result from the necessary encoding of various operations on the data types supported by RISCAL (such as non-linear arithmetic, arithmetic quantifiers, or set size computations) where the built-in evaluation mechanisms of RISCAL may perform better than the corresponding RISCAL encodings; however, while this effect can be observed in artificial benchmarks, it seems not to be a major problem in most real-life examples.

Our work demonstrates that there is still room for improvement in current SMT solvers for the SMT-LIB logic QF_UFBV with respect to deciding theorems with substantial existential content and applications of axiomatized functions. On the other side, we will continue to investigate how the translation of RISCAL formulas to SMT-LIB formulas can be optimized to take the presented findings into account.

References

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reinher Hähnle, Peter H. Schmitt & Mattias Ulbrich, editors (2016): *Deductive Software Verification — The KeY Book — From Theory to Practice*. *LNCS* 10001, Springer International Publishing, Cham, doi:10.1007/978-3-319-49812-6.

[2] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs & K. Rustan M. Leino (2005): *Boogie: A Modular Reusable Verifier for Object-Oriented Programs*. In: *FMCO 2005: Formal Methods for Components and Objects*, *LNCS* 4111, Springer, Berlin, Germany, pp. 364–387, doi:10.1007/11804192_17.

[3] Clark Barrett, Pascal Fontaine & Cesare Tinelli (2016): *The Satisfiability Modulo Theories Library (SMT-LIB)*. Available at `http://www.SMT-LIB.org`.

[4] Jasmin Christian Blanchette, Sascha Böhme & Lawrence C. Paulson (2011): *Extending Sledgehammer with SMT Solvers*. In: *CADE-23: Automated Deduction, 23rd International Conference*, *LNCS* 6803, Springer, Berlin, Germany, pp. 116–130, doi:10.1007/978-3-642-22438-6_11.

[5] Jasmin Christian Blanchette & Tobias Nipkow (2010): *Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder*. In: *ITP 2010: Interactive Theorem Proving*, *LNCS* 6172, Springer, Berlin, Germany, pp. 131–146, doi:10.1007/978-3-642-14052-5_11.

[6] David Déharbe, Pascal Fontaine, Yoann Guyot & Laurent Voisin (2012): *SMT Solvers for Rodin*. In: *ABZ 2012: Abstract State Machines, Alloy, B, VDM, and Z, Third International Conference, Pisa, Italy, June 18–21, 2012*, *LNCS* 7316, Springer, Berlin, Germany, pp. 194–207, doi:10.1007/978-3-642-30885-7_14.

[7] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz et al. (2017): *SMTCoq: A Plug-In for Integrating SMT Solvers into Coq*. In: *CAV 2017: Computer Aided Verification, Heidelberg, Germany, July 24–28, 2017*, *LNCS* 10427, Springer, Cham, Switzerland, pp. 126–133, doi:10.1007/978-3-319-63390-9_7.

[8] Aboubakr Achraf El Ghazi & Mana Taghdiri (2015): *Analyzing Alloy Formulas using an SMT Solver: A Case Study*. AFM10: Automated Formal Methods, July 14, 2010, Edinburgh, UK. Available at `https://arxiv.org/abs/1505.00672`.

[9] Daniel Jackson (2000): *Automating First-Order Relational Logic*. In: *SIGSOFT'00/FSE-8 International Symposium, San Diego, California, USA, November 2000*, *SIGSOFT Software Engineering Notes* 25(6), ACM, New York, NY, USA, pp. 130–139, doi:10.1145/355045.355063.

[10] K. Rustan M. Leino (2010): *Dafny: An Automatic Program Verifier for Functional Correctness*. In Edmund M. Clarke & Andrei Voronkov, editors: *LPAR-16: Logic Programming and Automated Reasoning, 16th International Conference, Dakar, Senegal, April 25–May 1, 2010*, *LNCS* 6355, Springer, Berlin, Germany, pp. 348–370, doi:10.1007/978-3-642-17511-4_20.

[11] Hsin-Hung Lin & Bow-Yaw Wang (2017): *Releasing VDM Proof Obligations with SMT Solvers*. In: *MEMOCODE '17: 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, Vienna, Austria, September 29–October 2, 2017*, ACM, New York, NY, USA, p. 132–135, doi:10.1145/3127041.3127066.

[12] Stephan Merz & Hernán Vanzetto (2016): *Encoding TLA$^+$ into Many-Sorted First-Order Logic*. In: *ABZ 2016: Abstract State Machines, Alloy, B, TLA, VDM, and Z: 5th International Conference, Linz, Austria, May 23–27, 2016*, *LNCS* 9675, Springer, Cham, Switzerland, pp. 54–69, doi:10.1007/978-3-319-33600-8_3.

[13] Giles Reger, Martin Suda & Andrei Voronkov (2017): *Instantiation and Pretending to be an SMT Solver with Vampire*. In: *SMT 2017 Workshop, Heidelberg, Germany, July 22–23, 2017*, *CEUR Workshop Proceedings* 1889, pp. 63–75. Available at `http://ceur-ws.org/Vol-1889/paper6.pdf`.

[14] Franz-Xaver Reichl (2020): *The Integration of SMT Solvers into the RISCAL Model Checker*. Master's thesis, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria. Available at `https://www.risc.jku.at/publications/download/risc_6103/Thesis.pdf`.

[15] Wolfgang Schreiner (2018): *Validating Mathematical Theories and Algorithms with RISCAL*. In: *CICM 2018, 11th Conference on Intelligent Computer Mathematics, Hagenberg, Austria, August 13–17*, *LNCS/Lecture Notes in Artificial Intelligence* 11006, Springer, Berlin, pp. 248–254, doi:10.1007/978-3-319-96812-4_21.

[16] Wolfgang Schreiner (2019): *The RISC Algorithm Language (RISCAL)*. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria. Available at `https://www.risc.jku.at/research/formal/software/RISCAL`.

[17] Wolfgang Schreiner & Franz-Xaver Reichl (2020): *Mathematical Model Checking Based on Semantics and SMT*. *Transactions on Internet Research* 16(2), pp. 4–13. Available at `http://ipsitransactions.org/journals/papers/tir/2020jul/p2.pdf`.

[18] Wolfgang Schreiner & Franz-Xaver Reichl (2021): *Semantic Evaluation versus SMT Solving in the RISCAL Model Checker*. Technical Report 21-11, RISC, Johannes Kepler University, Linz, Austria. Available at `https://www.risc.jku.at/publications/download/risc_6328/21-11.pdf`.

[19] Wolfgang Schreiner, William Steingartner & Valerie Novitzká (2020): *A Novel Categorical Approach to the Semantics of Relational First-Order Logic*. *Symmetry* 12(10), doi:10.3390/sym12101584.

[20] SMT-COMP (2021): *SMT-COMP: The International Satisfiability Modulo Theories (SMT) Competition.* Available at `https://smt-comp.github.io`.

[21] Emina Torlak & Daniel Jackson (2007): *Kodkod: A Relational Model Finder*. In: *TACAS 2007: Tools and Algorithms for the Construction and Analysis of Systems, 3th International Conference, Braga, Portugal, March 24–April 1, 2007*, *LNCS* 4424, Springer, Berlin, Germany, pp. 632–647, doi:10.1007/978-3-540-71209-1_49.

A Benchmark Diagrams

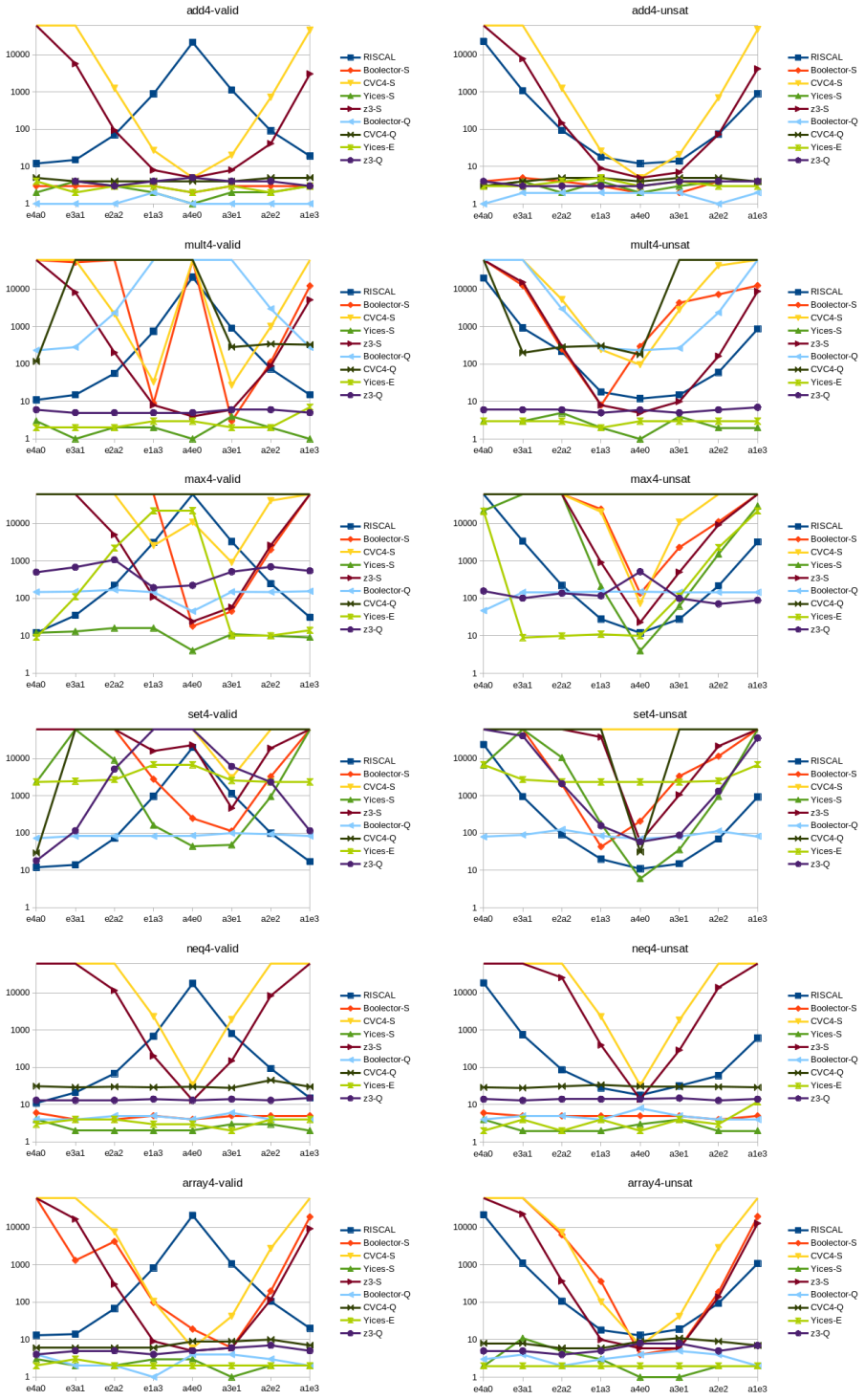


Figure 4: Artificial Benchmarks: Valid versus Unsatisfiable Predicates

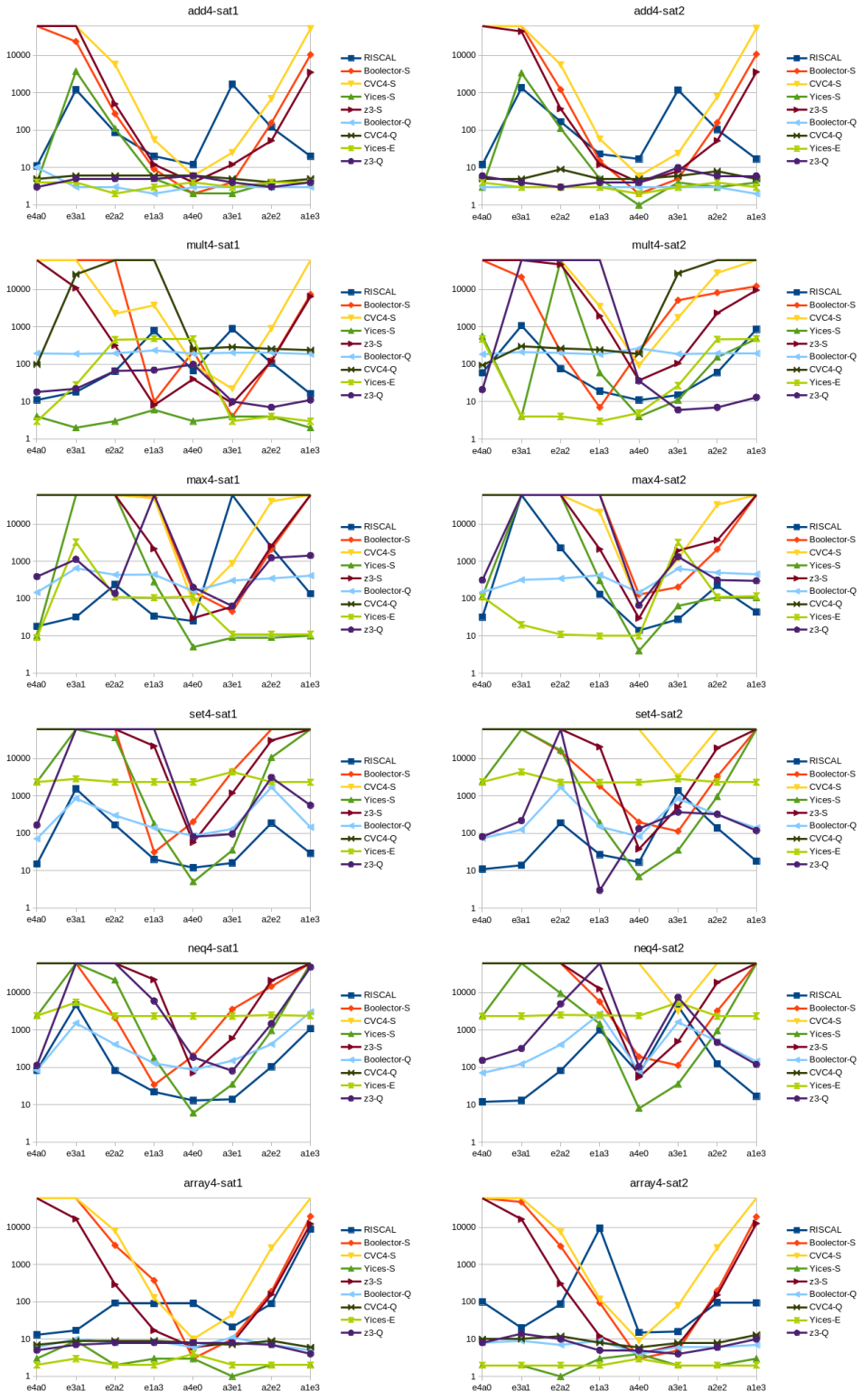


Figure 5: Artificial Benchmarks: Satisfiable Predicates and their Negations

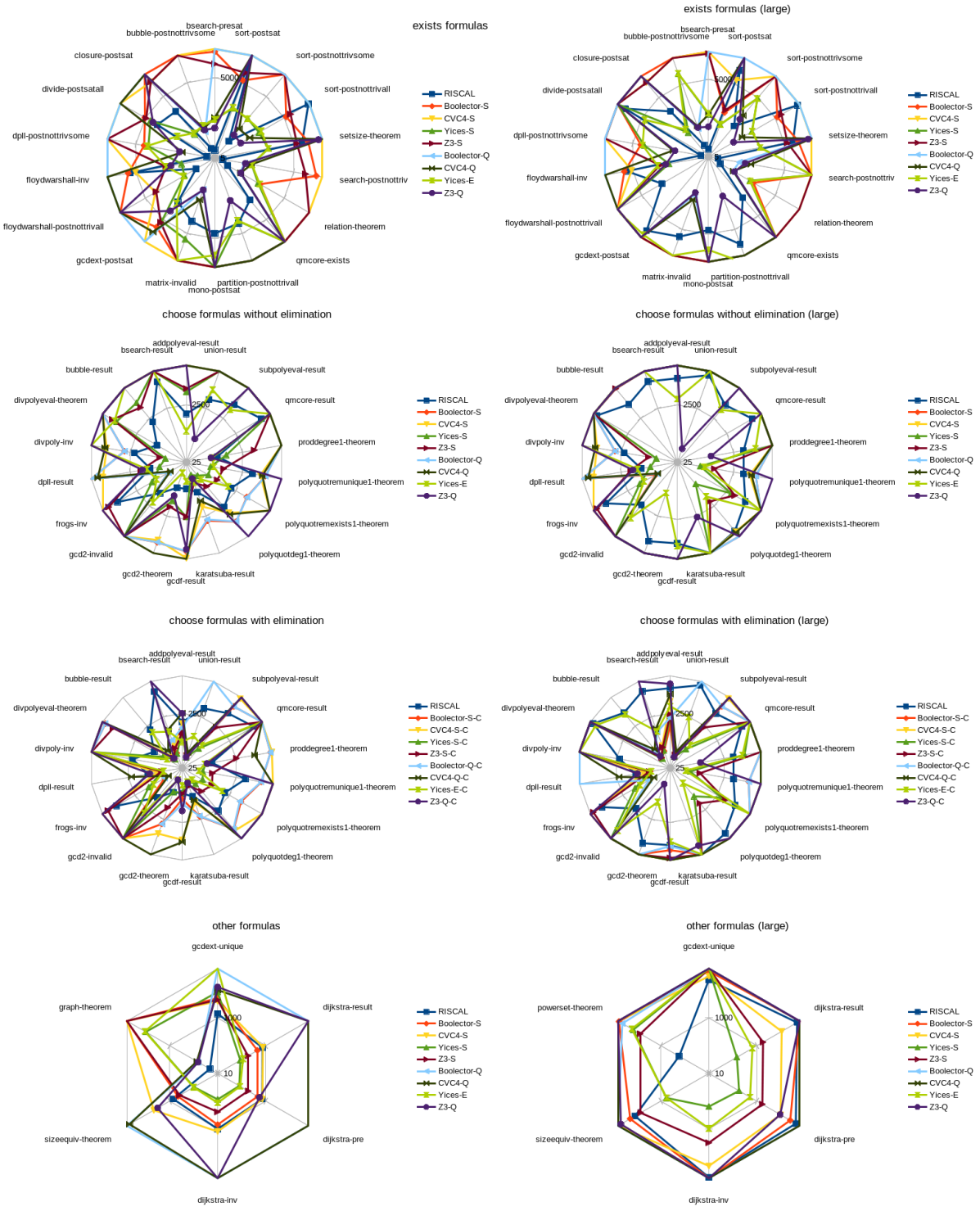


Figure 6: Real-Life Benchmarks (available from `https://www.risc.jku.at/research/formal/software/RISCAL/papers/EvalSMT2021-models.tgz`)

Failure Analysis of Hadoop Schedulers using an Integration of Model Checking and Simulation

Mbarka Soualhia
Concordia University,
Montréal, Canada
soualhia@ece.concordia.ca

Foutse Khomh
Polytechnique Montréal,
Montréal, Canada
foutse.khomh@polymtl.ca

Sofiène Tahar
Concordia University,
Montréal, Canada
tahar@ece.concordia.ca

Abstract: The Hadoop scheduler is a centerpiece of Hadoop, the leading processing framework for data-intensive applications in the cloud. Given the impact of failures on the performance of applications running on Hadoop, testing and verifying the performance of the Hadoop scheduler is critical. Existing approaches such as performance simulation and analytical modeling are inadequate because they are not able to ascertain a complete verification of a Hadoop scheduler. This is due to the wide range of constraints and aspects involved in Hadoop. In this paper, we propose a novel methodology that integrates and combines simulation and model checking techniques to perform a formal verification of Hadoop schedulers, focusing on the following properties: schedulability, fairness and resources-deadlock freeness. We use the CSP language to formally describe a Hadoop scheduler, and the PAT model checker to verify its properties. Next, we use the proposed formal model to analyze the scheduler of OpenCloud, a Hadoop-based cluster that simulates the Hadoop load, in order to illustrate the usability and benefits of our work. Results show that our proposed methodology can help identify several tasks failures (up to 78%) early on, i.e., before the tasks are executed on the cluster.

1 Introduction

Motivated by the reasonable prices and the good quality of cloud services, several enterprises and governments are deploying their applications in the cloud. Hadoop [1] has become enormously popular for processing data-intensive applications in the cloud. A core constituent of Hadoop is the scheduler. The Hadoop scheduler is responsible for the assignment of tasks belonging to applications across available computing resources. Scheduling data-intensive applications' tasks is a crucial problem, especially in the case of real-time systems, where several constraints should be satisfied. Indeed, an effective scheduler must assign tasks to meet applications' specified deadlines and to ensure their successful completions. In the cloud, failures are the norm rather than the exception and they often impact the performance of tasks running in Hadoop. Consequently, the Hadoop scheduler may fail to achieve its typical goal for different reasons such as resources-deadlock, task starvation, deadline non-satisfaction and data loss [8] [11]. This results in extra delays that can be added and propagated to the overall completion time of a task, which could lead to resources wastage, and hence significantly decreasing the performance of the applications and increasing failure rates. For instance, Dinu and Ng [7] analyzed the performance of the Hadoop framework under different types of failures. They reported that the Hadoop scheduler experiences several failures due to the lack of information sharing about its environment (e.g., scheduler constraints, resources availability), which can lead to poor scheduling decisions. For instance, they claim that the Hadoop scheduler experiences several failures because of prioritized executions of some long tasks before small ones, long delays of the struggling tasks or unexpected resources contentions [7] [14]. The widespread use of the Hadoop framework in safety and critical applications, such as healthcare [4] and

T. Kutsia (Ed.): Symbolic Computation
in Software Science (SCSS'21)
EPTCS 342, 2021, pp. 114–128, doi:10.4204/EPTCS.342.10

aeronautics [3], poses a real challenge to software engineers for the designing and testing of such framework to meet its typical goal and avoid poor scheduling decisions. Although several fault-tolerance mechanisms are integrated in Hadoop to overcome and recover from these failures, several failures still occur when scheduling and executing tasks [14]. Indeed, these failures can occur because of poor scheduling decisions (e.g., resources deadlock, task starvation) or constraints related to the environment where they are executed (e.g., data loss, network congestion). As such, verifying the design of Hadoop scheduler is an important and open challenge to identify the circumstances leading to task failures and performance degradation so that software engineers studying Hadoop can anticipate these potential issues and propose solutions to overcome them. This is to improve the performance of the Hadoop framework.

Different approaches have been adopted by the research community to verify and validate the behavior of Hadoop with respect to scheduling requirements. Traditionally, simulation and analytical modeling have been widely used for this purpose. However, with many Hadoop-nodes being deployed to accommodate the increasing number of demands, they are inadequate because they are not efficient in exploring large clusters. In addition, given the highly dynamic nature of Hadoop systems and the complexity of its scheduler, they cannot provide a clear understanding and exhaustive coverage of the Hadoop system especially when failures occur. Particularly, they are not able to ascertain a complete verification of the Hadoop scheduler because of the wide range of constraints and unpredictable aspects involved in the Hadoop model and its environment (e.g., diversity of running loads, availability of resources and dependencies between tasks). Formal methods have recently been used to model and verify Hadoop. However, to the best of our knowledge very few efforts have been invested in applying formal methods to verify the performance of Hadoop (e.g., data locality, read and write operations, correctness of running application on Hadoop) [15] [12] [9]. In addition, there is no work that addresses the formal verification of Hadoop schedulers. Considering the above facts, we believe that it is important to apply formal methods to verify the behavior of Hadoop schedulers. This is because formal methods are more able to model complex systems and thoroughly analyze their behavior than empirical testing. Our aim is to formally analyze the impact of the scheduling decisions on the failures rate and avoid simulation cost in terms of execution time and hardware cost (especially for large clusters). This allows to early identify circumstances leading to potential failures, in a shorter time compared to simulation, and prevent their occurrences. In contrast to simulation and analytical modeling, knowing these circumstances upfront would help practitioners better select the cluster settings (e.g., number of available resources, type of scheduler) and adjust the scheduler design (e.g., number of queues, priority handling strategies, failures recovery mechanisms) to prevent poor scheduling decisions in Hadoop.

In this paper, we present a novel methodology that combines simulation and model checking techniques to perform a formal analysis of Hadoop schedulers. Particularly, we study the feasibility of integrating model checking techniques and simulation to help in formally verifying some of the functional scheduling properties in Hadoop including schedulability, resources-deadlock freeness, and fairness [5]. To this aim, we first present a formal model to construct and analyze the three mentioned properties within the Hadoop scheduler. We use the Communicating Sequential Processes (CSP) language [16] to model the existing Hadoop schedulers (FIFO, Fair and Capacity schedulers [14]), and use the Process Analysis Toolkit (PAT) model checker [18] to verify the schedulers' properties. Indeed, the CSP language has been successfully used to model the behavior of synchronous and parallel components in different distributed systems [16]. Furthermore, PAT is a CSP-based model checker that has been widely used to simulate and verify concurrent, real-time systems, etc. [18]. Based on the generated verification results in PAT, we explore the relation between the scheduling decisions and the failures rate while simulating different load scenarios running on the Hadoop cluster. This is in order to propose possible scheduling strategies to reduce the number of failed tasks and improve the overall cluster performance

(resources utilization, total completion time, etc). In order to illustrate the usability and benefits of our work to identify failures that could have been prevented using our formal analysis methodology, we apply our approach on the scheduler of OpenCloud, a Hadoop-based cluster that simulates the real Hadoop load [10]. Using our proposed methodology, developers would identify up to 78% of tasks failures early on before the deployment of the application. Based on the performed failures analysis, our methodology could provide potential strategies to overcome these failures. For instance, our solution could propose new scheduling strategies to adjust the Hadoop cluster settings (e.g., size of the queue, allocated resources to the queues in the scheduler, etc.) and reduce the failures rate when compared to the real-execution simulation results. To the best of our knowledge, the present work is different from existing research in applying and integrating both formal methods and simulation for the analysis of Hadoop schedulers and formally analyzing the impact of the scheduling decisions on the failures rate in Hadoop. Furthermore, our proposed methodology to integrate model checking and simulation to verify Hadoop schedulers could be also applied to formally analyze other schedulers, than Hadoop, such as Spark [2], which has become one of the key cluster-computing framework that can be running on Hadoop.

The rest of the paper is organized as follows: Section 2 describes basics of Hadoop architecture, CSP language, and the tool PAT. Section 3 presents the proposed methodology for the formal analysis of the Hadoop scheduler. We describe the analysis of the scheduler of the OpenCloud a Hadoop cluster in Section 4. Section 5 summarizes the most relevant related work. Finally, Section 6 concludes the paper and highlights some future directions.

2 Preliminaries

In this section we briefly present the Hadoop architecture and some of the basic of the CSP language and the tool PAT, which will be used in the rest of the paper. This is in order to better understand the different steps of our proposed methodology.

2.1 Hadoop Architecture

Hadoop [14] is an open source implementation of the MapReduce programming model [8]. MapReduce is designed to perform parallel processing of large datasets using a large number of computers. A MapReduce job is comprised of two functions: a map and reduce, and the input data [14]. Hadoop has become the *de facto* standard for processing large data in today's cloud environment. It is a master-slave-based framework, the master node consists of a JobTracker and NameNode. The worker/slave node consists of a TaskTracker and DataNode. The Hadoop Distributed File System (HDFS) is the storage unit responsible for controlling access to data in Hadoop. Hadoop is equipped with a centerpiece; the scheduler which distributes tasks across worker nodes according to their availability. The default scheduler in Hadoop is based on the First In First Out (FIFO) principle. Facebook and Yahoo! have developed two new schedulers for Hadoop: Fair scheduler and Capacity scheduler, respectively [14].

2.2 CSP and PAT

CSP is a formal language used to model and analyze the behavior of processes in concurrent systems. It has been practically applied in modeling several real time systems and protocols [16]. In the sequel, we present a subset of the CSP language, which will be used in this work, where P and Q are processes, a is an event, c is a channel, and e and x are values:

`P , Q ::= Stop` | `Skip` | `a` $\rightarrow$ `P` | `P ; Q` | `P` $\|\|$ `Q` | `c!e` $\rightarrow$ `P` | `c?x` $\rightarrow$ `P`

- `Stop`: indicates that a process is in the state of deadlock.
- `Skip`: indicates a successfully terminated process.
- `a` $\rightarrow$ `P`: means that an object first engages in the event a and then behaves exactly as described by P.
- `P ; Q`: denotes that P and Q are sequentially executed.
- `P` $||$ `Q`: denotes that P and Q are processed in parallel. The two processes are synchronized with the same communication events.
- `c!e` $\rightarrow$ `P`: indicates that a value e was sent through a channel c and then a process P.
- `c?x` $\rightarrow$ `P`: indicates a value was received through a channel c and stored in a variable x and then a process P.

PAT [18] is a CSP-based tool used to simulate and verify concurrent, real-time systems, etc. [17]. It implements different model checking techniques for system analysis and properties verification in distributed systems (e.g., deadlock-freeness, reachability, etc.). Different advanced optimizations techniques, such as partial order reduction, symmetry reduction, etc., are available in PAT to reduce the number of explored states and CPU time.

3 Formal Analysis Methodology

In this section, we first present a general overview of our methodology followed by a description of each step.

3.1 Methodology Overview

Figure 1 provides an overview of the main idea behind our formal analysis of the Hadoop scheduler when integrating model checking and simulation, to early identify failure occurrences. The inputs of our proposed methodology are (1) the description of the Hadoop scheduler, (2) the specification of the properties to be verified, and (3) the scheduling metrics (e.g., type of scheduler, number of nodes, workload and failure distributions, schedulability rate). Our proposed methodology is comprised of three main steps, including (a) the formal modeling of the Hadoop scheduler and its properties in CSP and LTL, (b) the quantitative analysis of failures using model checking in PAT, and (c) the qualitative analysis of the failures using simulation of the proposed scheduling strategies and real-simulation traces. The outputs of our methodology are the rate of failures of the verified scheduler, and a set of possible scheduling strategies to overcome these failures.

We have chosen the CSP language to formally describe the scheduler as it allows to model the behavior of processes in concurrent systems. It has been successfully applied in modeling synchronous and parallel components for several real-time and distributed systems [16] (which is the case of Hadoop). The properties we aim to verify are written in Linear Temporal Logic (LTL). Thereafter, we use the PAT model checker to perform the formal quantitative analysis of failures in Hadoop scheduler. PAT is based on CSP and implements various model checking techniques to analyze and simulate the behavior of several distributed systems (e.g., transportation management system, Web authentication protocols, wireless sensor networks, etc.) [18]. Furthermore, it allows to model timed and probabilistic processes in the studied systems [17]. Based on the generated results from PAT, we perform a qualitative failures analysis to determine the circumstances and specifications leading to tasks' failures in the scheduler. The remainder of this section elaborates more on each of the steps of our methodology.

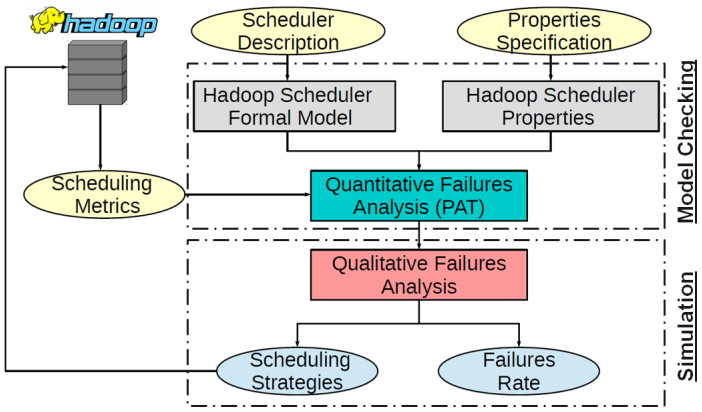


Figure 1: Overview of the Formal Analysis of Hadoop Schedulers Methodology

3.2 Hadoop Scheduler Formal Model: Model Checking

The first step in conducting the proposed formal analysis of the Hadoop scheduler using a model checker is to construct a formal model of the main components in Hadoop responsible for the scheduling of tasks, using the CSP language. To do so, we start by writing a formal description of the Hadoop master node: the JobTracker and NameNode. At the master level, we model the scheduler and the main entities responsible for task assignment and resources allocation in Hadoop. Next, we model the TaskTracker and the DataNode including the entity responsible for the task execution at the worker nodes. In addition, we integrate some of the important scheduling constraints in Hadoop in the model of the TaskTracker nodes including the data locality, data placement, and speculative execution of tasks [14]. Here, we selected these three constraints because of their direct impact on the scheduling strategies and the performance of executed tasks [14]. At this level, we should mention that the provided model of the Hadoop scheduler represents a close representation of the actual ones (e.g., FIFO, Fair, and Capacity) because it includes the main functionalities responsible for assigning the received tasks on available nodes. Furthermore, we checked the correspondence between the provided model and the real Hadoop scheduler by comparing the scheduling outcomes of scheduled tasks when using the formal model and the existing ones. More details about this comparison is given in Section 4.2.2. The obtained results showed a good matching between the two models. Nevertheless, further functionalities can be added to the presented model in order to improve its results (*i.e.,* recovery mechanisms, efficient resources assignment). In the following, we present a formal description of the steps to model a Hadoop cluster. Then, we present examples of implemented CSP processes[1] to describe the *Hadoop scheduler*, *TaskTracker activation* and *task assignment*, where "*Cluster()*" is the main process and *N* is the number of available TaskTracker nodes in the cluster:

```
Cluster() = initialize(); NameNode_activate() ||| JobTracker_activate() |||
            (|| i:{0..(N-1)}@DataNode_activate(i)) |||
            (|| i:{0..(N-1)}@TaskTracker_activate(i))||
            Hadoop_Scheduler();
```

The following process presents a formal description of the steps in *"Hadoop_Scheduler()"* to check scheduling constraints of map/reduce tasks. First, it checks the availability of resource slots (*slotTT[i]>0*). Then, it checks the type of task to be scheduled, either a map (*Queue[index] == 1*) or reduce (*Queue[index] == 2*) task. It also checks whether a task is speculatively executed or not (e.g., map:

[1]The entire CSP script is available at:
http://hvg.ece.concordia.ca/projects/cloud/fvhs.html

Queue[index] == 3 or reduce: *Queue[index] == 4*). Next, it assigns the received task to the TaskTracker node where it will be executed (*signedtask?i* $\rightarrow$ *signedtask_i* $\rightarrow$ *Task_Assignment(location,type);*).

```
Hadoop_Scheduler() =
{  if( (slotTT[i]>0) )
   {while( (found==0) && (index < maxqueue ) )
    { if (Queue[index] == 1)  //it is a map task
      {  schedulable = 1; found =1; location = index;
         type = MapTask; IDjob_ task = IDJob[index];}
      if((Queue[index] == 2))  //it is a reduce task
      { if(FinishedMap[IDJob[index]] == Map[IDJob[index]]  )
       { schedulable = 1; found =1; location = index;
         type = ReduceTask; IDjob_ task = IDJob[index];}    }
      if(Queue[index] == 3)  //it is a speculated map task
       { schedulable = 1; found =1; location = index; type = MapTask;
         IDjob_ task = IDJob[index]; SpeculateTask[location] = 1;}
      if(Queue[index] == 4)  //it is a speculated reduce task
      { schedulable = 1; found =1; location = index; type = ReduceTask;
         IDjob_ task = IDJob[index]; SpeculateTask[location] = 1;}
        ...
    }
   }
  } → signedtask?i→ signedtask_i→Task_Assignment(location, type);
```

"TaskTracker_activate(i)" presents our proposed process to activate the TaskTracker, after checking that the JobTracker was already activated (*JobTracker == ON*) and this TaskTracker was not already activated (*TaskTracker[i] == OFF*). Next, it activates this Tasktracker (*TaskTracker[i] == ON*) and the number of slots specified to this TaskTracker (*slotsnb*). These activated slots are ready to execute tasks.

```
TaskTracker_activate(i) = activate_jt_success  → ifa(TaskTracker[i] ==
                      OFF && JobTracker == ON) {activate_tt.i{
                      TaskTracker[i] = ON; trackercount++;}
                       → atomic{activate_tt_success.i →
                      (|| j:{1..(slotsnb)}@TaskTracker_sendready(i))}};
```

The following *"TaskTracker_execute(i)"* process presents an example of executing a task after checking its locality in Hadoop. For instance, it checks the availability of the slot assigned to a given task by the scheduler (if slot is free then *task_running[nbTT][k]* is equal to 0, where *nbTT* is the ID of the TaskTracker and *k* is the ID of the assigned slot). Next, it checks the locality of the task by checking whether the node where to execute the task (*selectedTT*) is the same node where its data is located (*DataLocalTT[idtask]*).

```
  TaskTracker_execute(i) =
{ var nbTT =i; var found = 0; var k = 0;
 while((k<slotsnb) && (found == 0) ){
  if(task_ at_ tasktracker[nbTT][k]==1 && task_running[nbTT][k]== 0)
   {selectedslot =k; found = 1; }
```

```
  k++; } ...
  if(Data-LocalTT[idtask] == selectedTT) //check locality of the task
   {locality = locality + 1;  Locality[idtask] = 1;}
    else {nonlocality = nonlocality + 1;  Locality[idtask] = 0;   }
 ...}
} → if(pos== -1) {TaskTracker_ execute(i)}
          else {execute(i,selectedslot)};
```

3.3 Hadoop Scheduler Properties: Model Checking

The three selected properties we aim to verify in our work are the schedulability, fairness and resources-deadlock freeness. We select these properties because they represent some of the main critical properties affecting the execution of tasks in real-time systems (e.g., task outcome, delays, resources utilization) [5]. The three properties can be described as follows: the *schedulability* checks whether a task is scheduled and satisfies its specified deadline when scheduled according to a scheduling algorithm. The *fairness* checks whether the submitted tasks are served (e.g., receiving resources slots that ensure their processing) and there are no tasks that will never be served or will wait in the queue more than expected. The *resources-deadlock* checks whether two tasks are competing for the same resource or each is waiting for the other to be finished.

To better illustrate the properties to be verified, we explain as an example the schedulability property and its corresponding states in our approach. For the schedulability, a task can go from state: *submitted* to *scheduled* to *processed* then to *finished-within-deadline* or *finished-after-deadline* or *failed*. Let X be the total number of scheduled tasks and Y be the number of tasks finished within their expected deadlines. The schedulability rate can be defined as the ratio of X over Y. The property *"schedulabilityrate* > 80*"* means checking whether the scheduler can have a total of 80% of tasks finished within their deadlines.

To verify above properties in PAT, we need to provide their descriptions in LTL. For example, the following LTL formulas check the schedulability and resource-deadlock freeness of given tasks. The first example checks whether a given task eventually goes from the state submitted to the state finished within the deadline. The second example checks whether a given task should not go to a state of waiting-resources. Here, $\Diamond$, $\models$, ->, and $\neg$ represent *eventually*, *satisfy*, *imply*, and *not*, respectively, in the LTL logic.

```
# assert ♢ (task ⊨ (submitted -> finished-within-deadline) );
# assert ¬ (task ⊨ (submitted -> waiting-resources) );
```

3.4 Quantitative Failures Analysis using Model Checking

Provided the CSP model of the Hadoop scheduler and the properties to be verified, we perform the formal analysis of the scheduler performance using the PAT model checker while simulating different Hadoop workload scenarios. Here, we can vary the property requirements to assess the performance of the scheduler under different rates and evaluate their impact on the failures rates of the cluster and the simulated scenarios. For example, we can define *"goal0"* to check whether all the submitted tasks (*"workload"*) are successfully scheduled. Using PAT, we can verify if the modeled cluster, *"cluster1"*, can reach this goal or not. Another example could be to check if *"cluster1"* meets *"goal0"* with a *"schedulabilityrate"* of 80%. The following examples present some of the properties that can be verified using our approach.

```
#define goal0 completedscheduled  == workload && workload >0;
#assert cluster1 reaches goal0;
#assert cluster1 reaches goal0  && schedulabilityrate >80;
#define goal1 fairnessrate ==50;
#assert cluster1 reaches goal0  && goal1;
#define goal2 resourcedeadlockrate ==50;
#assert cluster1 reaches goal0  && goal1  && goal2;
```

3.5 Qualitative Failures Analysis using Simulation

The last step in our proposed methodology is to use the traces simulated and generated by the PAT model checker to extract information about the applied scheduling strategies and explore their impact on tasks' failures using simulation. For instance, we can investigate the states where the scheduler does (not) meet a given property and map these states to the obtained scheduler performance and to the input cluster settings. This step allows us to find a possible correlation between the cluster settings, the applied scheduling strategies, and the failures rate. Next, we use these correlations to suggest scheduling strategies to overcome these failures by either (1) recommending to the Hadoop developers to change the scheduling decisions (e.g., delay long tasks, wait for a local task execution) or (2) to customers to change and adjust their cluster settings (e.g., number of nodes, number of allowed speculative executions). In next section, we propose a case study to evaluate and simulate the suggested scheduling strategies on a Hadoop environment and measure their impact on the failure rates and the Hadoop cluster performance. Generally, our solution could propose new scheduling strategies to adjust the Hadoop cluster settings and hence reduce failures rate when compared to the real-execution simulation results.

4 Case Study: Formal Analysis of OpenCloud Scheduler

In this section, we illustrate the practical usability and benefits of our work by formally analyzing the scheduler of OpenCloud, a Hadoop-based cluster [10] that simulates the Hadoop load.

4.1 Case Study Description

To apply our formal approach on a case study, we investigated existing Hadoop case studies in the open literature. Overall, we found three main public case studies including: Google [19], Facebook [20], and OpenCloud [10] traces. We found out that the Google traces do not provide data about the cluster settings, which is an important factor affecting the analysis results in our approach. For the Facebook traces, we noticed that they do not provide data about the cluster settings, capacity of nodes, failures rate, etc., which are essential for our approach. Whereas, we found that the OpenCloud traces provide the required inputs of our verification approach (e.g., # nodes, capacity of nodes, workload). Therefore, we choose to formally analyze the scheduler of this cluster because it provides public traces of real-executed Hadoop workload for more than 20 months. OpenCloud [10] is an open research cluster managed by Carnegie Mellon University. It is used by several groups in different areas such as machine learning, astrophysics, biology, cloud computing, etc.

Based on the modeled system of the OpenCloud's scheduler and the specified properties, we start the verification by parsing the trace files to extract the input information needed in our methodology. Specifically, we use the description of the workload included in the first month traces, and the first six months traces together. This allows us to evaluate the scalability of our methodology, in terms of number of visited states and execution time, using different traces. Although these traces provide the required inputs for our methodology, we did not find any information describing the type of scheduler

used in the cluster. Since the type of the scheduler is an important factor that impacts the performance of the cluster, we evaluate the performance of the modeled cluster for the three existing schedulers of Hadoop (FIFO, Fair and Capacity). We vary the property requirements to assess the performance of the used scheduler under different rates and evaluate their impact on the failures rate in the cluster while executing the same Hadoop workload. For the search engines in PAT, we use the "First Witness Trace using Depth First Search" in order to perform an analysis without reduction, and the "First Witness Trace with Zone Abstraction" for the analysis with symmetry reduction. The testbed is a workstation with Intel i7-6700HQ (2.60GHz*8) CPU and 16 GB of RAM.

4.2 Verification and Evaluation

In this section, we present the obtained scalability results of our methodology along with the results of the formal quantitative and qualitative analyses of failures in Hadoop schedulers.

4.2.1 Properties Verification and Scalability Analysis

Tables 1 and 2 summarize the verification results of the first month trace, and the first six months traces together, respectively. The first trace provides information about 1,772,144 scheduled tasks, whereas the six files describing the executed workload for six months contain information about 4,006,512 scheduled tasks. In the sequel, we discuss the results of the performed analysis.

The results presented in Table 1 show that only the Fair scheduler satisfies the schedulability property (for the two given rates), meaning that up to 80% are scheduled and executed within their expected deadlines. The Capacity scheduler does not meet the schedulability rate of 80%. However, the FIFO does not satisfy the schedulability properties for the two input rates, meaning that more than 50% of scheduled tasks are exceeding their expected deadlines. Hence, these tasks are using their assigned resources more than expected, which can affect the overall performance of the cluster.

For the fairness property, only the Fair scheduler satisfies the property of fairness with a rate of 50% and 80%, meaning that at most 80% of tasks get served and executed on time. However, both the FIFO and the Capacity schedulers violate the fairness property for the two given values. Therefore, we can claim that more than 80% of the submitted tasks are waiting longer than expected in the queue before getting executed. This may lead to a problem of task starvation.

For the resources-deadlock, we can report that the Capacity scheduler is characterized by more tasks that experience a resources-deadlock compared to the FIFO and the Fair schedulers. This is because it satisfies the property of resources-deadlock for the two given rates (e.g., 10% and 30%). This can be explained by the fact that the Capacity scheduler suffers from the problem of miscalculation of resources (headroom calculation). The FIFO scheduler shows that only 10% of the scheduled tasks experience the problem of resources-deadlock. The Fair scheduler does not satisfy the resources-deadlock property for two given rates (10% and 30%) for the first trace, which means that less than 10% of tasks may experience an issue of resources-deadlock. Hence, we can conclude that the Fair scheduler generates less scheduling decisions leading to resources-deadlock situations for the OpenCloud cluster.

Overall, our proposed methodology is able to verify the given three properties for the three existing Hadoop schedulers, while exploring on average 11086 K states in 3724 seconds without symmetry reduction, and 742 K states in 1619 seconds with symmetry reduction, as shown in Table 1.

In order to evaluate the scalability of our methodology, we perform the formal analysis of a cumulative workload. This is done by incrementally adding the Hadoop workload of each month to the previous trace(s) to be analyzed. We start the evaluation by analyzing the trace of the first month, and add the traces up to the trace where the tool cannot perform the analysis. This is in order to check the bounds of the analysis performed in terms of explored states. The obtained results, presented in Table 2, show

Table 1: Verification Results: Trace for the First Month (1,772,144 Tasks)

Property	Scheduler	Results without SR			Results with SR		
		Valid?	#States	Time(s)	Valid?	#States	Time(s)
Schedulability = 50%	FIFO	No	11086 K	3869	No	742 K	1648
	Fair	Yes	11086 K	3524	Yes	742 K	1597
	Capacity	Yes	11086 K	3601	Yes	742 K	1604
Schedulability = 80%	FIFO	No	11086 K	3789	No	742 K	1650
	Fair	Yes	11086 K	3676	Yes	742 K	1614
	Capacity	No	11086 K	3721	No	742 K	1602
Fairness = 50%	FIFO	No	11086 K	3714	No	742 K	1594
	Fair	Yes	11086 K	3759	Yes	742 K	1615
	Capacity	No	11086 K	3736	No	742 K	1612
Fairness = 80%	FIFO	No	11086 K	3855	No	742 K	1675
	Fair	Yes	11086 K	3795	Yes	742 K	1642
	Capacity	No	11086 K	3761	No	742 K	1619
Resources-Deadlock = 10%	FIFO	Yes	11086 K	3615	Yes	742 K	1602
	Fair	No	11086 K	3698	No	742 K	1610
	Capacity	Yes	11086 K	3704	Yes	742 K	1632
Resources-Deadlock = 30%	FIFO	No	11086 K	3742	No	742 K	1596
	Fair	No	11086 K	3733	No	742 K	1618
	Capacity	Yes	11086 K	3752	Yes	742 K	1623

SR = Symmetry Reduction

that our approach could formally analyze the first six traces (combined) describing the workload (about 4,006,512 tasks) for the first six months. It could analyze the first five traces together with and without symmetry reduction, however, it could analyze the six traces together only with symmetry reduction by exploring on average 17,692 K states in 4346 seconds.

Table 2: Verification Results: Trace for the 1-6 Months (4,006,512 Tasks)

Property	Scheduler	Results without SR			Results with SR		
		Valid?	#States	Time(s)	Valid?	#States	Time(s)
Schedulability = 50%	FIFO	**	**	**	Yes	17692K	4350
	Fair	**	**	**	Yes	17692K	4362
	Capacity	**	**	**	Yes	17692K	4359
Schedulability = 90%	FIFO	**	**	**	No	17692K	4346
	Fair	**	**	**	No	17692K	4341
	Capacity	**	**	**	No	17692K	4367
Fairness = 30%	FIFO	**	**	**	Yes	17692K	4377
	Fair	**	**	**	Yes	17692K	4312
	Capacity	**	**	**	Yes	17692K	4335
Fairness = 90%	FIFO	**	**	**	No	17692K	4352
	Fair	**	**	**	No	17692K	4328
	Capacity	**	**	**	No	17692K	4369
Resources-Deadlock = 10%	FIFO	**	**	**	Yes	17692K	4322
	Fair	**	**	**	No	17692K	4360
	Capacity	**	**	**	Yes	17692K	4354
Resources-Deadlock = 50%	FIFO	**	**	**	No	17692K	4338
	Fair	**	**	**	No	17692K	4342
	Capacity	**	**	**	No	17692K	4328

SR = Symmetry Reduction, ** = Not Available

4.2.2 Quantitative Failures Analysis

We use the traces generated by the PAT model checker during the verification of the three properties described in Section 4.2.1, to explore states where the scheduler did not satisfy a given property value. This step is important to map these states to the failed tasks, if found, and examine the relationship between

the verified property and the task failure. We apply this step on the first trace file because it contains an important number of scheduled tasks (i.e., 1,772,144 tasks). To do so, we first identify the tasks that did not satisfy a given property, for example schedulability = 80% or resources-deadlock = 10%. Then, we check whether these tasks were failed when they were executed in the real cluster. Next, we classify the observations into four main categories: *True Positive (TP)*, *True Negative (TN)*, *False Positive (FP)*, and *False Negative (FN)*. *TP* represents the successfully executed tasks, based on the simulation traces, that are identified as finished tasks using our approach. Likewise, *TN* represents the failed tasks, based on the simulation traces, that are identified as failed tasks. While *FP* denotes the amount of identified finished tasks that failed during the real simulation. Finally, *FN* denotes the number of identified failed tasks that are finished in reality. These four metrics are calculated by comparing the status of the tasks in the generated traces using our methodology, specifically from PAT, and the simulation traces across the total number of scheduled tasks in the input trace (1,772,144 tasks).

Overall, we observe that our formal analysis approach could identify up to 61.49% of the finished tasks (*TP*, Fair, schedulability = 50%), and up to 4.62% of the failed tasks (*TN*, Fair, schedulability = 80%) without symmetry reduction. Here, we notice that the *TN* values are small compared to the *TP* ones. This can be explained by the fact that the first trace contains more than 94% of successful tasks. For this reason, we computed another metric that we call the *Detected Failures (DF)*. The *DF* is calculated by mapping the *TN* rate over the total number of failed tasks in the input trace. Our aim here is to quantify the amount of detected failures using our formal verification approach when compared to the given OpenCloud traces (obtained from real-execution simulation). This is to answer the question of how many failures could be identified by our formal verification approach before the application is deployed in the field?

At this level, we can claim that our approach is able to catch between 42% and 78.57% of the total failed tasks without symmetry reduction. While it can catch between 42% and 78.91% of the failures when using the symmetry reduction. On the other hand, we notice that the *FN* is in the order of 40%, which means that more than 40% of tasks are finished, in the simulation traces, but our methodology indicates their failures. This can be explained by the internal mechanisms implemented in Hadoop internally to recover these tasks in case of a failure. For example, the mechanism of pausing and resuming running tasks allows higher priority tasks to be executed without killing the lower priority ones [11]. Furthermore, we notice that the false positive rate varies from 1.26% and 3.41% and indicates failed tasks that are identified as finished using our methodology. A possible explanation for these false positive results can be the lack of some real-world constraints that affect the execution of tasks (e.g., network congestion, lack of resources). In addition, failed tasks identified as finished can generate more false positive results due to the tight dependency between the map and the reduce tasks (the reduce tasks will be launched when all the map tasks are successfully executed).

Next, we check the states of the failed tasks and analyze the factors that may cause the failure of these tasks. We find that the scheduler of OpenCloud experiences several failures, up to 32%, because of long delays that exceed the maximum timeout for a task to be finished (property "mapred.task.timeout": defining the maximum timeout for a task to be finished). We carefully checked the sates of these tasks and found that these delays are mainly caused by data locality constraints [14] and that they can reach 10 minutes (for small and medium tasks). Moreover, we found out that about 40% of these straggling tasks cause the failure of the job to which they belong, resulting in a waste of resources. Moreover, we noticed that many failures are cascaded from one job to another, especially in the long Hadoop chains. A Hadoop chain is a connection between more than one Hadoop job to execute a specific workload. This may result in a degradation in performance and many failures. We can claim that the Hadoop scheduler lacks mechanisms to share information about these failures between its components to avoid their occurrence.

Table 3: Coverage Results(%): Trace for the First Month (1,772,144 Tasks)

Property	Scheduler	Results without SR					Results with SR				
		TP	TN	FP	FN	DF	TP	TN	FP	FN	DF
Schedulability = 50%	**FIFO**	56.03	3.2	2.68	38.09	54.76	47.29	2.47	3.41	46.83	42.00
	Fair	61.49	4.53	1.35	32.63	77.04	55.38	3.82	2.06	38.74	64.96
	Capacity	49.16	2.47	3.41	44.96	42.00	46.09	2.85	3.03	48.03	48.46
Schedulability = 80%	**FIFO**	50.14	3.46	2.42	43.98	58.84	47.98	2.79	3.09	46.14	47.44
	Fair	59.83	4.62	1.26	34.29	78.57	56.82	4.21	1.67	37.3	71.59
	Capacity	43.61	3.05	2.83	37.73	51.87	42.18	3.03	2.85	51.94	51.53
Fairness = 50%	**FIFO**	49.28	2.92	2.96	44.84	49.65	47.84	2.92	2.96	46.28	49.65
	Fair	55.36	3.89	1.99	38.76	66.15	49.63	3.86	2.02	44.49	65.64
	Capacity	47.03	2.47	3.41	47.09	42.00	44.11	3.04	2.84	50.01	51.70
Fairness = 80%	**FIFO**	44.88	3.76	2.12	49.29	63.94	49.77	3.32	2.56	44.35	56.46
	Fair	57.21	4.07	1.81	36.91	69.21	50.14	4.64	1.24	43.98	78.91
	Capacity	43.14	3.01	2.87	50.98	51.19	48.29	3.48	2.4	45.83	76.18
Resources-Deadlock = 10%	**FIFO**	43.73	3.06	2.82	50.39	52.04	46.21	3.19	2.69	47.92	54.25
	Fair	49.99	3.82	2.06	44.13	64.96	51.44	3.63	2.25	42.68	61.73
	Capacity	46.07	3.19	2.69	48.05	54.25	43.77	2.84	3.04	50.35	48.29
Resources-Deadlock = 30%	**FIFO**	50.22	3.53	2.35	43.9	60.03	48.54	2.91	2.97	45.58	49.48
	Fair	52.39	4.17	1.71	41.73	70.91	55.26	3.75	2.13	38.86	63.77
	Capacity	49.02	3.16	2.72	45.1	53.74	49.71	3.19	2.69	44.41	54.25

SR = Symmetry Reduction, DF = Detected Failures
TP = True Positive, TN = True Negative, FP = False Positive, FN = False Negative

On the other hand, we found out that 26% of tasks failed because they exceeded the maximum number of allowed speculative execution (e.g., property "mapred.map.tasks.speculative.execution": defining the maximum number of allowed speculative execution, they have likely more chances to fail). When checking the tasks waiting for a long time in the queue before being served, we observed that this is due to the fact that the long tasks are scheduled first and they occupy their assigned resources for a long time (e.g., a large input file to be processed, a job composed of more than 1000 tasks). Consequently, we can conclude that knowing these factors and issues, one can adapt the scheduler system of the Hadoop framework to overcome these failures and improve its performance.

4.2.3 Qualitative Failures Analysis

To show the benefits of our formal analysis approach, we propose to integrate and simulate some guidelines or strategies to adapt the scheduling decisions of the created Hadoop cluster and evaluate their impact on the failures rate. This is based on the generated traces and the performed failures analysis to check whether our work can help early identify the occurrence of failures and then propose appropriate mechanisms to overcome them. For instance, one possible strategy could be to change the cluster settings by adding more resources on its nodes or adding the number of nodes on it. This could solve for example the fairness and resources-deadlock issues. Another scheduling strategy could be to change the value of timeout of scheduled task, which represents the number of milliseconds before terminating a task. Another strategy could be to adjust the type of the scheduler used in the cluster (e.g., FIFO, Fair, Capacity, etc.). In this paper, we evaluate the impact of the strategy to change the cluster settings by adding more resources on the OpenCloud cluster where we change the number of nodes from 64 to 100 and simulate the same Hadoop workload. Figure 2 gives an overview of the impact of adding more resources in the cluster for the three schedulers considering a failures' rate of 5.88%; the identified failure rate from the first trace file. Overall, we noticed that adding more nodes in the cluster could reduce the failure rates by up to 2.34% (Fair scheduler), which means a reduction rate of 39.79%. This was expected since when we analyzed the traces we found out that several tasks are straggling for more than 10 minutes, waiting for other resources to be released.

Given the obtained findings, we can claim that our solution could identify new scheduling strategies to adjust the Hadoop cluster to reduce failures rates by combining simulation and model checking techniques and formally verifying the functional scheduling properties.

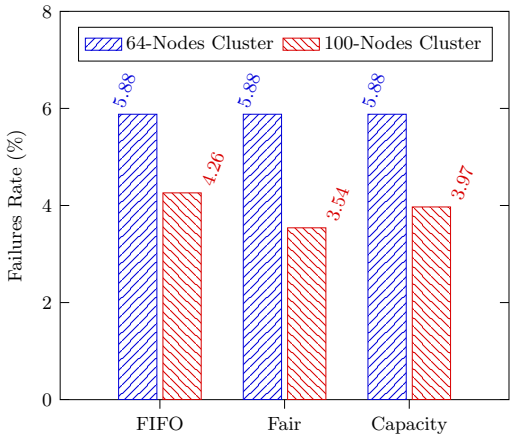


Figure 2: Impact of Resources Adding on Failures Rate

5 Related Work

In this section, we discuss the most relevant work that apply formal methods in the context of Hadoop framework. In [6] and [13], the authors analyzed the behavior of MapReduce using Stochastic Petri Nets and Timed Colored Petri Nets, respectively. They modeled the mean delay in each time transition of the scheduled tasks as formulas, and the Hadoop jobs were simulated based on the used Petri Nets. The proposed approaches could evaluate the correctness of the system and analyze the performance of MapReduce. But, they lack several constraints about the scheduling of the jobs and cannot cover larger Hadoop clusters. Su *et al.* [15] used the CSP language to formalize four key components in MapReduce. Specifically, they formally modeled the master, mapper, reducer and file system while considering the basic operations in Hadoop (e.g., task state storing, error handling, progress controlling). However, none of the properties of the Hadoop framework is verified using the formalized components. Xie *et al.* [22] address the formal verification of the HDFS reading and writing operations using CSP and the PAT model checker. For instance, they formally modeled the reading and writing operations for the HDFS based on the formalized components proposed in [15]. Moreover, they verified some of the HDFS properties including the deadlock-freeness, minimal distance scheme, mutual exclusion, write-once scheme and robustness. While this approach allows to detect unexpected traces generating errors and verify data consistency in the HDFS, a limitation of this work is that it only models the reading and writing operations for just one file system and requires to investigate the validity of these operations for distributed files as in HDFS. In [12], Reddy *et al.* propose an approach to model Hadoop's system using the PAT model checker. They used CSP to model Hadoop software components including the "NameNode", "DataNode", task scheduler and cluster setup. They identified the benefits of some properties like data locality, deadlock-freeness and non-termination among others and proved the correctness of these properties. However, their proposed model is evaluated on a small workload, and none of the properties is verified to check the performance of the Hadoop scheduler. Kosuke *et al.* [9] used the proof assistant

Coq to write an abstract computation model of MapReduce. They modeled the mapper and reducer as Coq functions and proved that the implementation of the mapper and reducer satisfies the specification of some applications such as WordCount [21]. The authors present an abstracted model of MapReduce, where many details are not included such as task assignment or resources allocation. These issues can affect the performance of applications running on Hadoop.

6 Conclusion and Future Work

Given the dynamic behavior of the cloud environment, the verification of the Hadoop scheduler has become an open challenge especially with many Hadoop-nodes being deployed to accommodate the increasing number of demands. Existing approaches such as performance simulation and analytical modeling are not able to ascertain a complete verification of the Hadoop scheduler because of the wide range of constraints involved in Hadoop. In this paper, we presented a novel methodology that combines and integrates simulation and model checking techniques to perform a formal analysis of Hadoop schedulers. Particularly, we studied the feasibility of integrating model checking techniques and simulation to help in formally verifying some of the functional scheduling properties. So, we formally verified some of the most important scheduling properties on Hadoop including the schedulability, resources-deadlock freeness, and fairness properties, and analyzed their impact on the failure rates. The ultimate goal of this work is to be able to propose possible scheduling strategies to reduce the number of failed tasks and improve the overall cluster performance and practitioner better assign and configure resources to mitigate the failures that a Hadoop scheduler may experience while simulating the Hadoop workload. We used CSP to model Hadoop schedulers, and the PAT model checker to verify the mentioned properties. To show the practical usefulness of our work, we applied our approach on the scheduler of OpenCloud, a real Hadoop-based cluster. The obtained results show that our approach is able to formally verify the selected properties and that such analysis can help identify up to 78% of failures before they occur in the field.

An important direction of future work is to verify other properties that can impact the failures rate in Hadoop, like the resource assignment, load balancing, and modeling the internal recovery mechanisms of Hadoop. Another direction can be the use of our work to automatically generate scheduling guidelines to improve the performance of the Hadoop framework and reduce the failures rate. Finally, the failure analysis approach and findings of this paper can be extended and evaluated on Spark [2], which has become one of the key cluster-computing framework that can be running on Hadoop. In fact, one can easily adapt the proposed methodology according to the architecture of Spark.

References

[1] Apache Hadoop: `http://hadoop.apache.org/`.

[2] Apache Spark: `http://spark.apache.org/`.

[3] Applying Apache Hadoop to NASA's Big Climate Data: `http://events.linuxfoundation.org/sites/events/files/slides/ApacheCon_NASA_Hadoop.pdf`.

[4] D. Chen, Y. Chen, B. N. Brownlow, P. P. Kanjamala, C. A. G. Arredondo, B. L. Radspinner & M. A. Raveling (2017): *Real-Time or Near Real-Time Persisting Daily Healthcare Data Into HDFS and ElasticSearch Index Inside a Big Data Platform*. *IEEE Transactions on Industrial Informatics* 13(2), pp. 595–606, doi:10.1109/TII.2016.2645606.

[5] A.M. K. Cheng (2002): *Real-Time Systems: Scheduling, Analysis, and Verification*. John Wiley & Sons, Inc., doi:10.1002/0471224626.

[6] S.-T. Cheng, H.-C. Wang, Y.-J. Chen & C.-F. Chen (2015): *Performance Analysis Using Petri Net Based MapReduce Model in Heterogeneous Clusters*. In: *Advances in Web-Based Learning*, *LNCS* 8390, Springer, pp. 170–179, doi:10.1007/978-3-662-46315-4_18.

[7] F. Dinu & T.S. E. Ng (2012): *Understanding the Effects and Implications of Compute Node Related Failures in Hadoop*. In: *International Symposium on High-Performance Parallel and Distributed Computing*, pp. 187–198, doi:10.1145/2287076.2287108.

[8] S. Li, S. Hu, S. Wang, L. Su, T. Abdelzaher, I. Gupta & R. Pace (2014): *WOHA: Deadline-Aware Map-Reduce Workflow Scheduling Framework over Hadoop Clusters*. In: *IEEE International Conference on Distributed Computing Systems*, pp. 93–103, doi:10.1109/ICDCS.2014.18.

[9] K. Ono, Y. Hirai, Y. Tanabe, N. Noda & M. Hagiya (2011): *Using Coq in Specification and Program Extraction of Hadoop Mapreduce Applications*. In: *International Conference on Software Engineering and Formal Methods*, pp. 350–365, doi:10.1007/978-3-642-24690-6_24.

[10] OpenCloud: `http://ftp.pdl.cmu.edu/pub/datasets/hla/dataset.html`.

[11] J. A. Quiané-Ruiz & *et al.* (2011): *RAFTing MapReduce: Fast Recovery on the RAFT*. In: *IEEE International Conference on Data Engineering*, pp. 589–600, doi:10.1109/ICDE.2011.5767877.

[12] G.S. Reddy, F. Yuzhang, L. Yang, S.D. Jin, J. Sun & R. Kanagasabai (2013): *Towards Formal Modeling and Verification of Cloud Architectures: A Case Study on Hadoop*. In: *International World Congress on Services*, pp. 306–311, doi:10.1109/SERVICES.2013.47.

[13] M. C. Ruiz, J. Calleja & D. Cazorla (2015): *Petri Nets Formalization of Map/Reduce Paradigm to Optimise the Performance-Cost Tradeoff*. In: *IEEE Trustcom/BigDataSE/ISPA*, 3, pp. 92–99, doi:10.1109/Trustcom.2015.617.

[14] M. Soualhia, F. Khomh & S. Tahar (2017): *Task Scheduling in Big Data Platforms: A Systematic Literature Review*. *Journal of Systems and Software* 134, pp. 170 – 189, doi:10.1016/j.jss.2017.09.001.

[15] W. Su, F. Yang, H. Zhu & Q. Li (2009): *Modeling MapReduce with CSP*. In: *IEEE International Symposium on Theoretical Aspects of Software Engineering*, pp. 301–302, doi:10.1109/TASE.2009.28.

[16] J. Sun, Y. Liu, J. S. Dong & C. Chen (2009): *Integrating Specification and Programs for System Modeling and Verification*. In: *IEEE International Symposium on Theoretical Aspects of Software Engineering*, pp. 127–135, doi:10.1109/TASE.2009.32.

[17] J. Sun, Y. Liu, J. S. Dong & J. Pang (2009): *PAT: Towards Flexible Verification under Fairness*. In: *Computer Aided Verification*, *LNCS* 5643, pp. 709–714, doi:10.1007/3-540-10843-2_22.

[18] Process Analysis Toolkit: `http://sav.sutd.edu.sg/PAT/`.

[19] Google Traces: `https://github.com/google/cluster-data`.

[20] Facebook Traces: `https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository`.

[21] WordCount Example: `http://wiki.apache.org/hadoop/WordCount`.

[22] W. Xie, H. Zhu, X. Wu, S. Xiang & J. Guo (2016): *Modeling and Verifying HDFS Using CSP*. In: *IEEE Annual Computer Software and Applications Conference*, 1, pp. 221–226, doi:10.1109/COMPSAC.2016.158.

E-CYCLIST: Implementation of an Efficient Validation of $\mathrm{FOL_{ID}}$ Cyclic Induction Reasoning

Sorin Stratulat

Université de Lorraine, CNRS, LORIA
Metz, F-57000, FRANCE

sorin.stratulat@univ-lorraine.fr

Checking the soundness of cyclic induction reasoning for first-order logic with inductive definitions ($\mathrm{FOL_{ID}}$) is decidable but the standard checking method is based on an exponential complement operation for Büchi automata. Recently, we introduced a polynomial checking method whose most expensive steps recall the comparisons done with multiset path orderings. We describe the implementation of our method in the CYCLIST prover. Referred to as E-CYCLIST, it successfully checked all the proofs included in the original distribution of CYCLIST. Heuristics have been devised to automatically define, from the analysis of the proof derivations, the trace-based ordering measures that guarantee the soundness property.

Introduction. Cyclic pre-proofs for the classical first-order logic with inductive predicates ($\mathrm{FOL_{ID}}$) have been extensively studied in [2, 3, 5]. They are finite sequent-based derivations where some terminal nodes, called *buds*, are labelled with sequents already occurring in the derivation, called *companions*. Bud-companion (BC) relations, graphically represented as *back-links*, are described by an *induction function* attached to the derivation, such that only one companion is assigned to each bud, but a node can be the companion of one or several buds. The pre-proofs can be viewed as digraphs whose cycles, if any, are introduced by the BC-relations.

It is easy to build unsound pre-proofs, for example by creating a BC-relation between the nodes labelled by the sequents from a stuttering step. The classical soundness criterion is the *global trace condition*. Firstly, the paths are annotated by traces built from inductive atoms occurring on the lhs of the sequents in the path, referred to as *inductive antecedent atoms* (IAAs). Then, it is shown that, for every infinite path p in the cyclic derivation of a false sequent, there is some trace following p such that all successive steps starting from some point are decreasing and certain steps occurring infinitely often are strictly decreasing w.r.t. some semantic ordering. We say that a *progress point* occurs in the trace when a step is strictly decreasing. A *proof* is a pre-proof if every infinite path has an infinitely progressing trace starting from some point.

The standard checking method [3] of the global trace condition is decidable but based on an exponential complement operation for Büchi automata [8]. It has been implemented in the CYCLIST prover [4] and experiments showed that the soundness checking can take up to 44% of the proof time. On the other hand, we presented in [9, 10] a less costly, polynomial-time, checking method. The pre-proof to be checked is firstly normalized into a digraph $\mathscr{P}$ consisting of a set of derivation trees to which is attached an extended induction function. The resulting digraph counts the companions among its roots, as well as the root of the pre-proof to be checked. Also, all infinite paths in the pre-proof, starting from some point, can be reconstructed by concatenating root-bud paths (*rb*-paths) in $\mathscr{P}$. Finally, a sufficient condition for ensuring the global trace condition is to show that every *rb*-path from the strongly connected components

T. Kutsia (Ed.): Symbolic Computation
in Software Science (SCSS'21)
EPTCS 342, 2021, pp. 129–135, doi:10.4204/EPTCS.342.11

(SCCs) of $\mathscr{P}$ has a trace that satisfies some trace-based ordering constraints. Therefore, in theory, if the soundness of some pre-proof can be validated with the new method, it can also be validated with the standard one.

Implementation. Our method has been integrated in the CYCLIST release labelled as CSL-LICS14, by *replacing* the standard checking method. The result was called E-CYCLIST. CYCLIST builds the pre-proofs using a depth-first search strategy that aims at closing open nodes as quickly as possible. Whenever a new cycle is built, model-checking techniques provided by an external model checker are called to validate it. If the validation result is negative, the prover backtracks by trying to find another way to build new cycles. Hence, the model checker may be called several times during the construction of a pre-proof.

Here is how our method works. Firstly, the pre-proof is normalized to a digraph $\mathscr{P}$. To each root r from $\mathscr{P}$, the method attaches a measure $\mathscr{M}(r)$ consisting of a multiset of IAAs of the sequent labelling r, denoted by $S(r)$. One of the challenges is to find the good measure values that satisfy the trace-based ordering constraints. A procedure for computing these values is given by Algorithm 1.

Algorithm 1 GenOrd($\mathscr{P}$): to each root r of $\mathscr{P}$ is attached a measure $\mathscr{M}(r)$

for all root r **do**
  $\mathscr{M}(r) := \emptyset$
end for
for all rb-path $r \to b$ from a non-singleton SCC **do**
  if there is a trace between an IAA A of $S(b)$ and an IAA A' of $S(r)$ **then**
    add A to $\mathscr{M}(rc)$ and A' to $\mathscr{M}(r)$, where rc is the companion of b
  end if
end for

At the beginning, the value attached to each root is the empty set. Then, for each *rb*-path from a cycle, denoted by $r \to b$, and for every trace along $r \to b$, leading some IAA of $S(r)$ to another IAA of $S(b)$, we add the corresponding IAAs to the values of r and the companion of b, respectively. Since the number of rb-paths is finite, Algorithm 1 terminates.

Algorithm 1 may compute values that do not pass the comparison test for some non-singleton SCCs that are validated by the model checker. For this case, we considered an improvement consisting of the incremental addition of IAAs from a root sequent that are not yet in the value of the corresponding root r. Since the validating orderings are trace-based variants of multiset extension orderings, such an addition does not affect the comparison value along the rb-paths starting from r. On the other hand, it may affect the comparison tests for the rb-paths ending in the companions of r. This may duplicate some IAAs from the values of the roots from the rb-paths leading to these companions. The duplicated IAAs have to be processed as any incrementally added IAA, and so on, until no changes are performed.

Table 1 illustrates some statistics about the proofs of the conjectures considered in Table 1 from [4], using inductive predicates as N, E, O, and Add, referring to the naturals, even and odd numbers, as well as the addition on naturals. All inductive predicates but p are defined in [4]. The proofs have been checked with the standard as well as our method. The IAAs are *indexed* in CYCLIST to facilitate the construction of traces; the way they are indexed influences how the pre-proofs are built. Different indexations for a same conjecture may lead to different proofs (see the statistics for the second and third conjectures). The column labelled 'Time-E' presents the proof time measured in milliseconds by using

our method. Similarly, the 'Time' column displays the proof time when using the standard method, while 'SC%' shows the percentage of time taken to check the soundness by the model checker. 'Depth' shows the depth of the proof, 'Nodes' the number of nodes in the proof, and 'Bckl.' the number of back-links in the proof. The last column gives the number of calls for pre-proof validations. The proof runs have been performed on a MacBook Pro featuring a 2,7 GHz Intel Core i7 processor and 16 GB of RAM. It can be noticed that, by using our method, the execution time is reduced by a factor going from 1.43 to 5.

Theorem	**Time-E**	Time	SC%	Depth	Nodes	Bckl.	Queries
$O_1x \vdash N_2x$	**2**	7	61	2	9	1	3
$E_1x \vee O_2x \vdash N_3x$	**4**	11	63	3	19	2	6
$E_1x \vee O_1x \vdash N_3x$	**2**	9	77	2	13	2	6
$N_1x \vdash O_2x \vee E_3x$	**3**	7	52	2	8	1	4
$N_1x \wedge N_2y \vdash Q_1(x,y)$	**297**	425	40	4	19	3	665
$N_1x \vdash Add_1(x,0,x)$	**1**	5	76	1	7	1	4
$N_1x \wedge N_2y \wedge Add_3(x,y,z) \vdash N_1z$	**8**	14	38	2	8	1	16
$N_1x \wedge N_2y \wedge Add_3(x,y,z) \vdash Add_1(x,sy,sz)$	**15**	22	32	2	14	1	14
$N_1x \wedge N_2y \vdash R_1(x,y)$	**266**	484	48	4	35	5	759
$N_1x \wedge N_2y \vdash p_1(x,y)$	**597**	?	?	4	28	3	2315

Table 1: Statistics for proofs checked with the standard and our method.

The last conjecture was not tested in [4] and refers to the 2-Hydra example [1]. A pre-proof of it, reproduced in Figure 1, can also be generated by CYCLIST, as shown in Figure 4. Unfortunately, CYCLIST was not able to validate it using the standard method, the missing figures being denoted by ?.

$$\dfrac{\dfrac{\vdash p00 \qquad \dfrac{N0 \vdash p10 \qquad \dfrac{(a)Nx,Ny \vdash pxy}{Nsz,Nz \vdash pszz}}{Nsz,Nz \vdash pssz0}}{Nx' \vdash psx'0}\,Nx}{Nx \vdash px0} \qquad \dfrac{\dfrac{N0,Nx \vdash px1 \qquad \dfrac{(a)Nx,Ny \vdash pxy}{Nsu,Nu \vdash psuu}}{Nsu,Nu \vdash p0ssu} \qquad \dfrac{\dfrac{(a)Nx,Ny \vdash pxy}{Nx',Nu \vdash px'u}}{Nsu,Nx',Nu \vdash psx'ssu}\,Nx}{Nsu,Nx,Nu \vdash pxssu}}{Nx,Ny' \vdash pxsy'}\,Ny$$
$$(a)Nx,Ny \vdash pxy$$

Figure 1: The Berardi and Tatsuta's cyclic pre-proof of the 2-Hydra example.

It also may occur that the proposed measure values, as shown in Figure 5 for a non-optimised proof of 2-Hydra, may not pass some comparison tests that succeed with the standard method, even when using the improved version of Algorithm 1. Indeed, this happened while proving $N_1x \wedge N_2y \vdash R(x,y)$. Luckily, the prover backtracked and finally found the same pre-proof as that originally built with CYCLIST.[1]

We detail now how our method has been applied for validating the 2-Hydra pre-proof from Figure 4.

The 2-Hydra case. The 2-Hydra problem is a particular case showing the termination of the battle between Hercules and Hydra [6] when Hydra has at most two heads that hang on the top of necks of

[1]The source code of the implementation and the examples can be downloaded at `https://members.loria.fr/SStratulat/files/e-cyclist.zip`

different lengths. Hercules prevails if either Hydra has i) no heads at all, or ii) the length of the first neck is 1 unit and it has lost the second head (i.e., the length of the neck is 0), or iii) the length of the second neck is 1 unit, as in Figure 2.

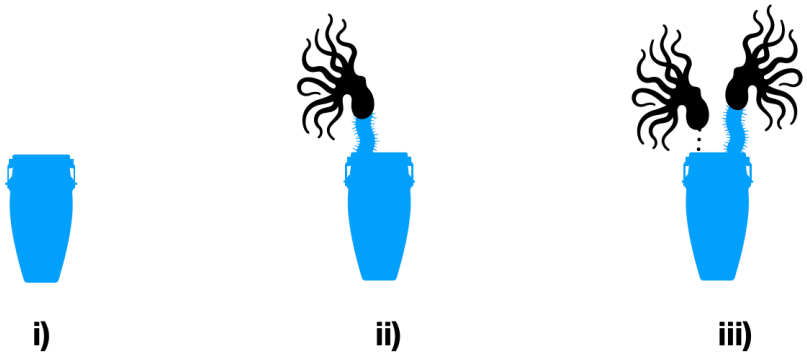


Figure 2: The cases when Hercules wins.

Hercules can cut the Hydra's necks according to the following rules. If both necks have strictly positive lengths, then Hercules can cut them such that the first neck is shorter by 1 unit and the second by 2 units (see the case iv in Figure 3). If Hydra has already lost one of the heads and the neck of the other head has a length l of at least 2 units, the first head will have a neck of length $l-1$ units and the second head a neck of length $l-2$ units (see the cases v and vi in Figure 3).

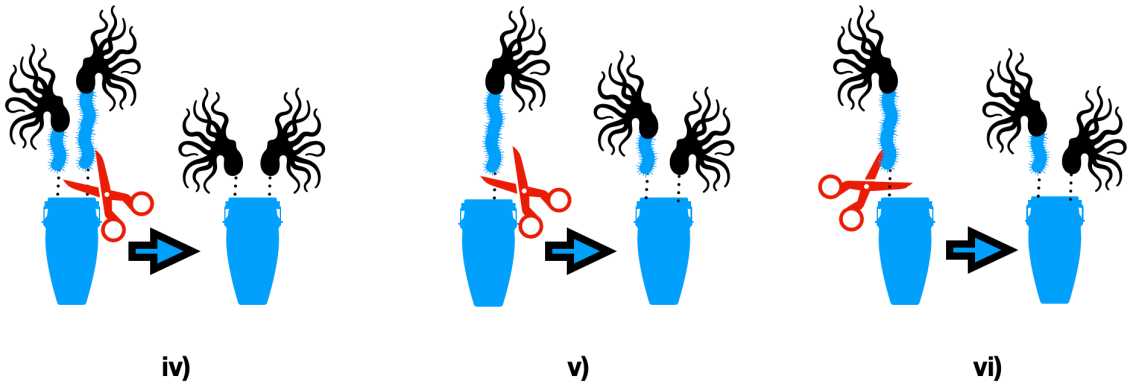


Figure 3: The cases when Hercules cuts the necks of Hydra.

Next, we introduce the notations, the specification of the inductive predicates, the inference rules, then explain the pre-proof from Figure 4. Contrary to the pre-proof from Figure 1, the CYCLIST pre-proof is horizontally indented by the level of nodes. The nodes are numbered and labelled by sequents where the comma (,) is replaced on the lhs of the sequents by the conjunction connector ($/\backslash$).

The axioms defining the inductive predicates N and p are:

$$\begin{array}{rrr} & \Rightarrow p(0,0) & p(x,y) \Rightarrow p(s(x),s(s(y))) \\ \Rightarrow N(0) & \Rightarrow p(s(0),0) & p(s(y),y) \Rightarrow p(0,s(s(y))) \\ N(x) \Rightarrow N(s(x)) & \Rightarrow p(x,s(0)) & p(s(x),x) \Rightarrow p(s(s(x)),0) \end{array}$$

The applied inference rule for each sequent is pointed out at the end of the sequent.

(N L.Unf) [n_1, n_2] generates the nodes n_1 and n_2 by choosing an IAA of the form $N(t)$. If t is a variable, t will be replaced by 0 and $s(z)$, where z is a fresh variable. For the second instantiation, the IAA is replaced by $N(z)$. This represents a *progres point*. If t is of the form $s(t')$, the original sequent is reduced to another sequent by replacing the chosen IAA $N(s(t'))$ with $N(t')$.

(p R.Unf) [n] produces the node n resulting from the replacement of the consequent atom from the sequent labelling n with the condition of some axiom defining p and whose conclusion matches the atom.

(Id) and **(Ex Falso)** delete trivial conjectures. **(Weaken)** (resp., **(Subst)**) [n] is the LK's weakening (resp., substitution) rule [7] whose premise labels n. Finally, **(Backl)** [n] shows that the current node is a bud for the companion n.

```
0: N_1(x) /\ N_2(y) |- p_1(x,y) (N L.Unf.) [1,2]
  1: N_1(x) /\ N_3(0) |- p_1(x,0) (N L.Unf.) [3,4]
    3: N_3(0) /\ N_4(0) |- p_1(0,0) (p R.Unf.) [5]
      5: N_3(0) /\ N_4(0) |- T (Id)
    4: N_1(y) /\ N_3(0) /\ N_4(s(y)) |- p_1(s(y),0) (N L.Unf.) [6,7]
      6: s(y)=0 /\ N_1(y) /\ N_3(0) /\ N_5(s(y)) |- p_1(s(y),0) (Ex Falso)
      7: N_1(y) /\ N_3(0) /\ N_4(y) /\ N_5(s(y)) |- p_1(s(y),0) (N L.Unf.) [8,9]
        8: N_1(0) /\ N_3(0) /\ N_5(s(0)) /\ N_6(0) |- p_1(s(0),0) (p R.Unf.) [10]
          10: N_1(0) /\ N_3(0) /\ N_5(s(0)) /\ N_6(0) |- T (Id)
        9: N_1(s(z)) /\ N_3(0) /\ N_4(z) /\ N_5(s(s(z))) /\ N_6(s(z)) |- p_1(s(s(z)),0) (p R.Unf.) [11]
          11: N_1(s(z)) /\ N_3(0) /\ N_4(z) /\ N_5(s(s(z))) /\ N_6(s(z)) |- p_1(s(z),z) (Weaken) [12]
            12: N_1(s(z)) /\ N_2(z) |- p_1(s(z),z) (Subst) [13]
              13: N_1(x) /\ N_2(y) |- p_1(x,y) (Backl) [0]
  2: N_1(x) /\ N_2(z) /\ N_3(s(z)) |- p_1(x,s(z)) (N L.Unf.) [14,15]
    14: N_2(z) /\ N_3(s(z)) /\ N_4(0) |- p_1(0,s(z)) (N L.Unf.) [16,17]
      16: N_3(s(0)) /\ N_4(0) /\ N_5(0) |- p_1(0,s(0)) (p R.Unf.) [18]
        18: N_3(s(0)) /\ N_4(0) /\ N_5(0) |- T (Id)
      17: N_2(y) /\ N_3(s(s(y))) /\ N_4(0) /\ N_5(s(y)) |- p_1(0,s(s(y))) (p R.Unf.) [19]
        19: N_2(y) /\ N_3(s(s(y))) /\ N_4(0) /\ N_5(s(y)) |- p_1(s(y),y) (Weaken) [20]
          20: N_1(s(y)) /\ N_2(y) |- p_1(s(y),y) (Subst) [21]
            21: N_1(x) /\ N_2(y) |- p_1(x,y) (Backl) [0]
    15: N_1(y) /\ N_2(z) /\ N_3(s(z)) /\ N_4(s(y)) |- p_1(s(y),s(z)) (N L.Unf.) [22,23]
      22: N_1(y) /\ N_3(s(0)) /\ N_4(s(y)) /\ N_5(0) |- p_1(s(y),s(0)) (p R.Unf.) [24]
        24: N_1(y) /\ N_3(s(0)) /\ N_4(s(y)) /\ N_5(0) |- T (Id)
      23: N_1(y) /\ N_2(w) /\ N_3(s(s(w))) /\ N_4(s(y)) /\ N_5(s(w)) |- p_1(s(y),s(s(w))) (p R.Unf.) [25]
        25: N_1(y) /\ N_2(w) /\ N_3(s(s(w))) /\ N_4(s(y)) /\ N_5(s(w)) |- p_1(y,w) (Weaken) [26]
          26: N_1(y) /\ N_2(w) |- p_1(y,w) (Subst) [27]
            27: N_1(x) /\ N_2(y) |- p_1(x,y) (Backl) [0]
```

Figure 4: The screenshot of the 2-Hydra pre-proof generated by CYCLIST.

The pre-proof from Figure 4 is already normalized and has one non-singleton SCC with three rb-paths.

Our validity method is based on properties to be satisfied *locally*, at the level of rb-paths. An rb-path $r \rightarrow b$ is *valid* if b is "smaller" than r w.r.t. a trace-based multiset extension relation. This relation guarantees the existence of traces following each infinite path p, built from the concatenation of the traces defined for the rb-paths along p. The definitions for the standard and trace-based multiset extension are:

- (*standard multiset extension*) $B <_{mul} A$ if there are two finite multisets X and Y such that $B = (A - X) \uplus Y$, $X \neq \emptyset$ and $\forall y \in Y$, $\exists x \in X, y < x$ holds.
- (*trace-based multiset extension*) b is "smaller" than r if, after pairwisely deleting the IAAs linked by a non-progressing trace along $r \to b$ (the result is X and Y as above), $X \neq \emptyset$ and $\forall y \in Y$, $\exists x \in X$ such that there is a progressing trace along $r \to b$ between x and y.

In Figure 5, we summarize the result of the application of the improved version of Algorithm 1 to a non-optimized version of the pre-proof from Figure 4, for which the node 27 was denoted as 28. The found measure of the root is the multiset of its IAAs indexed by 2 and 1, i.e., $\{N_2(x), N_1(y)\}$.

```
Measures proposed for the roots in cycles:
    0: [2, 1]
Checking the link of IAAs from buds to roots:
    28 to 0: | 1 -> 1 [true ]| 2 -> 2 [true ] ==> true
    21 to 0: | 1 -> 2 [true ]| 2 -> 2 [true ] ==> true
    13 to 0: | 1 -> 1 [true ]| 2 -> 1 [true ] ==> true
The proof has succeeded
```

Figure 5: The E-CYCLIST validation of the 2-Hydra pre-proof from Figure 4.

In Figure 5, for each rb-path, *i* -> *j* denotes that there is a trace linking the root IAA indexed by *j* to the bud IAA indexed by *i*, `[true ]` means that the trace is progressing, and '`===> true`' informs that the rb-path is valid, as follows:

1. 0 to 28 (27 in Figure 4); the possible traces following this path are: $[N_1(x), N_1(x), \underline{N_1(y)}, N_1(y), N_1(y), N_1(y), N_1(x)]$ and $[N_2(y), \underline{N_2(z)}, N_2(z), \underline{N_2(w)}, N_2(w), N_2(w), N_2(y)]$,
2. 0 to 21; the possible traces are: $[N_2(y), \underline{N_2(z)}, N_2(z), \underline{N_2(y)}, N_2(y), N_2(y), N_2(y)]$ and $[N_2(y), \underline{N_2(z)}, N_2(z), N_5(s(y)), N_5(s(y)), \overline{N_1(s(y))}, N_1(x)]$, and
3. 0 to 13; the possible traces are: $[N_1(x), N_1(x), \underline{N_1(y)}, N_1(y), N_1(s(z)), N_1(s(z)), N_1(s(z)), N_1(x)]$ and $[N_1(x), N_1(x), N_4(s(y)), \underline{N_4(y)}, \underline{N_4(z)}, N_4(z), \overline{N_4(z)}, N_2(y)]$.

All the above traces are progressing, where the underlined IAAs correspond to progress points. By definition, these rb-paths are valid and conclude that the 2-Hydra pre-proof is a proof, by using arguments as in [9, 10].

Conclusions and future work. We have implemented in CYCLIST a more effective technique for validating FOL$_{\text{ID}}$ cyclic pre-proofs which allows to speed up the proof runs by 5. Besides its polynomial time complexity, an important factor for its efficiency is the lack of the overhead time required to communicate with external tools. In practice, our method can validate pre-proofs that cannot be validated by the CSL-LICS14 release of CYCLIST. Even if we do not have yet a clear evidence, we strongly believe that this also holds for the other way around, as this might have happened for the $N_1 x \wedge N_2 y \vdash R(x, y)$ example.

The considered pre-proof examples are rather small. We intend to test our method more extensively and on cyclic pre-proofs from domains other than FOL$_{\text{ID}}$, e.g., separation logic.

References

[1] S. Berardi & M. Tatsuta (2019): *Classical System of Martin-Lof's Inductive Definitions is not Equivalent to Cyclic Proofs*. *Logical Methods in Computer Science* 15(3), doi:10.23638/LMCS-15(3:10)2019.

[2] J. Brotherston (2005): *Cyclic Proofs for First-Order Logic with Inductive Definitions*. In: *Proceedings of TABLEAUX-14*, *LNAI* 3702, Springer-Verlag, pp. 78–92, doi:10.1007/11554554_8.

[3] J. Brotherston (2006): *Sequent Calculus Proof Systems for Inductive Definitions*. Ph.D. thesis, University of Edinburgh.

[4] J. Brotherston, N. Gorogiannis & R. L. Petersen (2012): *A Generic Cyclic Theorem Prover*. In: *APLAS-10 (10th Asian Symposium on Programming Languages and Systems)*, *LNCS* 7705, Springer, pp. 350–367, doi:10.1007/978-3-642-35182-2_25.

[5] J. Brotherston & A. Simpson (2011): *Sequent calculi for induction and infinite descent*. *Journal of Logic and Computation* 21(6), pp. 1177–1216, doi:10.1093/logcom/exq052.

[6] N. Dershowitz & G. Moser (2007): *The Hydra Battle Revisited*. *Rewriting, Computation and Proof*, pp. 1–27, doi:10.1007/978-3-540-73147-4_1.

[7] G. Gentzen (1935): *Untersuchungen über das logische Schließen. I.* *Mathematische Zeitschrift* 39, pp. 176–210, doi:10.1007/BF01201353.

[8] M. Michel (1988): *Complementation is more difficult with automata on infinite words*. Technical Report, CNET.

[9] S. Stratulat (2017): *Cyclic Proofs with Ordering Constraints*. In R. A. Schmidt & C. Nalon, editors: *TABLEAUX 2017 (26th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods)*, *LNAI* 10501, Springer, pp. 311–327, doi:10.1007/978-3-319-66902-1_19.

[10] S. Stratulat (2018): *Validating Back-links of FOL_{ID} Cyclic Pre-proofs*. In S. Berardi & S. van Bakel, editors: *CL&C'18 (Seventh International Workshop on Classical Logic and Computation)*, *EPTCS* 281, pp. 39–53, doi:10.4204/EPTCS.281.4.