UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

SMT-based Flat Model-Checking for LTL with Counting

SMT-basiertes Flaches Model-Checking für LTL mit Zählquantoren

Masterarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Anton Pirogov

ausgegeben und betreut von
Prof. Dr. Martin Leucker

mit Unterstützung von
Normann Decker

Lübeck, den 15. September 2017

Abstract In this thesis a new approach for underapproximation-based model-checking is implemented. It relies on theoretical results in model-checking of temporal logics with counting capabilities over flat structures. The developed method uses the concept of *path schemas* to encode program runs which contain multiple loops compactly. Applied to general counter systems this yields a new approach that generalizes bounded model-checking in a way that in certain circumstances enables the calculation of counterexamples which otherwise would not be practically obtainable. After a formal introduction of all concepts the logic lcLTL is defined and proposed as a specification language. Then a translation of the existential model-checking problem for this logic into a satisfiability problem of quantifier-free Presburger arithmetic is described and finally a prototype implementation is discussed and evaluated on a selection of examples.

Kurzfassung In dieser Arbeit wird ein neuer Ansatz für unterapproximierendes Model-Checking implementiert. Er basiert auf theoretischen Ergebnissen bezüglich Model-Checking von Temporallogiken mit Zähloperationen auf flachen Strukturen. Die hier entwickelte Methode verwendet das Konzept der *Pfadschemata*, um Programmläufe mit mehreren Schleifen kompakt darstellen zu können. Angewandt auf allgemeine Zählersysteme erhält man dadurch eine Technik, die als Verallgemeinerung von Bounded-Model-Checking gesehen werden kann und es unter bestimmten Voraussetzungen ermöglicht, Gegenbeispiele zu finden, deren Berechnung ansonsten nicht praktikabel wäre. Nach einer formalen Einführung aller Konzepte wird die Logik lcLTL definiert und als Spezifikationssprache vorgeschlagen. Daraufhin wird das existenzielle Model-Checking-Problem für diese Logik als Erfüllbarkeitsproblem der quantorenfreien Presburgerschen Arithmetik kodiert. Schließlich wird eine Prototyp-Implementierung dieser Übersetzung diskutiert und an einer Auswahl von Beispielen evaluiert.

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne die Benutzung anderer als der angegebenen Hilfsmittel selbständig verfasst habe; die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe des Literaturzitats gekennzeichnet.

(Anton Pirogov)
Lübeck, den 15. September 2017

Acknowledgements

First, I would like to thank Prof. Leucker for giving me the opportunity to write this thesis. Next I want to thank Normann Decker wholeheartedly for his valuable feedback and all the helpful conversations about the topics of this thesis and beyond. He was truly the best supervisor a student could wish for and I am really grateful for all the support and guidance I received. Finally I want to thank Daniel Thoma for contributing the idea of using propagation chains to transport information, which was critical for making the encoding developed in this thesis efficient.

Contents

1 Introduction **1**

1.1 Software Verification Techniques 1

1.2 Model Checking 3

1.3 Contribution of This Thesis 6

1.4 Related Work 7

2 The Logic lcLTL **9**

2.1 Definitions and Examples 9

2.2 lcLTL in a Bigger Picture 16

2.3 Flat Model-Checking 18

3 From Flat Model-Checking to SMT **20**

3.1 Quantifier-free Presburger Arithmetic 20

3.2 Basic Structure 22

3.3 Labelling Positions with Subformulas 25

3.4 The Constrained **U** Operator 28

3.4.1 Difficulties and a Partial Solution 28

3.4.2 Evaluation of Constraints 32

3.5 Counter System Guards 38

3.6 The Complete Formula 40

3.7 Upper Bounds 44

3.8 Remarks 46

4 Implementation **49**

4.1 Choice of Technology 49

4.2 Overview and Interface 50

4.3 Calculation of the Simple Loop Lengths 51

4.4 Choice of the Path Schema Size 53

4.5 Theory Versus Practice 54

4.6 Additional Features 55

4.6.1 Disjunctions in Counter System Guards 55

4.6.2 Counter Systems with Resets 56

4.7 A Small Usage Example 57

5 Application **59**

5.1 Evaluating Scalability 59

5.1.1 Enumeration of the Cycle Lengths 59

5.1.2 A Simple Growable Benchmark 60

5.2 Combining Most Features 61

5.3 The RERS Challenge 2017 62

5.3.1 Modelling the Counter System 64

5.3.2 Adapting the LTL Formulas 64

5.3.3 Setting the Other Parameters 66

5.3.4 Results 67

6 Discussion **71**

6.1 Critical Reflection 71

6.2 Outlook 72

1 Introduction

In a world that depends on critical computer infrastructure it is of great importance to ensure its safe and correct operation. The diverse field of software verification is an endeavour to develop techniques to achieve this goal. This chapter provides a broad overview of different approaches in the field, describes the contribution of this thesis and puts it in the wider context.

1.1 Software Verification Techniques

One simple approach to reduce the number of mistakes in software is simply testing it. The importance of software testing is acknowledged in the industry, but just a minority seems to be practising it rigorously. The difficulty with testing is that the overhead for the software developer is quite high. Classical software testing requires much discipline from the developers, as each unit of code ideally must be covered by a set of multiple representative tests that should consider every imaginable behaviour and failure in that unit. Even when executed well, testing is just a "good enough"-approach to correctness, as in general no finite number of test cases can guarantee freedom of mistakes.

While not yet widespread in application, there are multiple approaches to software correctness that are being developed in the field of formal verification. These quite different approaches share only one assumption—the presence of some specification of the expected correct behaviour of a software system. The system is then verified against this specification.

In the one extreme, there is the field of computer-assisted *theorem proving*. One known approach for imperative programming languages involves the deduction of properties using *Hoare logic* [Hoa69], i.e. starting with a set of assumptions (called

preconditions) that hold before the execution of some code, to deduce *postconditions* that describe what can be said about the state of those variables afterwards. For sufficiently restricted languages this can be done mechanically.

In the realm of typed functional programming languages there exists much work based on a certain relationship of types in programming languages and logical propositions, the *Curry-Howard-correspondence* [How80]. It basically states that type signatures can be interpreted as logical propositions and the code that type-checks with that type signature is a proof for this proposition.

While being in itself an interesting and deep mathematical relationship between computation and logic, it has multiple practical applications. On the one hand, it can be used to codify and verify areas of constructive mathematics, by formulating the mathematical theorems in some type calculus, and providing the proof in form of a program witnessing that type. The program is either obtained by hand, or derived in a guided semi-automatic fashion. This is the rationale behind programming languages/proof assistants like *Coq* [DFH+04], which is based on an extension of the lambda calculus called *Calculus of constructions* [CH88] and *Isabelle* [NWP02], which is based on higher order logic.

On the other hand, the correspondence can be used for verification of software, by expressing the desired properties in a sufficiently sophisticated type system. The program should only compile successfully if it satisfies the type, and thereby the encoded property. This perspective is investigated in the development of programming languages like *Agda* [Nor07] and *Idris* [Bra13].

This ambitious approach has the big advantage of giving the strongest guarantees—the program is verified directly and can only compile if the specified properties are satisfied. The disadvantage is that now the developer needs to know two languages and their relationship very well—the actual programming language and the type language, which becomes a whole meta-language in its own right. Such powerful type systems are only comfortable with a good type inference (automatic deduction without explicit annotations), but inference becomes infeasible or even impossible with growing expressiveness of the types. So, using type systems for serious verification currently is tedious and requires sufficiently qualified developers with a strong mathematical background.

The other extreme is inhabited by the field of *runtime verification* [LS09, FHR13]. The philosophy here is to give up the ambition of static and universal verification, but to stay close to the actual system. In runtime verification, a program is extended with annotations emitting information about the current state of the system, basically a kind of logging. These logs can be evaluated on-the-fly or afterwards by *monitors*, programs generated from the desired properties, often encoded as a logical formula. The monitors provide a verdict based on these logs whether the property is satisfied or not. This can be used to find mistakes in programs during testing.

The advantage here is that a program can be observed in realistic action instead of artificially constructed test cases that employ simplifications to isolate components from each other. Also, applying runtime verification is much simpler than using an advanced type system and can be added retrospectively to some piece of code. The disadvantage is that runtime verification provides no additional guarantees, in the end it is just an extension of software testing that allows using each execution of the software as a test case. Like testing, it says nothing about the general behaviour of the software, just about a specific execution. Hence, it can only falsify, not verify a system.

1.2 Model Checking

A different approach that tries both to minimize the cognitive burden on the developer and to provide strong guarantees is *model checking* [CGP99]. In model checking, properties are checked over a representative abstraction of the system. Model checking ideally consists of automatic abstraction generation from the code and an exhaustive check of the desired properties. The vision is that the developer can write the code and specify some correctness properties and then use model-checking to automatically verify these or obtain a counterexample, i.e. an execution trace of the code where the property is violated. If no counterexample has been found, the developer then can be sure about the satisfaction of the property by the abstraction in the same way as he can be sure about properties enforced in the type system (as discussed above).

However, difficulties arise in all steps of the process. One area of research is the automatic extraction of an adequate representation of the system. It must be

adequate in the sense that the properties under investigation are mirrored correctly in this abstraction while irrelevant information is reduced as much as possible. For example, the presence of each 32-bit integer variable in a program multiplies the number of possible states of the system by 2^{32}, when representing the system faithfully. Parallelism also introduces state explosion due to the different states inside of the threads. The combinatorial state explosion quickly renders many kinds of programs infeasible for model checking, in the worst case the number of states can be infinite.

But the different values some variable can possibly take may be completely irrelevant for the property that is to be checked and in a program with multiple threads it may not make a difference in which order some sequence of instructions is executed by the threads, so it would suffice to check just one representative execution. By using some knowledge about the system one can reduce the state space accordingly. A technique that tries to reduce the number of equivalent execution paths that are checked is called *partial order reduction* [Val91].

Instead of trying to combat the state explosion by reducing redundant and irrelevant information without changing the meaning, one can tackle information reduction more radically by starting out with a very simplified model, for example by collapsing many different states into one by some equivalence criterion. A simple example would be a system with only one state for each line of code, i.e. ignoring the different values variables can have when executing that line. By collapsing states in such a way, additional sequences of states become possible that are not valid in the original system. For example, going out of a loop or repeating it usually depends on some condition. If the condition is abstracted away, one could leave the loop after any iteration, so a property can possibly be violated by such a spurious execution of the code, but not in the original system.

This kind of approach is called *overapproximation*, as these abstractions describe a superset of the possible executions of the program. This means, that if no counterexample is found, the property is also satisfied in the real system, but if one is found, it may be a spurious one. In this case, the abstraction must be refined, i.e. collapsed states must be split again to represent the distinct relevant cases and thereby prevent the impossible execution. This can be done iteratively until either no counterexample is found or a real counterexample is found which is

valid. This strategy is called *counterexample-guided abstraction refinement* (CEGAR) [CGJ+00].

Such techniques can be seen as a form of *abstract interpretation* [CC77], which tries to gain insight into the semantics of a program, e.g. by approximately evaluating it over some abstract value domain, e.g., by calculating on the abstract values $+, 0, -$ that represent a positive number, a negative number or zero, if some property of interest just needs this information to determine satisfaction. In the case of CEGAR, the set of the obtained predicates is the abstract domain that is refined in each iteration.

A, in some sense, dual approach is called *underapproximation*. In underapproximation techniques the used abstractions describe a subset of the real system. This means that all counterexamples which can be found are also valid in the real system, but there is no guarantee that an existing counterexample can be actually found. The most prominent approach is called *bounded model-checking* [BCCZ99]. Here, the problem is restricted to finding a property-violating execution with some specified maximum length. Usually, bounded model-checking is implemented by a translation of the system and the desired property into a logical formula. A logical formula usually has a fixed number of variables, thereby prohibiting the encoding of arbitrarily long representations of program executions in the same formula, making this approach an underapproximation.

Techniques like bounded model-checking are also suitable for *existential model-checking*, as they can falsify a property by proving the existence of a counterexample. This contrasts with the usual approach of *universal model-checking*, which tries to verify a system by proving the absence of violating executions. When searching for mistakes in software, the existential approach can be applied in a similar use-case as runtime verification.

The formulas solved in bounded model-checking are usually expressed in propositional logic or some superset that is still practically feasible for automatic solving with specialized external tools, like SAT-solvers. The more powerful variety of SAT-solvers that are capable of solving formulas expressed in a richer logic are called *SMT-solvers*. SMT means “satisfiability modulo theories” which basically indicates the fact that multiple extensions of the logic and thereby expressible theorems are supported, e.g. formulas including integer variables, arrays, lists, etc.

SAT- and SMT-solving is a whole separate research area on its own and modern tools use a plethora of techniques and heuristics to achieve good performance on a varied range of formulas despite the fact that the satisfiability problem of all supported theories is usually at least **NP**-hard.

Both model checking and runtime verification often use some temporal logic to specify the properties that are to be verified. Temporal logics are extensions of propositional logic with operators that allow for expressing statements that contain a notion of time. The most well-known logics for this purpose are *linear temporal logic* (LTL) [Pnu77] and *computation tree logic* (CTL) [CE82]. Neither can express all properties expressible by the other, but there exists a superset of both logics called CTL*. Nevertheless, LTL and CTL are more widely used in practice, as these logics lend themselves for some efficient algorithmic techniques.

LTL is defined over execution traces, i.e., words over some finite alphabet. The two basic temporal operators in LTL are *next* (**X**) and *until* (**U**). The unary **X** allows for specifying what must be true in the next time-step while **U** is a binary operator requiring that at some point in the future the formula on the right holds and from now until then the left formula must hold continuously. There are many extensions and variants of LTL, including operators for different new capabilities beyond just checking whether a proposition holds.

CTL formulas are defined over the whole computation tree of a program which includes branches for all possible executions and each temporal operator is quantified universally or existentially, specifying whether it suffices that one branch exists that satisfies the formula or that all following branches must satisfy it. LTL formulas are defined over single execution traces of a program and an LTL formula is satisfied for a program if all possible executions satisfy it.

1.3 Contribution of This Thesis

In this thesis an approach with some similarity to bounded model-checking is investigated which we will call *flat model-checking*. It can be seen as a generalization of the former in the sense that it extends it with the possibility of a compact loop representation, effectively enhancing BMC with a technique for loop acceleration.

This approach is applied to the search for counterexamples for formulas of an extension of LTL that allows for the possibility to calculate linear arithmetic expressions over the number of positions in a specified range where different subformulas were satisfied. Additionally, the approach allows for using a stateful abstraction of the system that also can express counting operations directly.

After formally describing the setting and defining the logic, first a translation is presented that maps the model-checking problem for this logic into a satisfiability problem of a restricted first-order logic called *Presburger arithmetic*, which is a decidable theory [Pre29] that is known to be practically solvable by SMT-solvers [DMB08, BCD+11]. Finally a prototype implementation of the described approach is discussed and evaluated.

1.4 Related Work

The idea of bounded model-checking, i.e. using SAT-solvers to approach model-checking problems, has been first proposed in [BCCZ99], where it is compared to techniques based on binary decision diagrams and a prototype implementation for bounded LTL model-checking is presented. A more practice-oriented introduction is given by the same authors in [CBRZ01].

Bounded model-checking using SMT-solvers is presented in [AMP06] as a generalization of SAT-based bounded model-checking, proposing the translation of a program into a quantifier-free formula in a richer background theory that includes arithmetic and array manipulation instead of plain propositional logic, resulting in considerably smaller formulas and better scaling with program size.

This approach is pushed further in [CFM09] by providing a translation of many constructs of ANSI-C programs into quantifier-free formulas, including bit operations, floating-point and pointer arithmetic. The tool CBMC is presented with promising results on embedded software benchmarks.

The problem of an efficient handling of loops is being investigated in many subdisciplines of the software verification community. In [KLW15] an underapproximation technique for loops in C programs is presented which approximates the effect of multiple loop iterations. The approach is evaluated in combination with the bounded model-checker CBMC.

In the context of predicate abstraction techniques, multiple approaches to *loop acceleration* have been investigated. The authors of [CFLZ08] and [HIK+12] present different approaches that combine CEGAR with interpolation-based acceleration techniques to reduce families of invalid counterexamples induced by loop unrollings. In [BHMR07] an approach is described where *path programs* are used to derive *path invariants* that prove the infeasibility of families of counterexamples represented by the path program. A similar approach based on the computation of invariants is presented in [KW10].

In the context of abstract interpretation approaches, the paper [JSS14] presents a technique based on *abstract matrices* and linear algebra to compute the exact effect of arbitrary linear loops, i.e. loops containing linear assignments and guards.

In [BDL12] the logic fLTL is defined which allows for expressing the requirement that some formula must hold just sufficiently often (specified by a fraction), instead of always. The satisfiability problem of fLTL is shown to be undecidable and a decidable fragment is identified. The logic presented in this thesis is a generalization of fLTL.

A key idea utilized in this thesis is using linear state sequences with non-overlapping loops as underapproximations to prevent needless unrolling of loops. A similar idea is used in [BFLS05], where it is presented under the term *restricted linear regular expressions* in the context of symbolic model-checking of infinite-state systems and is used for efficient calculation of reachable states.

In [DDS15] this concept is introduced under the term *path schemas*, where it is used to obtain a decomposition of structures exhibiting a certain flatness property and prove an upper bound for the complexity of model-checking for an LTL logic with past operators. A logical encoding for the set of all runs in a path schema that grows quadratically in the schema size is performed and used in a nondeterministic algorithm with multiple successive guesses. In contrast, the construction in this thesis is linear in the schema size, encodes just a single run, does not fix a path schema and is better suited for a practical SMT-based implementation by encoding the whole problem in one formula. An approach based on path schemas is also used in [DHL+17] to prove the decidability of fLTL model-checking over flat structures. Some ideas from the witness construction in that proof served as an inspiration for aspects of the encoding presented in this thesis.

2 The Logic lcLTL

In this chapter the logic lcLTL and most other required structures and concepts used in this thesis are formally defined and illustrated. Then we will see where this logic is located in a broader theoretical framework.

2.1 Definitions and Examples

Definition 2.1 (Numbers and words)**.** *Let* $\mathbb{N}$ *denote the set of natural numbers* $\{0, 1, 2, \ldots\}$ *and* $\mathbb{Z}$ *the set of integers* $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$*. Further, let* $[i, j]$ *define the closed interval of integer numbers* $\{k \in \mathbb{Z} \mid i \leq k \leq j\}$*.*

For a finite alphabet Σ*,* Σ^* *denotes the set of finite words (sequences) over* Σ*,* Σ^ω *denotes the set of infinite words (sequences) over* Σ *and for some word* $w = w_0 w_1 w_2 \ldots$ $w(i) = w_i$ *is the* $(i+1)$*th symbol of the word and* $|w|$ *denotes the length of the word (and equals* ω *for infinite words). The expression* w^ω *denotes the infinite sequence of repetitions of* w*.*

Definition 2.2 (Kripke structures)**.** *Let* AP *be a finite set of* atomic propositions*. A* Kripke structure *is a tuple* $\mathcal{K} = (S, s_I, E, \lambda)$ *where* S *is the set of states,* $s_I \in S$ *is the initial state,* $E \subseteq S \times S$ *is the set of directed edges and* $\lambda : S \to 2^{AP}$ *is the node labelling function.*

A finite path *in* $\mathcal{K}$ *is a finite sequence of states* $w = s_0 s_1 \ldots s_n \in S^+$ *with* $(s_i, s_{i+1}) \in E$ *for all* $0 \leq i < n$*. A* run *in* $\mathcal{K}$ *is an infinite sequence of states* $w = s_0 s_1 s_2 \ldots \in S^\omega$ *such that* $s_0 = s_I$ *and for all positions* $i > 0 : (s_{i-1}, s_i) \in E$*, i.e. an infinite path starting at* s_I*. A* simple loop *in* $\mathcal{K}$ *is a finite path* $w = s_0 s_1 \ldots s_n$ *such that for all* $0 \leq i, j \leq n$*,* $i \neq j$ *implies* $s_i \neq s_j$ *and* $(s_n, s_0) \in E$*.*

The structure $\mathcal{K}$ *is called* flat *if for each state* $s \in S$ *there is at most one simple loop* w *in* $\mathcal{K}$ *with* $w(0) = s$*, the size* $|K|$ *is defined as the number of edges* $|E|$*.*

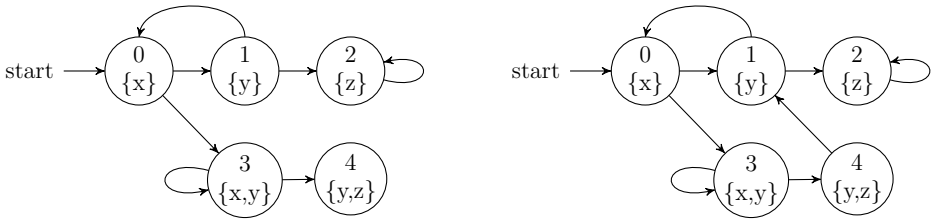


Figure 2.1: Example of Kripke structures. A flat structure on the left, a non-flat structure on the right. The additional edge makes some states part of multiple different loops. Numbers identify states, sets represent labels.

A Kripke structure is just a directed graph with a specified start node (also called *transition system*) where nodes are labelled with sets of atomic propositions that are true at this node.

Definition 2.3 (Counter systems)**.** *A* counter system *is a tuple* $\mathcal{S} = (\mathcal{K}, C, \delta)$ *where* $\mathcal{K} = (S, s_I, E, \lambda)$ *is a Kripke structure,* C *is a finite set of counter names and* $\delta(e) = (\delta_u(e), \delta_g(e))$*,* $\delta : E \to \mathbb{Z}^C \times 2^{\mathfrak{G}}$ *is an edge labelling function that assigns each edge an* update vector *and a set of* guards *from* $\mathfrak{G} = \{(k_c, \oplus, k) \mid k_c \in \mathbb{Z}^C, \oplus \in \{<, \geq\}, k \in \mathbb{Z}\}$*.*

Let $\delta_{u(c)}(e) = \delta_u(e)(c)$ *denote the constant update value for counter* $c \in C$ *at edge* $e \in E$*.*

A run *in* $\mathcal{S}$ *is an infinite sequence* $w = (s_0, v_0)(s_1, v_1) \ldots \in \left(S \times \mathbb{Z}^C\right)^\omega$ *so that*

- $s = s_0 s_1 \ldots \in S^\omega$ *is a run in* $\mathcal{K}$
- $\forall c \in C : v_0(c) = 0$ *and* $\forall_{i>0} : v_i(c) = v_{i-1}(c) + \delta_{u(c)}(s_{i-1}, s_i)$
- $\forall_{i>0} : \forall (k_c, \oplus, k) \in \delta_g(s_{i-1}, s_i) : \sum_{c \in C} k_c(c) \cdot v_i(c) \oplus k$

$\mathsf{Runs}(\mathcal{S})$ *denotes the set of all runs in* $\mathcal{S}$*.* $\mathcal{S}$ *is called* flat *if* $\mathcal{K}$ *is flat. The size* $|S|$ *is defined as* $|E| + |C| + \sum_{e \in E} |\delta_g(e)|$*.*

So a run in a counter system is just a run in the underlying Kripke structure that additionally updates the counters accordingly and satisfies the edge guard conditions. An edge can only be traversed, if the counters satisfy the guard after applying the update of that edge.

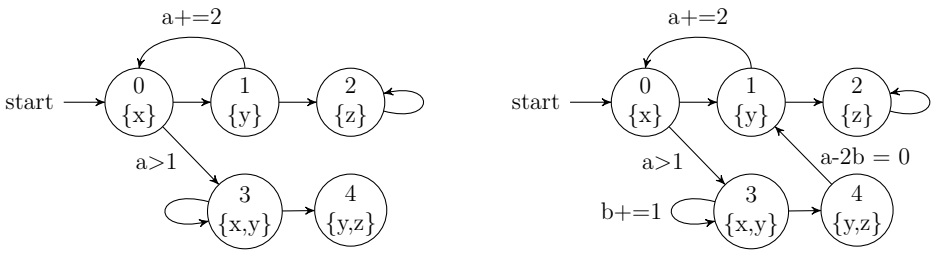


Figure 2.2: Example of flat and non-flat counter systems based on the previous example by adding counter updates and guards. The flatness properties are unchanged.

Consider the counter system on the right in figure 2.2. An infinite sequence starting with $(0, \binom{a=0}{b=0})(3, \binom{a=0}{b=0})(3, \binom{a=0}{b=1})\ldots$ is not a run in this system, because the edge from state 0 to state 3 cannot be taken with $a \leq 1$, hence this sequence describes an invalid transition that violates the guard. On the other hand, the infinite sequence $(0, \binom{a=0}{b=0})(1, \binom{a=0}{b=0})(0, \binom{a=2}{b=0})(3, \binom{a=2}{b=0})(3, \binom{a=2}{b=1})(4, \binom{a=2}{b=1})(1, \binom{a=2}{b=1})(0, \binom{a=4}{b=1})\ldots$ that infinitely loops the state sequence $3, 3, 4, 1, 0$ with increasing counters is a valid run in that system, because with this state visiting sequence the counters always satisfy the guard conditions.

Definition 2.4 (Path schema)**.** *Let $\mathcal{S} = (S_\mathcal{S}, s_{I\mathcal{S}}, E_\mathcal{S}, \lambda_\mathcal{S}, C, \delta_\mathcal{S})$ be an arbitrary counter system. A* path schema *$\mathcal{P} = (S_\mathcal{P}, s_{I\mathcal{P}}, E_\mathcal{P}, \lambda_\mathcal{P}, C, \delta_\mathcal{P})$ for $\mathcal{S}$ is a flat counter system with a total ordering $\leq$ on the states and in- and outdegree of at most 2 that shares the atomic propositions AP and counters C with $\mathcal{S}$ and there exist functions $\sigma : S_\mathcal{P} \to S_\mathcal{S}, \zeta : E_\mathcal{P} \to E_\mathcal{S}$ such that:*

- $\forall_{s \in S_\mathcal{P}} : \lambda_\mathcal{P}(s) = \lambda_\mathcal{S}(\sigma(s))$
- $\forall_{(a,b)=e \in E_\mathcal{P}} : (\sigma(a), \sigma(b)) = \zeta(e) \in E_\mathcal{S} \wedge \delta_\mathcal{P}(e) = \delta_\mathcal{S}(\zeta(e))$
- $\forall_{(a,b)=e \in E_\mathcal{P}} : b \leq a \vee \nexists_{s \in S_\mathcal{P}}, s \neq a \wedge s \neq b \wedge a < s < b$

A run *in $\mathcal{P}$ is an infinite sequence $w = (s_0, v_0)(s_1, v_1)\ldots \in \left(S_\mathcal{P} \times \mathbb{Z}^C\right)^\omega$ such that $\hat{w} = (\sigma(s_0), v_0)(\sigma(s_1), v_1)\ldots \in \left(S_\mathcal{S} \times \mathbb{Z}^C\right)^\omega$ is a run in $\mathcal{S}$. Let $\mathcal{P}(\mathcal{S})$ denote the set of all path schemas consistent with the counter system $\mathcal{S}$ and let $|\mathcal{P}| := |S_\mathcal{P}|$.*

A path schema can be thought to be a linear chain of states, where some states possibly have a self-loop or loop to a previous state and these loops never overlap. Each state and each edge of a path schema is identified with a state or edge, respectively,

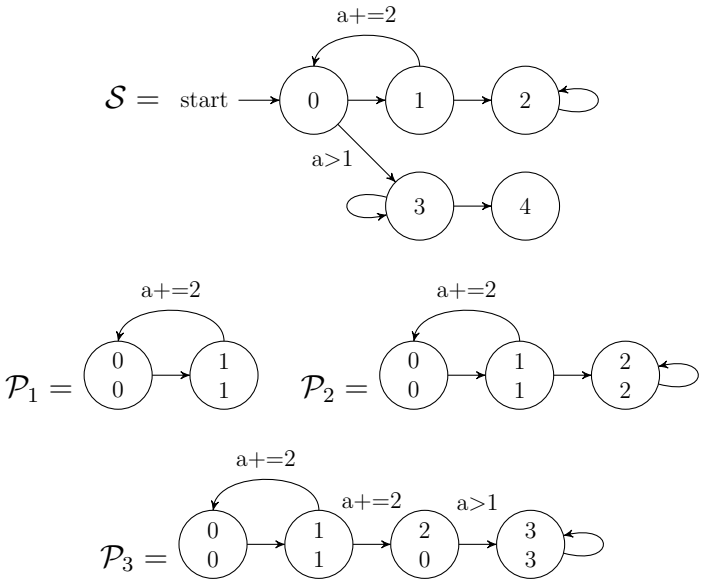


Figure 2.3: A finite set of path schemas that is sufficient to express all possible runs in the flat counter system $\mathcal{S}$. In the schemas the number at the bottom of state labels indicates the state in the original system.

in some other counter system $\mathcal{S}$ and each run in a path schema can be simulated via the provided morphisms. We can assume that path schemas always end with a loop—no run can visit non-loop states at the end of a path schema, because then the sequence cannot continue. Because a path schema can represent a subset of runs in a different system, we will also refer to them as *flat underapproximations*.

Notice that if $\mathcal{S}$ is flat, for every run in $\mathcal{S}$ we can find a finite path schema $\mathcal{P}$ by restricting it to contain only loops and linear path segments that are visited by the run, because due to flatness we can never return back to a loop after we left it. So the upper bound for the required path schema size is proportional to the size of $\mathcal{S}$ [DDS15, Lemma 3.2].

Figure 2.3 shows path schemas that allow for expressing any infinite run in the depicted system $\mathcal{S}$. As runs by definition must be infinite and flatness ensures that each loop is entered at most once, in a flat counter system a run has always the form of a sequence of different finitely repeated loops connected by linear segments and a final loop that is repeated infinitely often. Observe that in the example we could forever stay in the loop between the states 0 and 1 ($\mathcal{P}_1$). We could also decide to leave and enter state 2 and loop there forever ($\mathcal{P}_2$) or we could traverse the first

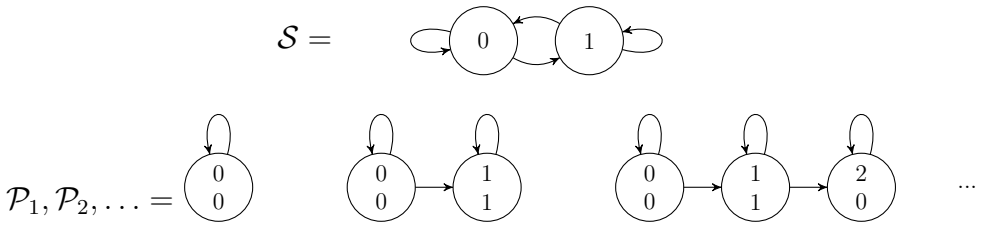


Figure 2.4: A non-flat counter system $\mathcal{S}$. No finite set of path schemas can represent all runs. Initial states are always called 0 and will not be marked explicitly in upcoming examples.

loop at least once to increase to counter and then switch to state 3 and loop there forever. Also you can see that state 4 can never be part of a run as the sequence cannot progress there.

On the other hand, if $\mathcal{S}$ is not flat, we cannot represent every infinite run in terms of a finite path schema if the run is not periodic and especially we cannot use a path schema with some fixed upper bound on the size, as in $\mathcal{S}$ the loops can be alternated arbitrarily often, forcing us to provide copies in the path schema due to the flatness constraint. You can see a simple example in figure 2.4. Hence, for non-flat counter systems a finite set $\{\mathcal{P}_i\}$ of path schemas with some fixed maximum size is just an underapproximation, i.e. each run in one of the schemas $\mathcal{P}_i$ has a corresponding run in $\mathcal{S}$, but not every run in $\mathcal{S}$ is representable in one of the schemas.

Definition 2.5 (lcLTL syntax)**.** *Let AP be a finite set of atomic propositions and $\mathfrak{C}$ be a set of linear constraints over formulas with the form $\sum_{i=0}^{n} k_i\varphi_i \oplus k$ for some $k_i, k \in \mathbb{Z}, n \in \mathbb{N}, \oplus \in \{<, \geq\}$ and lcLTL-formulas φ_i.*

The syntax of lcLTL is defined for $p \in AP$ and $c \in \mathfrak{C}$ by the following grammar:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}[c]\varphi$$

Additionally following notations are defined:

$$\begin{aligned}
\mathsf{true} &:= p \vee \neg p \\
\mathsf{false} &:= p \wedge \neg p \\
\varphi\mathbf{U}\psi &:= \varphi\mathbf{U}[1 \cdot \mathsf{true} \geq 0]\psi \\
\mathbf{F}[c]\varphi &:= \mathsf{true}\mathbf{U}[c]\varphi \\
\mathbf{G}[c]\varphi &:= \neg\mathbf{F}[c]\neg\varphi
\end{aligned}$$

$\Phi = \varphi \mathbf{U}[2\xi > 0]\psi \qquad w = \overset{[\ 0+\ \ 2\ \ +0+0\ ]=2>0}{\rightarrow\{\varphi\}\rightarrow\{\varphi,\xi\}\rightarrow\{\varphi\}\rightarrow\{\varphi\}\rightarrow\{\psi\}\rightarrow\emptyset^\omega} \qquad \Rightarrow$ satisfied

$\Phi = \varphi \mathbf{U}[2\xi > 0]\psi \qquad w = \rightarrow\{\varphi\}\rightarrow\{\xi,\varphi\}\rightarrow\{\varphi\}\rightarrow\ \emptyset\ \rightarrow\{\psi\}\rightarrow\emptyset^\omega \qquad \Rightarrow \varphi\mathbf{U}\psi$ violated

$\Phi = \varphi \mathbf{U}[2\xi > 0]\psi \qquad w = \overset{[\ 0+0+0+0\ ]=0>0}{\rightarrow\{\varphi\}\rightarrow\{\varphi\}\rightarrow\{\varphi\}\rightarrow\{\varphi\}\rightarrow\{\psi\}\rightarrow\emptyset^\omega} \qquad \Rightarrow$ constraint violated

$\Phi = \varphi \mathbf{U}[2\xi > 0]\psi \qquad w = \overset{[\ ]=0>0}{\rightarrow\{\xi,\psi\}\rightarrow\emptyset^\omega} \qquad \Rightarrow$ constraint violated for empty scope

$\Phi = \varphi \mathbf{U}[2\xi \geq 0]\psi \qquad w = \overset{[\ ]=0\geq 0}{\rightarrow\{\xi,\psi\}\rightarrow\emptyset^\omega} \qquad \Rightarrow$ satisfied (even with empty scope)

Figure 2.5: Illustration of $\mathbf{U}[\ldots]$ semantics for different situations that can arise. The satisfaction is always discussed wrt. the first position of the sequence.

Definition 2.6 (lcLTL semantics)**.** *Let $\mathcal{S}$ be a counter system, w a run in $\mathcal{S}$, $i \in \mathbb{N}$ and φ, ψ, η_i* lcLTL*-formulas over the atomic propositions AP. The model relation is defined as:*

$$
\begin{aligned}
(w,i) &\models p && :\Leftrightarrow && p \in \lambda(w(i)) \\
(w,i) &\models \neg\varphi && :\Leftrightarrow && (w,i) \not\models \varphi \\
(w,i) &\models \varphi \wedge \psi && :\Leftrightarrow && (w,i) \models \varphi \text{ and } (w,i) \models \psi \\
(w,i) &\models \varphi \vee \psi && :\Leftrightarrow && (w,i) \models \varphi \text{ or } (w,i) \models \psi \\
(w,i) &\models \mathbf{X}\varphi && :\Leftrightarrow && (w,i+1) \models \varphi \\
(w,i) &\models \varphi \mathbf{U} \left[\sum_{j=0}^{n} k_j \eta_j \oplus k\right] \psi && :\Leftrightarrow && \exists_{l \geq i} : (w,l) \models \psi \text{ and } \forall_{i \leq m < l} : (w,m) \models \varphi \\
& && && \text{and } \sum_{j=1}^{n} k_j \cdot \#^w_{[i,l-1]}(\eta_j) \oplus k \ ,
\end{aligned}
$$

where $p \in AP, k_i, k \in \mathbb{Z}, n \in \mathbb{N}, \oplus \in \{<, \geq\}$ and $\#^w_I(\varphi) := |\{(w,i) \models \varphi \mid i \in I\}|$ denotes the function that counts in how many positions of a word a given formula is satisfied for a given index set. We also write $w \models \varphi$ for $(w,0) \models \varphi$.

The linear constraint $\mathbf{U}$ operator, to which we will refer as $\mathbf{U}[\ldots]$ from now on, is more expressive than the regular $\mathbf{U}$ operator by the fact that additionally to its

usual meaning the sequence of φ positions from the current position up to the last position before ψ defines a region where the satisfaction of different subformulas can be counted and calculated with. The $\mathbf{U}[\ldots]$ formula is then only satisfied, if the linear constraint over these counts is also satisfied.

The linear constraints also have an intuitive meaning when used with other temporal operators that are expressible with $\mathbf{U}[\ldots]$, e.g. $\mathbf{F}[c]\varphi$ holds if at some point in the future φ holds and constraint c is satisfied and $\mathbf{G}[c]\varphi$ holds if in all positions where constraint c is satisfied the formula φ holds. Here the constraint acts as a filter for the considered positions and can be seen as a weakening of the $\mathbf{G}$ operator.

The possible cases that can occur with the $\mathbf{U}[\ldots]$ operator are illustrated in figure 2.5. Now let us look at a more concrete example. Consider a counter system that represents some network device. Assuming an according modelling of the counter system, we can express properties like

$$\mathsf{connected}\mathbf{U}[\mathsf{recv} \leq 8]\mathsf{close}$$

stating that during a connection at most 8 packets are received, or

$$\mathbf{G}[\mathsf{open} - \mathsf{close} < 0]\mathsf{false} \wedge \mathbf{G}[\mathsf{open} - \mathsf{close} > 5]\mathsf{false}$$

demanding that no more than five network connections are open at any time and also ensuring that at any time not more close commands have been issued that open commands, which is a necessary (but not sufficient) condition to verify that no connection is closed more than once.

We can also model more complex properties and even nest the quantifier:

$$\mathbf{G}[\mathsf{connected}\mathbf{U}[\mathsf{error} > 3]\mathsf{close} \geq 5]\mathsf{shutdown}$$

This formula expresses the property that the system shall shutdown itself after it had more than five connections in which more than three errors were noticed.

While it is technically possible to translate simple formulas to classical LTL, the numeric constraints would blow up the formula and make an actual verification

infeasible, e.g. consider these formulas and their translations:

$$\begin{aligned} pU[r \geq 2]q \quad &\Leftrightarrow \quad p\mathbf{U}(p \wedge r \wedge \mathbf{X}(p\mathbf{U}(p \wedge r \wedge \mathbf{X}(p\mathbf{U}q)))) \\ pU[r \leq 1]q \quad &\Leftrightarrow \quad (p \wedge \neg r)\mathbf{U}q \vee (p \wedge \neg r)\mathbf{U}((p \wedge r) \wedge \mathbf{X}((p \wedge \neg r)\mathbf{U}q)) \end{aligned}$$

More subformulas in the constraint would add a combinatorial number of possibilities that need to be considered, hence the constraints in lcLTL add a finite and compact representation for a natural extension of LTL.

2.2 lcLTL in a Bigger Picture

The logic called fLTL that is defined in [BDL12] and used in [DHL+17] is a fragment of lcLTL, as its frequency constraint $\mathbf{U}^{\frac{n}{m}}$ operator can be expressed using $\mathbf{U}[\ldots]$ of lcLTL in the following way:

$$\varphi\mathbf{U}^{\frac{n}{m}}\psi \quad := \quad \mathbf{F}\left[m \cdot \varphi - n \cdot \mathsf{true} \geq 0\right]\psi \qquad\qquad 0 \leq \frac{n}{m} \leq 1$$

The intuition behind the $\mathbf{U}^{\frac{n}{m}}$ operator is the following. The usual requirement that φ must hold in $\varphi\mathbf{U}\psi$ in every position until ψ holds is relaxed. Instead there must be a position where ψ holds and φ held in $\geq \frac{n}{m}$ of the positions between the current position and that future position where ψ holds. Setting $n = m = 1$ recovers the usual $\mathbf{U}$ semantics.

A peculiarity of the $\mathbf{U}^{\frac{n}{m}}$ operator is the fact that for ratios with $\frac{n}{m} < 1$, every sequence can be extended in such a way that its ratio condition is satisfied. It is not possible to restrict the scope in a simple way, which makes modelling of nontrivial problems with this operator rather cumbersome. The $\mathbf{U}[\ldots]$ operator of lcLTL provides a scoping capability directly and hence offers an operator which is more versatile and also easier to understand and use.

The logic lcLTL is itself a fragment of CCTL* [DHL+17], because the $\mathbf{U}[\ldots]$ operator can be expressed in CCTL* in the following way:

$$\varphi\mathbf{U}\left[\sum_{i=0}^{n} k_i\eta_i \oplus k\right]\psi \quad := \quad x\ .\ \left(\varphi\mathbf{U}\Big((\mathbf{X}\psi) \wedge \sum_{i=0}^{n} k_i \cdot \#_x(\eta_i) \oplus k\Big)\right) \vee (\psi \wedge 0 \oplus k)$$

A formula $x.\varphi$ starts a scope at the current position that can be referred to inside φ. The scope can be used in arithmetic expressions with $\#_x(\psi)$, an operator that counts the positions where some other formula ψ holds from the starting position marked by x to the current position. More details can be found in [DHL+17].

Also notice that the $\mathbf{U}[\ldots]$ operator that we use here for LTL is very similar to the one used in [LMP10, LMP12]. In [LMP12] variations of this operator were investigated as an extension of CTL and the logic CLTL in [LMP10] is syntactically equivalent to lcLTL, but allows only natural coefficients in the constraints and is no more expressive than LTL because of this restriction.

Satisfiability of fLTL has been shown to be undecidable in [BDL12] and a simple reduction shows that existential model-checking of fLTL over general Kripke structures is also undecidable (using a fully connected graph with each node labelled with a distinct set of atomic propositions suffices), which means that lcLTL satisfiability and existential model-checking over general Kripke structures is also undecidable. On the other hand, existential model-checking of fLTL over flat Kripke structures has been shown to be in **NExp** in [DHL+17] and might as well be in **NP** as this is currently the best known lower bound for this problem [DHL+17, section 6], giving also a lower bound for the complexity of existential lcLTL model-checking over flat structures.

In the same paper, existential model-checking of CCTL* over flat structures has been shown to be reducible to Presburger arithmetic with the so-called Härtig quantifier, which is known to be decidable. Presburger arithmetic is first-order logic over natural numbers with addition, but without multiplication and has been shown to be decidable in [Pre29]. The Härtig-quantifier $\exists^{=x}$ is an existential quantifier that allows for specifying the number of values that satisfy a formula and is known to be a decidable extension of Presburger arithmetic [Ape66, HKPV91]. These results give an upper bound for the complexity of lcLTL model-checking over flat structures.

In fact, it appears that the proof for fLTL in [DHL+17] can be adapted, indicating that the complexity of existential lcLTL model-checking over flat structures might be the same as for fLTL, i.e. at most **NExp**. While out of the scope of this thesis, it should be possible to adapt the proof to lcLTL in a rather simple way, because the only case that needs to be changed is the case for $\mathbf{U}[\ldots]$ instead of $\mathbf{U}^{\frac{n}{m}}$. In the proof a witnessing path schema is constructed in such a way that for each $\varphi\mathbf{U}^{\frac{n}{m}}\psi$-subformula

a counter c keeps track of the balance of positions where φ is or is not satisfied and the witnessing ψ position is guarded with $c \geq 0$, which by construction is only the case if the supplied fraction constraint is satisfied. The same idea could be applied to keep track of a more general constraint as used in $\mathbf{U}[\ldots]$, by incrementing the counter at φ positions, accordingly weighted by the coefficients assigned to the subformulas in the constraint. Additionally, some care must be taken to restrict the scope to continuous φ sequences. In fact, this is very similar to what is done in the translation in the next chapter and in principle could also be used in a formal proof.

2.3 Flat Model-Checking

Definition 2.7 (Model-checking problems for lcLTL)**.** *Let $\mathcal{S}$ be a counter system and φ an* lcLTL *formula. We say $\mathcal{S}$ is a* universal model *of φ and write $\mathcal{S} \models \varphi$, iff for all runs w in $\mathcal{S}$ we have $w \models \varphi$. We say $\mathcal{S}$ is an* existential model *of φ and write $\mathcal{S} \models_\exists \varphi$, iff there exists a run w in $\mathcal{S}$ such that $w \models \varphi$.*

We can falsify $\mathcal{S} \models \varphi$ by providing a run that violates φ, i.e. if we can obtain some run w with $w \models \neg\varphi$. Instead of using the system $\mathcal{S}$ directly, we will try to find a suitable flat underapproximation $\mathcal{P}$ and a run within it that is also a valid run witnessing $\neg\varphi$ in $\mathcal{S}$. When restricted to fLTL (and as discussed, likely also for lcLTL), it is known from [DHL+17] that for flat structures $\mathcal{S}$ the following holds:

$$\mathcal{S} \not\models \varphi \;\Leftrightarrow\; \mathcal{S} \models_\exists \neg\varphi \;\Leftrightarrow\; \exists n \in \mathbb{N}, \mathcal{P} \in \mathcal{P}(\mathcal{S}) : |\mathcal{P}| \leq n \wedge \exists w \in \mathsf{Runs}(\mathcal{P}) : w \models \neg\varphi$$

This approach to falsify a formula in a counter system is what we call *flat model-checking*. Notice that due to the undecidability of fLTL this method is incomplete for non-flat systems, i.e. there is no guarantee that a falsifying run can be obtained in this way. This means, the approach is underapproximation-based in a twofold sense—on the one hand we use path-schemas, i.e. flat underapproximations, to represent runs in more general counter systems and on the other hand we know that for unrestricted counter systems it does not work in every case due to the undecidability result.

The usage of flat underapproximations that contain multiple loops allows for a compact representation of runs that depend on counter values. Classical bounded

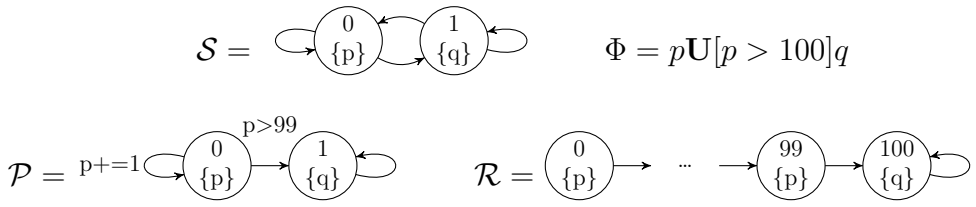


Figure 2.6: A short path schema $\mathcal{P}$ can be used to represent a witnessing run, bounded model-checking can only provide the longer solution $\mathcal{R}$.

model-checking uses just a linear path with one final infinite loop in the end. For classical LTL there is no advantage in the additional capabilities of path schemas, but when LTL is extended with counting operations, bounded model-checking may only be able to find witnesses with an exponentially larger size.

The sketch in figure 2.6 illustrates this restriction of bounded model-checking. In bounded model-checking one would need a sequence of length > 100 to encode the witness for a formula like $p\mathbf{U}[p > 100]q$. While bounded model-checking can only find witnesses for classical LTL where the length of the finite prefix of a run is proportional to the witness size, flat model-checking can find witnesses for LTL with counting such that the length of the finite prefix of the represented run can be exponential in the schema size.

In the next chapter we will be concerned with the search for runs that witness an lcLTL formula using that approach and see how flat model-checking can be implemented.

3 From Flat Model-Checking to SMT

In this chapter we will see how the existential model-checking problem for lcLTL can be solved using flat underapproximations. We will construct a formula that encodes both the underlying counter system and the lcLTL formula for which we want to obtain a run into a satisfiability problem of *quantifier-free Presburger arithmetic* (QPA). The formula will allow for representing a run in an arbitrary path schema of a fixed size that is consistent with the counter system and it is made sure that a solution for this formula represents a valid run in the original counter system which satisfies the given lcLTL formula.

The QPA formula obtained in the following construction can be used as input for some state-of-the-art SMT solver that will do all the heavy lifting. If we get a satisfying valuation for the variables in our formula, the encoded run can be easily reconstructed. The general process is visually illustrated in figure 3.1.

The roadmap for this chapter is as follows. First, we will see what quantifier-free Presburger arithmetic is and what kind of expressions can be stated in it. Then we will construct a formula for the "backbone" of the solution, namely a consistent path schema. Based on this, we will develop constraints that ensure a valid labelling of every state which denotes satisfied subformulas in this state. After a completing the description of the encoding, we will analyze the number of variables used and the growth of the formula.

3.1 Quantifier-free Presburger Arithmetic

Presburger Arithmetic has been briefly mentioned in the last chapter. Especially, it is known that satisfiability of Presburger formulas without quantifiers is in **NP** [BT76]. While solving quantified Presburger formulas remains computationally infeasible,

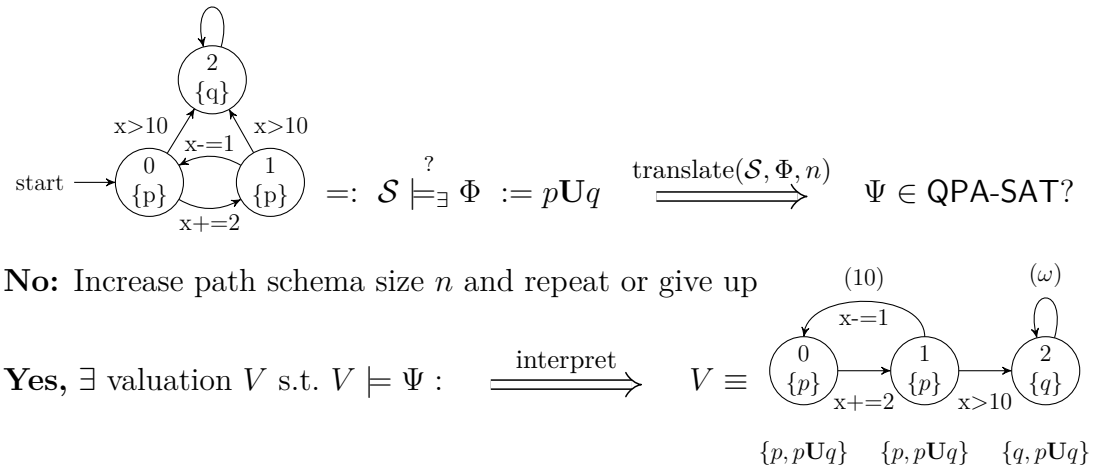


Figure 3.1: Illustration of the flat model-checking procedure. The counter system $\mathcal{S}$ and lcLTL formula Φ are translated into a QPA formula Ψ and an SMT-solver is used to search for a satisfying valuation that represents a run embedded in a flat underapproximation. The obtained run in the example states that the first loop is traversed 10 times to increase counter x sufficiently and moves on to stay in state 2 forever. Each state of the schema is labelled with the satisfied subformulas in that position.

quantifier-free Presburger formulas can often be solved in acceptable time using modern SMT solvers, so restricting ourselves to this fragment to encode the problem is a pragmatic choice.

Formally, we define the minimal syntactic core that can be used to express the translation. If fact, we will use syntactically richer expressions which can always be replaced with an expression using this reduced syntax.

Definition 3.1 (Quantifier-free Presburger Arithmetic)**.** *Let V be a finite set of variables. Quantifier-free Presburger formulas φ and terms τ are defined for constants $c \in \mathbb{N}$ and variables $x \in V$ by the following syntax:*

$$\begin{aligned} \varphi &::= \tau \leq \tau \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \\ \tau &::= c \mid c \cdot x \mid \tau + \tau \end{aligned}$$

Let $\theta : V \to \mathbb{N}$ be a valuation function, $c \in \mathbb{N}, x \in V$, τ_1, τ_2 terms and φ, ψ formulas. Then $\hat{\theta}$ is the substitution of variables defined as:

$$\hat{\theta}(c) := c \qquad \hat{\theta}(c \cdot x) := c \cdot \theta(x) \qquad \hat{\theta}(\tau_2 + \tau_2) := \hat{\theta}(\tau_1) + \hat{\theta}(\tau_2)$$

The satisfaction relation is defined as:

$$\begin{aligned}
\theta \models \tau_1 \leq \tau_2 \quad &:\Leftrightarrow \quad \hat{\theta}(\tau_1) \leq \hat{\theta}(\tau_2) \\
\theta \models \neg\varphi \quad &:\Leftrightarrow \quad \theta \not\models \varphi \\
\theta \models \varphi \wedge \psi \quad &:\Leftrightarrow \quad \theta \models \varphi \text{ and } \theta \models \psi \\
\theta \models \varphi \vee \psi \quad &:\Leftrightarrow \quad \theta \models \varphi \text{ or } \quad \theta \models \psi
\end{aligned}$$

It is easy to see that any finite (or countably infinite) value domain can be encoded in terms of Presburger arithmetic by assigning each element a natural number, so we will liberally use other domains like finite sets or integers for better readability. Also, we will use other comparison operations than $\leq$, for example $\tau_1 > \tau_2$ can be replaced with $\tau_2 \leq \tau_1 - 1$.

Finally, while it was claimed that the resulting formula is quantifier free, the following presentation will liberally use quantification over the finite and fixed number of variables. Every such quantifier can be easily eliminated by replacement of the quantified formula with an expanded instantiation for all variable indices that are quantified over, i.e. $\forall_{i\in[0,2]} : x_i \leq 5$ in fact denotes the natural expansion $x_0 \leq 5 \wedge x_1 \leq 5 \wedge x_2 \leq 5$. We will take special care to keep the required blow-up of the formula size to a minimum by avoiding nested quantification whenever possible.

A hidden usage of existential quantification over finite domains are subformulas of the form $x \in A$, where x is a variable and A some finite set. These are equivalent to $\exists_{a\in A} : x = a$, which is expanded to a disjunction proportional in size to $|A|$ as shown above.

As the complete formula is quite complex, we will develop the translation in a step-by-step manner.

3.2 Basic Structure

Let $\mathcal{S} = (S, s_I, E, \lambda, C, \delta)$ be the counter system that is to be checked, let $n \in \mathbb{N}$ be an arbitrary fixed constant specifying the path schema size, let Φ be an lcLTL formula over the atomic propositions underlying λ and let $\mathsf{sub}(\varphi)$ denote the set of all subformulas of φ. The notation $\mathsf{var} : X$ in the explanations denotes the fact that the variable symbol var which is used in a formula represents values from the domain X.

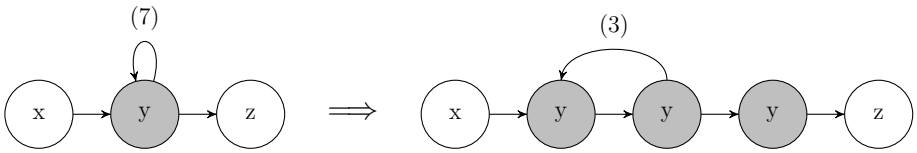


Figure 3.2: Example for the representation of a self-loop of some state y in a run.

We will encode the existence of a run that is representable in a flat underapproximation of $\mathcal{S}$ with a fixed length n. For each position $i \in [0, n-1]$ we first need an associated state in the counter system, $\mathsf{id}_i : S$. Every valid run starts with the initial state. Of course we want only paths that can really be taken in $\mathcal{S}$. So the following formula demands a finite path of length n in $\mathcal{S}$ starting with the initial state:

$$\mathsf{id}_0 = s_I \quad \wedge \quad \forall_{i \in [1,n-1]} : (\mathsf{id}_{i-1}, \mathsf{id}_i) \in E$$

Next we need to represent loops. As the path schema is not fixed, we must be able to encode arbitrary sequences of states and loops. To do this we introduce variables $\mathsf{lType}_i : \{-, \triangleright, +, \triangleleft\}$ for every position i that indicate whether we are outside $(-)$, inside $(+)$ or at the left/right border of a loop $(\triangleright/\triangleleft)$.

Variables $\mathsf{lCount}_i : \mathbb{N}$ indicate a loop count, i.e. how often the corresponding position is visited. Naturally, positions outside of loops shall have a loop count of 1 and strictly greater than 1 for positions inside of loops. Of course we want to have the same loop count for all positions on the same loop.

In this encoding we implicitly assume a minimal loop length of 2 by requiring distinct positions for the first and the last state of a loop. Self-loops of states in $\mathcal{S}$ can easily be represented by assigning two neighboring positions the same state with half the loop count, and for odd loop counts adding another position with the same state which is not a part of the loop (see figure 3.2).

As discussed in the last chapter, runs represented in a path schema necessarily end in an infinite loop. We mark positions that are part of this last loop with a loop count of 0, a value that is not used for regular finite loops. It is to be interpreted as ω is this context.

We want our run to end in the unique infinite loop:

$$\mathsf{lType}_{n-1} = \triangleleft \quad \wedge \quad \mathsf{lCount}_{n-1} = 0$$

We also need to enforce valid relationships between loop count and loop type values:

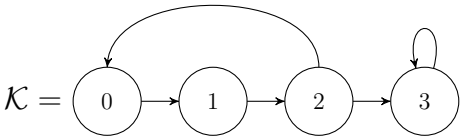
$$\begin{aligned}
\forall_{i\in[0,n-1]}: \quad & \mathsf{lCount}_i \geq 0 \quad \wedge \quad \big(\mathsf{lType}_i = - \;\Leftrightarrow\; \mathsf{lCount}_i = 1\big) \\
\forall_{i\in[0,n-2]}: \quad & \mathsf{lType}_i = \triangleleft \quad \Rightarrow \quad \mathsf{lCount}_i > 1 \\
\forall_{i\in[1,n-1]}: \big(&\mathsf{lType}_i \in \{-, \triangleright\} \quad \Rightarrow \quad \mathsf{lType}_{i-1} \in \{-, \triangleleft\}\big) \\
\wedge \big(&\mathsf{lType}_i \in \{+, \triangleleft\} \quad \Rightarrow \quad \mathsf{lType}_{i-1} \in \{+, \triangleright\} \;\wedge\; \mathsf{lCount}_i = \mathsf{lCount}_{i-1}\big)
\end{aligned}$$

Now we need to relate the loops to valid transitions of the counter system. What is missing for loops is that the edge from the last position of a loop to the first is also a valid edge of our counter system. But we do not know where the loops in our schema are located, where they begin or end, etc. So using what we have defined until now we could require the following:

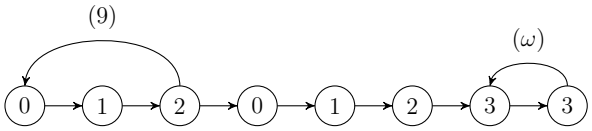
$$\forall_{i\in[1,n-1]}: \mathsf{lType}_i = \triangleleft \;\Rightarrow\; \exists_{0\leq j<i}: \mathsf{lType}_j = \triangleright \;\wedge\; (\mathsf{id}_i, \mathsf{id}_j) \in E \;\wedge\; \forall_{j<k<i}: \mathsf{lType}_k = +$$

But this formula grows quadratically in n when the quantifiers are eliminated (assuming efficient implementation). The problem is the non-locality of the required information, i.e. states associated with the positions. We can solve this problem in a simple manner by introducing more variables under simple constraints that serve the only purpose of transferring the value we are interested in to its destination. We introduce the variables $\mathsf{lStart}_i : S \cup \{\bot\}$, where $\bot$ represents an "undefined" value. At the beginning of a loop we demand that the state at the current position is copied and pushed forward to the end of the loop:

$$\begin{aligned}
\forall_{i\in[0,n-1]}: \big(&\mathsf{lType}_i = \triangleright \quad \Rightarrow \quad \mathsf{lStart}_i = \mathsf{id}_i\big) \\
\wedge \big(&\mathsf{lType}_i = - \quad \Rightarrow \quad \mathsf{lStart}_i = \bot\big) \\
\forall_{i\in[1,n-1]}: &\mathsf{lType}_i \in \{+, \triangleleft\} \quad \Rightarrow \quad \mathsf{lStart}_i = \mathsf{lStart}_{i-1}
\end{aligned}$$



$\mathcal{R} =$



i	0	1	2	3	4	5	6	7
id_i	0	1	2	0	1	2	3	3
lType_i	$\triangleright$	$+$	$\triangleleft$	$-$	$-$	$-$	$\triangleright$	$\triangleleft$
lCount_i	$9 \leftarrow$	$9 \leftarrow$	9	1	1	1	$0 \leftarrow$	0
lStart_i	$0 \rightarrow$	$0 \rightarrow$	0	$\perp$	$\perp$	$\perp$	$3 \rightarrow$	3

Figure 3.3: The encoding of a run in the Kripke structure $\mathcal{K}$. Arrows in the table indicate direction of information propagation.

Now we can check the backloop-edge locally:

$$\forall_{i \in [0,n-1]} : \mathsf{lType}_i = \triangleleft \quad \Rightarrow \quad (\mathsf{id}_i, \mathsf{lStart}_i) \in E$$

Hence, we replaced a quadratic blow-up of the formula with a linear increase in the number of variables and size of the formula. This basic idea of *information propagation* will be used in many, slightly more sophisticated variations later.

What we can represent now are just pseudo-runs, i.e. runs that not necessarily respect all constraints of the system, as we ignore the counters and guards of $\mathcal{S}$ for now and pretend that it is a simple Kripke structure. For an example see figure 3.3. The missing pieces will be introduced soon.

3.3 Labelling Positions with Subformulas

We want to obtain runs that satisfy a given formula Φ. This will be accomplished by a bottom-up inductive labelling scheme, not unlike the canonical labelling algorithm used for CTL [BCM+92]. First, each position is labelled with the atomic propositions that are present in the state associated with it. Then we label it with each subformula for which the constituent subformulas are already labelled according to its meaning.

We use boolean variables $\mathsf{lbl}_i^\varphi : \{\top, \bot\}$ for each $\varphi \in \mathsf{sub}(\Phi)$ to encode the subsets of subformulas that are labelled at position i. A set of subformulas can be thought to be encoded canonically as a characteristic bitstring where each position corresponds to a subformula. For convenience we will use lbl_i with its set semantics as a subset of $\mathsf{sub}(\Phi)$, so $\varphi \in \mathsf{lbl}_i$ is to be read in this context as $\mathsf{lbl}_i^\varphi = \top$.

A formula holds for a run if it holds in the first state of that run, so we require:

$$\Phi \in \mathsf{lbl}_0$$

The non-temporal operators are handled easily in the following way:

$$\begin{aligned}
&\forall_{i \in [0,n-1]} : \\
&\forall_{\xi \in AP} : \xi \in \mathsf{lbl}_i \quad &\Leftrightarrow \quad & \xi \in \lambda(\mathsf{id}_i) \\
\forall_{\neg\varphi = \xi \in \mathsf{sub}(\Phi)} &: \xi \in \mathsf{lbl}_i \quad &\Leftrightarrow \quad & \varphi \notin \mathsf{lbl}_i \\
\forall_{\varphi \wedge \psi = \xi \in \mathsf{sub}(\Phi)} &: \xi \in \mathsf{lbl}_i \quad &\Leftrightarrow \quad & \varphi \in \mathsf{lbl}_i \wedge \psi \in \mathsf{lbl}_i \\
\forall_{\varphi \vee \psi = \xi \in \mathsf{sub}(\Phi)} &: \xi \in \mathsf{lbl}_i \quad &\Leftrightarrow \quad & \varphi \in \mathsf{lbl}_i \vee \psi \in \mathsf{lbl}_i
\end{aligned}$$

For $\mathbf{X}\varphi$ we need to check two cases, as there are always two possible next states. One is the next state in the sequence and the other is the target of a possible backloop-edge, if we are in the last state of a loop. At least one successor is always present, as all states except the last one have a successor in the sequence and the last state necessarily closes a loop. To have a consistent labelling which is abstracted from the different times the state is visited, we want that φ holds in all successor states that are present.

The backloop-case of the labelling by $\mathbf{X}\varphi$ could be expressed using forward propagation of the label for φ from the beginning to the end of the loop, using distinct sequences of variables for each $\mathbf{X}\varphi$ subformula present in Φ with a similar scheme like the one used with lStart to verify the validity of backloop-edges.

However, we will solve this in a different way, which may look unnecessarily complex, but the necessity and utility of the approach will become clear soon. We introduce the variables $\mathsf{lCtr}_i : \mathbb{N}$ to calculate the length of every loop. As loops cannot overlap,

we can reuse these variables for all loops in the path schema.

$$\begin{aligned}\forall_{i\in[0,n-1]} : \big(\mathsf{lType}_i = - \quad &\Leftrightarrow \quad \mathsf{lCtr}_i = 0\big)\\ \wedge\ \big(\mathsf{lType}_i = \triangleright \quad &\Leftrightarrow \quad \mathsf{lCtr}_i = 1\big)\\ \forall_{i\in[1,n-1]} : \mathsf{lType}_i \in \{+, \triangleleft\} \quad &\Leftrightarrow \quad \mathsf{lCtr}_i = \mathsf{lCtr}_{i-1} + 1\end{aligned}$$

So basically, we set up a counter that is held together by the increment relationship between neighboring variables at positions within a loop. The last position r of a loop has the total length stored in lCtr_r. Now we would like to check the backloop-target on the left at the right border of the loop by checking $\mathsf{lbl}_{r-\mathsf{lCtr}_r+1}$. We cannot do this, though, as we would need to know the values before constructing the formula. What we can do is enumerating all possible cases, as the length of a loop in our schema is restricted between 2 and n. So we could do the following:

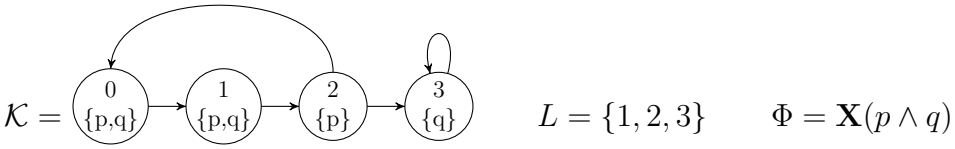
$$\begin{aligned}\forall_{\mathbf{X}\varphi=\xi\in\mathsf{sub}(\Phi)} : \forall_{i\in[0,n-1]} : \xi \in \mathsf{lbl}_i \ \Leftrightarrow\ \big((\varphi \in \mathsf{lbl}_{i+1} \vee i = n-1)\\ \wedge\ (\mathsf{lType}_i = \triangleleft \Rightarrow \exists_{2\le l<i} : \mathsf{lCtr}_i = l \wedge \varphi \in \mathsf{lbl}_{i-l+1})\big)\end{aligned}$$

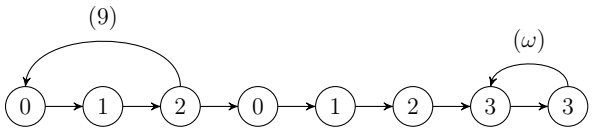
But as the vigilant reader will observe, this again would impose a quadratic blowup of the formula, so by itself the usage of the lCtr_i variables is not an acceptable solution. To mitigate this we need to use additional information about our system $\mathcal{S}$. A schema that is consistent with our system of course cannot have completely arbitrary loop lengths. Especially, if the counter system is flat, any consistent loop in our schema must have a length of a multiple of some length of a simple loop in $\mathcal{S}$.

Let $L = \{l \mid \exists w : w \text{ is simple loop in } \mathcal{S} \text{ and } |w| = l\} \cup \{2\}$ be the simple loop lengths in $\mathcal{S}$ (we ensure $2 \in L$ to represent self-loops). Now, without loss of generality, we can restrict the set of possible loop lengths to only those lengths:

$$\begin{aligned}\forall_{\mathbf{X}\varphi=\xi\in\mathsf{sub}(\Phi)} : \forall_{i\in[0,n-1]} : \xi \in \mathsf{lbl}_i \ \Leftrightarrow\ \big((\varphi \in \mathsf{lbl}_{i+1} \vee i = n-1)\\ \wedge\ (\mathsf{lType}_i = \triangleleft \Rightarrow \exists_{l\in L, l\le i+1} : \mathsf{lCtr}_i = l \wedge \varphi \in \mathsf{lbl}_{i-l+1})\big)\end{aligned}$$

So our formula grows only by a factor of $|L|$, which is dependent only on the system $\mathcal{S}$ and remains constant for varying schema sizes n. We will return to the $\mathbf{X}$ operator one more time, but this is an acceptable solution for now. An example of the current encoding can be seen in figure 3.4.





i	0	1	2	3	4	5	6	7
id_i	0	1	2	0	1	2	3	3
lType_i	$\triangleright$	$+$	$\triangleleft$	$-$	$-$	$-$	$\triangleright$	$\triangleleft$
lCount_i	9 $\leftarrow$	9 $\leftarrow$	9	1	1	1	0 $\leftarrow$	0
lStart_i	0 $\rightarrow$	0 $\rightarrow$	0	$\bot$	$\bot$	$\bot$	2 $\rightarrow$	2
lCtr_i	1 $\rightarrow$	2 $\rightarrow$	3	0	0	0	1 $\rightarrow$	2
lbl_i^p	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$
lbl_i^q	$\top$	$\top$	$\bot$	$\top$	$\top$	$\bot$	$\top$	$\top$
$\mathsf{lbl}_i^{p \wedge q}$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\bot$
$\mathsf{lbl}_i^{\mathbf{X}(p \wedge q) = \Phi}$	$\top$	$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$

Figure 3.4: The encoding of a run in the Kripke structure $\mathcal{K}$ satisfying Φ. lbl_2^{Φ} is witnessed by $\mathsf{lbl}_3^{p \wedge q}$ and also $\mathsf{lbl}_{2-\mathsf{lCtr}_2+1}^{p \wedge q} = \mathsf{lbl}_0^{p \wedge q}$.

3.4 The Constrained U Operator

3.4.1 Difficulties and a Partial Solution

Now we want to express the semantics of the $\varphi \mathbf{U}[\ldots]\psi$ operator. If there were no linear counting constraints, we could handle the $\mathbf{U}$ in some simple fashion like the rejected solution for $\mathbf{X}$ and would not even necessarily need the loop length counter construction at all.

The difficulties arise when constraints are in place. The path schema abstracts from the individual visits of the states in some position, while the values of the counters can differ for each visit. If the labelling depends on counting, as it is the case with linear constraints here, we cannot label positions inside of loops consistently, as in different loop iterations the counter constraints may or may not be satisfied. We

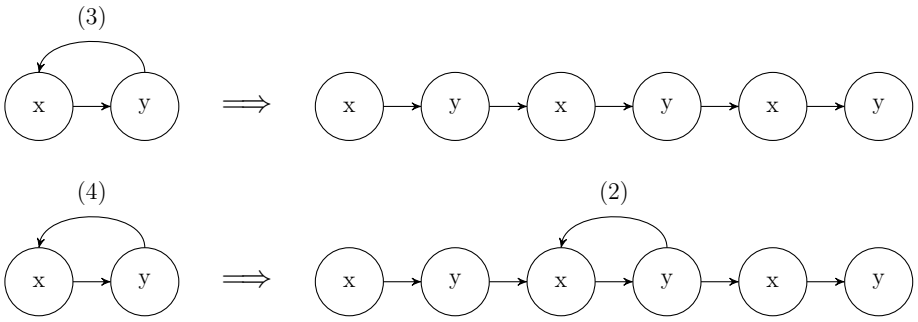


Figure 3.5: Compact representation works only for ≥ 4 loop iterations

cannot even assign the counters a really meaningful value inside of loops, as every loop position carries at least two different values.

We can solve the main issue of undetermined constraint satisfaction by requiring *constraint invariance* on the loops in our schema—we demand that if a constraint is satisfied in the first loop iteration, it is also satisfied in the last iteration and every iteration in between, and vice versa. We make sure that this is the case by demanding a *loop unrolling* on each side (i.e. a copy of the loop state sequence on the left and on the right of the loop) and then requiring that the labelling for all subformulas is equal in all three copies of a state.

Every loop that is repeated more than three times can still be represented in this form, loops with three or less repetitions must be unrolled completely. So we still can represent every loop, only losing the ability to represent loops with less than four repetitions compactly (see figure 3.5).

Notice that with these restrictions we get a lower bound on the schema size of $n = 4$, as we always need at least the infinite loop (consisting of at least 2 positions) and one left unrolling of it.

We cannot use a simple propagation scheme to compare the labels at different positions as the loop length and location is variable and we need information for every subformula and position inside of a loop, so that we would need a quadratic number of additional variables, rendering this a bad trade-off against the blow-up due to quantification. But we can again use our ICtr_i variables in combination with the set L to enforce the unrollings without such a big penalty.

As we need the loop length as an offset at each position of a loop, we need to propagate the obtained final lCtr value for the loop backwards from right to left. To do this we use new $\mathsf{lLen}_i : \mathbb{N}$ variables:

$$\begin{aligned} \forall_{i\in[0,n-1]} : \big(\mathsf{lType}_i = - \quad &\Leftrightarrow \quad \mathsf{lLen}_i = 0\big) \\ \wedge \big(\mathsf{lType}_i = \triangleleft \quad &\Leftrightarrow \quad \mathsf{lLen}_i = \mathsf{lCtr}_i\big) \\ \forall_{i\in[1,n-1]} : \mathsf{lType}_i \in \{+, \triangleleft\} \quad &\Leftrightarrow \quad \mathsf{lLen}_{i-1} = \mathsf{lLen}_i \end{aligned}$$

These can now be used to require the left and right unrolling for the finite loops and just a left unrolling for the last loop and make sure that they also have the same labels:

$$\begin{gathered} \forall_{i\in[0,n-1]} : \big(\mathsf{lType}_i \neq - \Rightarrow \exists_{l\in L, l\leq i} : \\ \mathsf{lLen}_i = l \wedge \mathsf{lType}_{i-l} = - \wedge \mathsf{id}_i = \mathsf{id}_{i-l} \wedge \mathsf{lbl}_i = \mathsf{lbl}_{i-l}\big) \\ \wedge \big(\mathsf{lType}_i \neq - \wedge \mathsf{lCount}_i \neq 0 \Rightarrow \exists_{l\in L, l\leq n-i-1} : \\ \mathsf{lLen}_i = l \wedge \mathsf{lType}_{i+l} = - \wedge \mathsf{id}_i = \mathsf{id}_{i+l} \wedge \mathsf{lbl}_i = \mathsf{lbl}_{i+l}\big) \end{gathered}$$

Now for $\varphi\mathbf{U}[\ldots]\psi$ to hold, first the conditions for regular $\mathbf{U}$ and second the additional constraint must be satisfied. For the first part it suffices to apply the usual known equivalence $\varphi\mathbf{U}\psi = \psi \vee (\varphi \wedge \mathbf{X}(\varphi\mathbf{U}\psi))$ and using the fact that we can always find a ψ in the right unrolling we enforced, treating finite loops and linear segments in a uniform way.

For the last position we cannot just look on the right, so we have no choice but to look within the infinite loop in positions on the left, but we know the maximum size of a simple loop already and can restrict the positions we have to look at. So we have only a constant formula overhead here. Hence, the usual $\mathbf{U}$ operator can be expressed with $\hat{l} = \min(n, \max(L))$:

$$\begin{gathered} \forall \varphi\mathbf{U}[\ldots]\psi = \xi \in \mathsf{sub}(\Phi) : \\ \Big(\forall_{i\in[0,n-2]} : \varphi\mathbf{U}\psi \in \mathsf{lbl}_i \;\Leftrightarrow\; \psi \in \mathsf{lbl}_i \vee \big(\varphi \in \mathsf{lbl}_i \wedge \xi \in \mathsf{lbl}_{i+1}\big)\Big) \\ \wedge \Big(\varphi\mathbf{U}\psi \in \mathsf{lbl}_{n-1} \;\Leftrightarrow\; \psi \in \mathsf{lbl}_{n-1} \vee \big(\varphi \in \mathsf{lbl}_{n-1} \wedge \exists_{j\in[1,\hat{l}]} : \mathsf{lCount}_{n-j} = 0 \wedge \psi \in \mathsf{lbl}_{n-j}\big)\Big) \end{gathered}$$

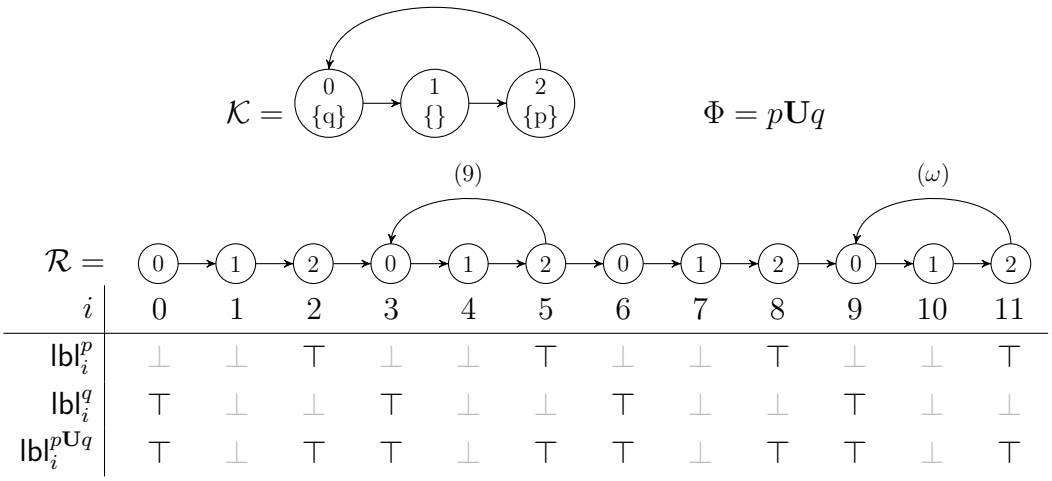


Figure 3.6: Example of a run which is correctly labelled with a **U**-subformula without constraints on finite and infinite loops

Notice that here for each constrained until operator we add a label for an unconstrained **U** for convenience. We will use this to construct a formula for the $\mathbf{U}[\ldots]$ operator, i.e. with the additional constraints. Stated more formally, we assume that $\varphi\mathbf{U}[\ldots]\psi \in \mathsf{sub}(\Phi) \Rightarrow \varphi\mathbf{U}\psi \in \mathsf{sub}(\Phi)$.

Let us consider the most interesting constellation and discuss an example labelling in detail for both the finite and infinite loop case. Observe the labelling of the run illustrated in figure 3.6, where the chain of p is interrupted in the linear part of the loop. First note that positions $0-2$ are the left unrolling of the loop at positions $3-5$ and positions $6-8$ are its right unrolling and also a left unrolling of the last loop located at $9-11$. Note that we demanded the labels of the unrolled loop states to be the same as inside of the loop, so positions $0, 3, 6, 9$, positions $1, 4, 7, 10$, etc. are labelled equally. The labels for p and q are witnessed directly by the state at that position.

The label for $p\mathbf{U}q$ at positions $0, 3, 6, 9$ is witnessed directly by the label q in the same positions. $p\mathbf{U}q$ is also witnessed for positions $2, 5, 8$ by the label p in these positions and by the label $p\mathbf{U}q$ in their corresponding successor positions $3, 6, 9$ and finally $p\mathbf{U}q$ is witnessed in position 11 by the label p in position 11 and q in position 9, which is also part of the loop and the fact that we can reach this q from position 11 is guaranteed by the fact that position 8 is the same state in the left unrolling of the last loop and is already correctly labelled. All finite loops are labelled correctly

without additional effort due to the copy of the witness on the right propagating the label to the left (if there is a witness).

3.4.2 Evaluation of Constraints

Next we need to label $\mathbf{U}$ with linear constraints correctly. In [DHL+17, Definition 6] a syntactic consistency criterion is defined for the labelling of $\varphi \mathbf{U}^{\frac{n}{m}} \psi$ formulas. Roughly, the basic idea described there is that a counter can be started at every position of the path schema and is updated according to the given fraction, i.e. decreased by n in every position and additionally increased by m at positions where φ holds. This ensures that the counter is greater than or equal to zero iff φ has been observed sufficiently often since the starting time of the counter. For the correct labelling of $\mathbf{U}[\ldots]$ we can generalize this idea, while removing some of technicalities that are required in the consistency criterion, because we are concerned with a concrete run (the criterion in [DHL+17] does more—it ensures that every run in the constructed schema is consistent with the labelling).

We need a single counter for each $\mathbf{U}[\ldots]$ subformula to be able to evaluate the constraint. It will keep track of the effect of each position on the total balance of a constraint. So each $\xi = \varphi \mathbf{U}[\ldots]\psi \in \mathsf{sub}(\Phi)$ gets a set of counter variables $\mathsf{udCtrs}_i^\xi : \mathbb{Z}$. These will be set up in such a way that the left unrolling contains values for the first iteration and the right unrolling contains values for the last iteration of the loop.

Now the question arises, how do we obtain the counter values for the position at the beginning of the right unrolling? The first naïve idea would be to use a set of variables that adds up the updates within loops and add the resulting number, multiplied with the number of iterations, to the counter value from before the loop. This is not possible directly though, as this would require a multiplication of two variables—the loop count and the variable containing the accumulated effect of the loop. Presburger arithmetic does not allow for multiplication of variables, so a different solution is required.

As discussed, counter values within loops are meaningless, but we can still use the variables inside loop positions to accumulate the total effect of the loop on the

counter by incrementing according to the constant coefficients and the loop count:

$$\forall \varphi \mathbf{U}[\sum_{j=0}^{m} k_j \zeta_j \oplus k]\psi = \xi \in \mathsf{sub}(\Phi) : \mathsf{udCtrs}_0^\xi = 0$$
$$\wedge\ \forall_{i \in [1,n-1]} : \mathsf{udCtrs}_i^\xi = \mathsf{udCtrs}_{i-1}^\xi + \sum_{\{j \mid \zeta_j \in \mathsf{lbl}_{i-1}\}} k_j \cdot \mathsf{lCount}_{i-1}$$

We can safely do this, because by construction every visit of a position in a loop iteration is equivalent (due to the same labels), so the update can be added for all iterations in one step. Thus, the multiplication of variables can be avoided by exploiting distributivity—instead of multiplying the total result, which is a variable, we can already multiply the update of each position individually, which is a constant. The resulting value that carries over to the right unrolling is the correct value obtained after executing the previous loop iterations. Also notice that the value for position i stores the value at the moment it was entered, the value after travesing an outgoing edge is stored at position $i + 1$.

The construction gives us counter values relative to the first position of the run, but we are interested in the counter values relative to any starting point. This can be easily achieved by just subtracting the value that has been accumulated up to the starting position from the constraint, so rather than checking the original constraint of some $\varphi \mathbf{U}[\ldots]\psi$, for position i we check $\varphi \mathbf{U} \psi$ without the constraint as shown above and also require that there is some future position $l > i$ such that:

$$\sum_{j=0}^{m} k_j \cdot \#_{[i,l-1]}(\eta_i) = \mathsf{udCtrs}_l - \mathsf{udCtrs}_i \oplus k \quad \Leftrightarrow \quad \mathsf{udCtrs}_l \oplus k + \mathsf{udCtrs}_i$$

Here the question arises how to obtain a valid future position l and corresponding counter value for each i? The answer is that we do not need the position, but just need a counter value from some valid future witness position. This is another information non-locality problem which we can solve with propagation. For $\varphi \mathbf{U}[\ldots]\psi$, given the counter values and knowing the kind of inequality we have in the constraint, we can propagate the best value that we have seen at some ψ position. We can push it to the left as long as all the positions are labelled with φ, i.e. as long as the propagated optimum value (maximum for $\oplus \in \{>, \geq\}$ and minimum for $\oplus \in \{<, \leq\}$) comes

from a ψ position that is a valid witness for $\varphi \mathbf{U} \psi$ at the current position. In every position of the chain where also ψ holds we push on the better value. When the φ-chain is broken, we need to reset to some worst possible value that does not satisfy the constraint (semantically to $\pm\infty$, depending on the operator $\oplus$).

There is one pitfall here which is easy to overlook—positions inside of loops have semantically meaningless intermediate values in the $\mathsf{udCtrs}_i^\varphi$ variables which we must not consider in the described procedure. On the other hand, we do need some valid values witnessing the constraint satisfaction for positions in the left loop unrollings and we cannot transfer the values from the corresponding right unrolling safely, as the chain might be broken in between (like in figure 3.6).

Therefore, we also calculate witness counter values $\mathsf{uwCtrs}_\mathsf{i}^\xi : \mathbb{Z}$ within loops which store the value of the first loop iteration (second iteration, when starting to count from the left unrolling). When leaving a loop, these are synchronized with the udCtrs_i^ξ values and are always equal to those outside of loops:

$$\begin{aligned} \forall \varphi \mathbf{U} [\sum_{j=0}^{m} k_j \zeta_j \oplus k] \psi = \xi \in \mathsf{sub}(\Phi) : \mathsf{uwCtrs}_0^\xi = 0 \\ \wedge \forall_{i \in [1,n-1]} : \mathsf{ite}\big(\mathsf{lType}_{i-1} = \triangleleft,\ \mathsf{uwCtrs}_i^\xi = \mathsf{udCtrs}_i^\xi, \\ \mathsf{uwCtrs}_i^\xi = \mathsf{uwCtrs}_{i-1}^\xi + \sum_{\{j \mid \zeta_j \in \mathsf{lbl}_{i-1}\}} k_j \big) \end{aligned}$$

Here $\mathsf{ite}(x, y, z) := (x \wedge y) \vee (\neg x \wedge z)$ denotes a logical if-then-else construction.

For the reset of the counter we need some unreachable value. We can obtain one by calculating the length of the run (up to the infinite loop) and multiplying it with a value that is bigger than the biggest increment possible at some position. To calculate the total run length we introduce the variables $\mathsf{steps}_i : \mathbb{N}$ and accumulate the loop counts, i.e. number of visits of each position:

$$\mathsf{steps}_0 = \mathsf{lCount}_0 \wedge \forall_{i \in [1,n-1]} : \mathsf{steps}_i = \mathsf{steps}_{i-1} + \mathsf{lCount}_i$$

For some constraint of a formula ξ with coefficients k_j let $\hat{\infty}^\xi := \mathsf{steps}_{n-1} \cdot |\mathsf{sub}(\Phi)| \cdot \max\{\mathsf{abs}(k_j)\} + 1$. It is easy to see that this positive value is unreachable by the

constraint counter of formula ξ. Now let $\mathsf{opt}^\xi := \max, \bot^\xi := -\hat{\infty}^\xi$ for $\oplus \in \{>, \geq\}$ and $\mathsf{opt}^\xi := \min, \bot^\xi := \hat{\infty}^\xi$ for $\oplus \in \{<, \leq\}$.

Using the $\mathsf{uwCtrs}_i^\varphi$ and new variables $\mathsf{uBest}_i^\varphi : \mathbb{Z}$ we can implement the suffix optimum backward propagation described above:

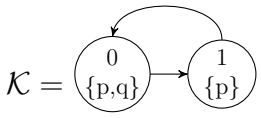
$$\begin{gathered}
\forall \varphi \mathbf{U}[\sum_{j=0}^{m} k_j \zeta_j \oplus k]\psi = \xi \in \mathsf{sub}(\Phi) : \\
\mathsf{ite}\big(\psi \in \mathsf{lbl}_{n-1},\ \mathsf{uBest}_{n-1}^\xi = \mathsf{uwCtrs}_{n-1}^\xi,\ \mathsf{uBest}_{n-1}^\xi = \bot^\xi\big) \\
\wedge\ \forall_{i \in [0,n-2]} : \mathsf{ite}\big(i > 0 \wedge \varphi \in \mathsf{lbl}_{i-1}, \\
\mathsf{ite}(\psi \in \mathsf{lbl}_i^\xi,\ \mathsf{uBest}_i^\xi = \mathsf{opt}^\xi(\mathsf{uBest}_{i+1}^\xi, \mathsf{uwCtrs}_i^\xi),\ \mathsf{uBest}_i^\xi = \mathsf{uBest}_{i+1}^\xi), \\
\mathsf{uBest}_i^\xi = \bot^\xi\big)
\end{gathered}$$

This can be used to check the constraints outside of loops and if they are satisfied in both unrollings, then the constraint has been satisfied for the whole range of counter values, implying that the identical labelling inside of finite loops is justified and correct. Hence we put the pieces together to obtain a formula to label $\varphi \mathbf{U}[\ldots]\psi$:

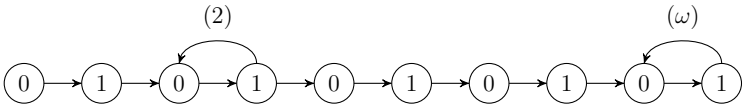
$$\begin{gathered}
\forall \varphi \mathbf{U}[\sum_{j=0}^{m} k_j \zeta_j \oplus k]\psi = \xi \in \mathsf{sub}(\Phi) : \forall_{i \in [0,n-1]} : \mathsf{lType}_i = - \Rightarrow \\
\Big(\xi \in \mathsf{lbl}_i \ \Leftrightarrow\ \varphi \mathbf{U} \psi \in \mathsf{lbl}_i \wedge \Big(\big(\psi \in \mathsf{lbl}_i\ \wedge 0 \oplus k\big) \\
\vee\ \big(\varphi \in \mathsf{lbl}_i\ \wedge\ \mathsf{uBest}_{i+1}^\xi \oplus k + \mathsf{uwCtrs}_i^\xi\big)\Big)\Big)
\end{gathered}$$

The check of $0 \oplus k$ is a simplification for ψ-positions. As these are not included in the scope spanned by the φ-sequence and the scope of $\varphi \mathbf{U}[\ldots]\psi$ when starting at ψ is empty, the constraint counter there is always zero. Hence, we can just check whether the constraint holds for zero directly.

Now consider the situation depicted in figure 3.7. The described reasoning and the proposed formula works well for the finite loop, as we get a witness value that is high enough propagated from the right. Our labelling of Φ for positions $6 - 9$ is wrong, though.



$$\Phi = p\mathbf{U}[\mathsf{true} > 2]q$$



i	0	1	2	3	4	5	6	7	8	9
lbl_i^p	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
lbl_i^q	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$
$\mathsf{lbl}_i^{p\mathbf{U}q}$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
lbl_i^{Φ}	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$(\bot)$	$(\bot)$	$(\bot)$	$(\bot)$
udCtrs_i^{Φ}	0	1	2	4	6	7	8	9	10	10
uwCtrs_i^{Φ}	0	1	2	3	6	7	8	9	10	10
uBest_i^{Φ}	$\bot^{\Phi}$	10	10	10	10	10	10	10	10	$\bot^{\Phi}$

Figure 3.7: Example of a run where labelling fails for the infinite loop.

Φ is satisfied in these positions too, because the infinite loop has no interruption of the p sequence and the loop strictly increases the constraint counter (let us call such loops *good loops*). This means, that after some finite number of iterations, the constraint is satisfied from the point of view of every starting point, even though we cannot obtain some specific value that witnesses this fact.

If the infinite loop was bad for the constraint, there would be no witness for any iteration, so not labelling Φ would have been correct. If the loop would be neutral, i.e. have no effect on the constraint counter, then the constraint would be either always true or always false and especially could be witnessed by the backpropagation of the same value from the uwCtrs in the infinite loop. Finally, if the sequence of p would have been interrupted in the loop, each iteration could be considered individually anyway, as an infinite accumulation of the counter could not happen. Hence, the current formula is not sufficient for the infinite loop if the following conditions hold:

1. $\varphi\mathbf{U}\psi$ holds and the φ-sequence is not interrupted in the loop
2. a single loop unrolling cannot generate a sufficient witnessing counter value

If we would allow to take multiple unrollings of a loop in the system as a single loop in the schema, we could circumvent this problem, but this would bloat up the required schema size to find runs for formulas that include constraints with big coefficients. Additionally the formula size would increase due to many more possible loop lengths, so this is not an acceptable solution.

So we have no other choice but to add an additional satisfaction condition that we must check in the left unrolling of the infinite loop to ensure correct labelling of the loop—$\varphi \mathbf{U}[\ldots]\psi$ is satisfied, if φ holds at every position inside of the last loop, the loop has a ψ and is a good loop for the constraint.

First, we need to identify the problematic positions of the left unrolling just before the infinite loop. So we add boolean variables lLast_i and constrain them accordingly:

$$\forall_{i\in[0,n-1]} : \mathsf{lLast}_i \Leftrightarrow \exists_{l\in L, l\leq n-i-1} : \mathsf{lLen}_{i+l} = l \wedge \mathsf{lCount}_{i+l} = 0$$

Then we want to check whether the discussed case can apply. For each $\xi = \varphi \mathbf{U}[\ldots]\psi$ we introduce a boolean variable allPhi_ξ that determines whether the sequence of φ is not interrupted for this formula ξ:

$$\forall_{\xi=\varphi\mathbf{U}[\ldots]\psi} : \mathsf{allPhi}_\xi \Leftrightarrow \forall_{i\in[n-\hat{l},n-1]} : \mathsf{lCount}_i = 0 \Rightarrow \varphi \in \mathsf{lbl}_i$$

If this is the case, we want to know whether the loop is strictly good for the constraint, which we do by accumulating the effects of the constraint over the course of the last loop, thereby obtaining a loop delta, i.e. the total effect of one loop iteration on the constraint counter. Again we need a set of variables $\mathsf{ulDelta}_i^\xi : \mathbb{Z}$ for each $\mathbf{U}[\ldots]$ subformula. We do not need intermediate values for every position of the schema, so we allocate only as many as the largest possible loop in our counting system. Hence we require:

$$\forall \varphi \mathbf{U}[\sum_{j=0}^{m} k_j\zeta_j \oplus k]\psi = \xi \in \mathsf{sub}(\Phi) : \forall_{i\in[0,\hat{l}-1]} :$$
$$\mathsf{lType}_{n-\hat{l}+i} = - \Rightarrow \mathsf{ulDelta}_i^\xi = 0 \wedge \big(\mathsf{lCount}_{n-\hat{l}+i} = 0 \Rightarrow$$
$$\mathsf{ite}(\mathsf{lType}_{n-\hat{l}+i} = \triangleright, \mathsf{ulDelta}_i^\xi = \sum_{\{j|\zeta_j\in\mathsf{lbl}_{n-\hat{l}+i}\}} k_j,$$
$$\mathsf{ulDelta}_i^\xi = \mathsf{ulDelta}_{i-1}^\xi + \sum_{\{j|\zeta_j\in\mathsf{lbl}_{n-\hat{l}+i}\}} k_j)\big)$$

The complete formula for $\mathbf{U}[\ldots]$ labelling can be expressed now like this:

$$\forall \varphi \mathbf{U}[\sum_{j=0}^{m} k_j \zeta_j \oplus k] \psi = \xi \in \mathsf{sub}(\Phi) : \forall_{i \in [0,n-2]} : \mathsf{lType}_i = - \Rightarrow$$
$$\Big(\xi \in \mathsf{lbl}_i \;\Leftrightarrow\; \varphi \mathbf{U} \psi \in \mathsf{lbl}_i \wedge \Big(\big(\psi \in \mathsf{lbl}_i \;\wedge 0 \oplus k \big)$$
$$\vee \; \big(\varphi \in \mathsf{lbl}_i \;\wedge\; (\mathsf{uBest}^{\xi}_{i+1} \oplus k + \mathsf{uwCtrs}^{\xi}_{i}) \vee (\mathsf{lLast}_i \wedge \mathsf{allPhi}_{\xi} \wedge \mathsf{ulDelta}^{\xi}_{\hat{l}-1} \hat{\oplus}\; 0) \big) \Big) \Big)$$

where $\hat{\oplus} := >$ for $\oplus \in \{\geq, >\}$ and $\hat{\oplus} := <$ for $\oplus \in \{\leq, <\}$.

Having introduced the enforced loop unrollings to solve $\mathbf{U}[\ldots]$ labelling, we can simplify $\mathbf{X}$ labelling. The special case for the backloop-target can be removed now. Thanks to the right unrolling, the following state must be equal to the first state of the loop and for the last position demanding that states in loop unrollings are labelled equally already ensures that the copy in the left unrolling serves as a witness. Therefore labelling of $\mathbf{X}$ reduces to:

$$\forall_{\mathbf{X}\varphi = \xi \in \mathsf{sub}(\Phi)} : \forall_{i \in [0,n-2]} : \xi \in \mathsf{lbl}_i \;\Leftrightarrow\; \varphi \in \mathsf{lbl}_{i+1}$$

3.5 Counter System Guards

With the experience of handling the $\mathbf{U}[\ldots]$-counters, we can now easily add the checking of guards over the counters that are present in $\mathcal{S}$ by adding a sequence of variables $\mathsf{gVals}^c_i : \mathbb{Z}$ for each counter $c \in C$. Again, it suffices to check just positions in the unrollings of each finite loop, because the accumulated update of each loop is constant for each iteration and the constraints are linear and therefore we do not leave the convex region imposed by the conjunction of the edge constraints (see sketch in figure 3.8).

We have a similar problem as above with the infinite loop, though. Checking the left unrolling guarantees that the edge guards are satisfied in the first loop execution, but we need to ensure that they will be satisfied in each of the infinite loop iterations. So we need to compute loop deltas for each guard of each edge. Inside the last loop we must check for every edge that the loop is good or neutral for every single constraint of that edge. This ensures that the counters, if changed at all, are increased in an

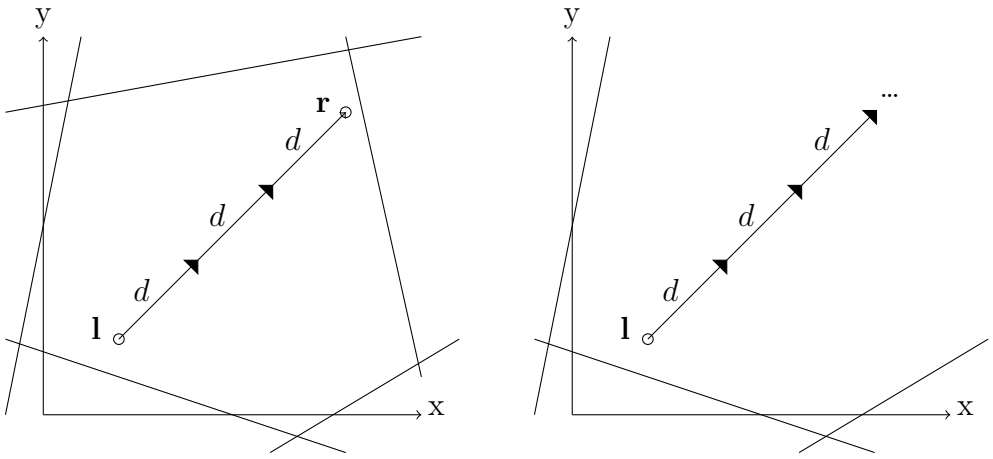


Figure 3.8: Left: Sketch illustrating finite loop case. A conjunction of constraints over 2 counters describes a convex polytope in 2-dimensional space, like in linear programming. $\mathbf{l}, \mathbf{r}$ indicate the counter values at a state copy in the left and right unrolling, The vector d indicates the loop delta, i.e. total update of one loop iteration. Right: Unbounded case necessary for infinite loop.

unbounded direction, i.e. no guard in the conjunction will ever become not satisfied by iterating the loop. To check this, we use variables $\mathsf{glDelta}_i^g : \mathbb{Z}$ for every guard g at edges of $\mathcal{S}$ and do basically the same as for the $\mathsf{ulDelta}_i^\xi$ variables:

$$\forall (k_c, \oplus, k) = g \in \bigcup_{e \in E} \delta_g(e) : \forall_{i \in [0, \hat{l}-1]} :$$
$$\mathsf{ite}\Big(\mathsf{lCount}_{n-\hat{l}+i} > 0,\ \mathsf{glDelta}_i^g = 0,$$
$$\mathsf{ite}\big(\mathsf{lType}_{n-\hat{l}+i} = \triangleright, \mathsf{glDelta}_i^g = \sum_{c \in C} k_c(c) \cdot \delta_{u(c)}(\mathsf{id}_{n-\hat{l}+i-1}, \mathsf{id}_{n-\hat{l}+i}),$$
$$\mathsf{glDelta}_i^g = \mathsf{glDelta}_{i-1}^g + \sum_{c \in C} k_c(c) \cdot \delta_{u(c)}(\mathsf{id}_{n-\hat{l}+i-1}, \mathsf{id}_{n-\hat{l}+i})\big)\Big)$$

Now, using $\check{\oplus} := \geq$ for $\oplus \in \{\geq, >\}$ and $\check{\oplus} := \leq$ for $\oplus \in \{\leq, <\}$, we can express the conditions for consistency with counter system updates and guards, completing our encoding:

$$\forall_{c \in C} : \mathsf{gCtrs}_0^c = 0 \ \wedge\ \forall_{i \in [1, n-1]} : \forall_{c \in C} : \mathsf{gCtrs}_i^c = \mathsf{gCtrs}_{i-1}^c + \mathsf{lCount}_{i-1} \cdot \delta_{u(c)}(\mathsf{id}_{i-1}, \mathsf{id}_i)$$
$$\wedge\ (\mathsf{lType}_i = - \Rightarrow \bigwedge_{(k_c, \oplus, k) = g \in \delta_g(id_{i-1}, id_i)} \sum_{c \in C} k_c(c) \cdot \mathsf{gVals}_i^c \oplus k)$$
$$\wedge\ (\mathsf{lCount}_i = 0 \Rightarrow \bigwedge_{g \in \delta_g(id_{i-1}, id_i)} \mathsf{glDelta}_{\hat{l}-1}^g \check{\oplus}\ 0)$$

3.6 The Complete Formula

After having understood every part of the encoding and reasons for all checks and conditions, we can look at the complete set of formulas. The conjunction of those formulas defines a formula $\Psi(\mathcal{S}, \Phi, n)$ that describes a path schema of size n which is consistent with the counter system $\mathcal{S}$ and a run in that path schema satisfying an lcLTL formula Φ. The correctness of this construction follows from the explanations in the previous sections.

$$\mathsf{id}_0 = s_I \ \wedge\ \mathsf{IType}_{n-1} = \triangleleft \ \wedge\ \mathsf{ICount}_{n-1} = 0 \ \wedge\ \forall_{i\in[1,n-1]} : (\mathsf{id}_{i-1}, \mathsf{id}_i) \in E$$

$$\begin{aligned}
\forall_{i\in[0,n-1]} :\quad & \mathsf{ICount}_i \geq 0 \quad \wedge \quad \big(\mathsf{IType}_i = - \ \Leftrightarrow\ \mathsf{ICount}_i = 1\big) \\
\forall_{i\in[0,n-2]} :\quad & \mathsf{IType}_i = \triangleleft \quad \Rightarrow \quad \mathsf{ICount}_i > 1 \\
\forall_{i\in[1,n-1]} : & \big(\mathsf{IType}_i \in \{-, \triangleright\} \quad \Rightarrow \quad \mathsf{IType}_{i-1} \in \{-, \triangleleft\}\big) \\
\wedge\ & \big(\mathsf{IType}_i \in \{+, \triangleleft\} \quad \Rightarrow \quad \mathsf{IType}_{i-1} \in \{+, \triangleright\} \ \wedge\ \mathsf{ICount}_i = \mathsf{ICount}_{i-1}\big)
\end{aligned}$$

$$\mathsf{steps}_0 = \mathsf{ICount}_0 \wedge \forall_{i\in[1,n-1]} : \mathsf{steps}_i = \mathsf{steps}_{i-1} + \mathsf{ICount}_i$$

$$\begin{aligned}
\forall_{i\in[0,n-1]} : \big(\mathsf{IType}_i = \triangleright \quad &\Rightarrow \quad \mathsf{IStart}_i = \mathsf{id}_i\big) \\
\wedge\ \big(\mathsf{IType}_i = - \quad &\Rightarrow \quad \mathsf{IStart}_i = \bot\big) \\
\forall_{i\in[1,n-1]} : \mathsf{IType}_i \in \{+, \triangleleft\} \quad &\Rightarrow \quad \mathsf{IStart}_i = \mathsf{IStart}_{i-1}
\end{aligned}$$

$$\forall_{i\in[0,n-1]} : \mathsf{IType}_i = \triangleleft \quad \Rightarrow \quad (\mathsf{id}_i, \mathsf{IStart}_i) \in E$$

$$\begin{aligned}
&\Phi \in \mathsf{lbl}_0 \ \wedge\ \forall_{i\in[0,n-1]} : \\
&\qquad \forall_{\xi\in AP} : \xi \in \mathsf{lbl}_i \quad \Leftrightarrow \quad \xi \in \lambda(\mathsf{id}_i) \\
&\qquad \forall_{\neg\varphi=\xi\in\mathsf{sub}(\Phi)} : \xi \in \mathsf{lbl}_i \quad \Leftrightarrow \quad \varphi \notin \mathsf{lbl}_i \\
&\qquad \forall_{\varphi\wedge\psi=\xi\in\mathsf{sub}(\Phi)} : \xi \in \mathsf{lbl}_i \quad \Leftrightarrow \quad \varphi \in \mathsf{lbl}_i \wedge \psi \in \mathsf{lbl}_i \\
&\qquad \forall_{\varphi\vee\psi=\xi\in\mathsf{sub}(\Phi)} : \xi \in \mathsf{lbl}_i \quad \Leftrightarrow \quad \varphi \in \mathsf{lbl}_i \vee \psi \in \mathsf{lbl}_i
\end{aligned}$$

$$\forall_{\mathbf{X}\varphi=\xi\in\mathsf{sub}(\Phi)} : \forall_{i\in[0,n-2]} : \xi \in \mathsf{lbl}_i \ \Leftrightarrow\ \varphi \in \mathsf{lbl}_{i+1}$$

$$\begin{aligned}\forall_{i\in[0,n-1]} : \big(\mathsf{lType}_i = - \quad &\Leftrightarrow \quad \mathsf{lCtr}_i = 0\big)\\ \wedge\ \big(\mathsf{lType}_i = \triangleright \quad &\Leftrightarrow \quad \mathsf{lCtr}_i = 1\big)\\ \forall_{i\in[1,n-1]} : \mathsf{lType}_i \in \{+, \triangleleft\} \quad &\Leftrightarrow \quad \mathsf{lCtr}_i = \mathsf{lCtr}_{i-1} + 1\end{aligned}$$

$$\begin{aligned}\forall_{i\in[0,n-1]} : \big(\mathsf{lType}_i = - \quad &\Leftrightarrow \quad \mathsf{lLen}_i = 0\big)\\ \wedge\ \big(\mathsf{lType}_i = \triangleleft \quad &\Leftrightarrow \quad \mathsf{lLen}_i = \mathsf{lCtr}_i\big)\\ \forall_{i\in[1,n-1]} : \mathsf{lType}_i \in \{+, \triangleleft\} \quad &\Leftrightarrow \quad \mathsf{lLen}_{i-1} = \mathsf{lLen}_i\end{aligned}$$

$$\begin{aligned}&\forall_{i\in[0,n-1]} : \big(\mathsf{lType}_i \neq - \Rightarrow \exists_{l\in L, l\leq i} :\\ &\mathsf{lLen}_i = l \wedge \mathsf{lType}_{i-l} = - \wedge \mathsf{id}_i = \mathsf{id}_{i-l} \wedge \mathsf{lbl}_i = \mathsf{lbl}_{i-l}\big)\\ \wedge\ &\big(\mathsf{lType}_i \neq - \wedge \mathsf{lCount}_i \neq 0 \Rightarrow \exists_{l\in L, l\leq n-i-1} :\\ &\mathsf{lLen}_i = l \wedge \mathsf{lType}_{i+l} = - \wedge \mathsf{id}_i = \mathsf{id}_{i+l} \wedge \mathsf{lbl}_i = \mathsf{lbl}_{i+l}\big)\end{aligned}$$

$$\begin{aligned}&\forall \varphi \mathbf{U}[\ldots]\psi = \xi \in \mathsf{sub}(\Phi) :\\ &\Big(\forall_{i\in[0,n-2]} : \varphi\mathbf{U}\psi \in \mathsf{lbl}_i \ \Leftrightarrow\ \psi \in \mathsf{lbl}_i \vee \big(\varphi \in \mathsf{lbl}_i \wedge \xi \in \mathsf{lbl}_{i+1}\big)\Big)\\ \wedge\ &\Big(\varphi\mathbf{U}\psi \in \mathsf{lbl}_{n-1} \ \Leftrightarrow\ \psi \in \mathsf{lbl}_{n-1} \vee \big(\varphi \in \mathsf{lbl}_{n-1} \wedge \exists_{j\in[1,\hat{l}]} : \mathsf{lCount}_i = 0 \wedge \psi \in \mathsf{lbl}_{n-j}\big)\Big)\end{aligned}$$

$$\begin{aligned}&\forall \varphi \mathbf{U}[\sum_{j=0}^{m} k_j\zeta_j \oplus k]\psi = \xi \in \mathsf{sub}(\Phi) : \mathsf{udCtrs}_0^\xi = 0\\ \wedge\ &\forall_{i\in[1,n-1]} : \mathsf{udCtrs}_i^\xi = \mathsf{udCtrs}_{i-1}^\xi + \sum_{\{j|\zeta_j\in\mathsf{lbl}_{i-1}\}} k_j \cdot \mathsf{lCount}_{i-1}\end{aligned}$$

$$\begin{aligned}&\forall \varphi \mathbf{U}[\sum_{j=0}^{m} k_j\zeta_j \oplus k]\psi = \xi \in \mathsf{sub}(\Phi) : \mathsf{uwCtrs}_0^\xi = 0\\ \wedge\ &\forall_{i\in[1,n-1]} : \mathsf{ite}\big(\mathsf{lType}_{i-1} = \triangleleft,\ \mathsf{uwCtrs}_i^\xi = \mathsf{udCtrs}_i^\xi,\\ &\qquad \mathsf{uwCtrs}_i^\xi = \mathsf{uwCtrs}_{i-1}^\xi + \sum_{\{j|\zeta_j\in\mathsf{lbl}_{i-1}\}} k_j\big)\end{aligned}$$

$$\forall \varphi \mathbf{U}[\sum_{j=0}^{m} k_j\zeta_j \oplus k]\psi = \xi \in \mathsf{sub}(\Phi) :$$
$$\mathsf{ite}\big(\psi \in \mathsf{lbl}_{n-1},\ \mathsf{uBest}^{\xi}_{n-1} = \mathsf{uwCtrs}^{\xi}_{n-1},\ \mathsf{uBest}^{\xi}_{n-1} = \bot^{\xi}\big)$$
$$\wedge\ \forall_{i\in[0,n-2]} : \mathsf{ite}\big(i > 0 \wedge \varphi \in \mathsf{lbl}_{i-1},$$
$$\mathsf{ite}(\psi \in \mathsf{lbl}^{\xi}_{i},\ \mathsf{uBest}^{\xi}_{i} = \mathsf{opt}^{\xi}(\mathsf{uBest}^{\xi}_{i+1}, \mathsf{uwCtrs}^{\xi}_{i}),\ \mathsf{uBest}^{\xi}_{i} = \mathsf{uBest}^{\xi}_{i+1}),$$
$$\mathsf{uBest}^{\xi}_{i} = \bot^{\xi}\big)$$

$$\forall_{i\in[0,n-1]} : \mathsf{lLast}_i \Leftrightarrow \exists_{l\in L, l\leq n-i-1} : \mathsf{lLen}_{i+l} = l \wedge \mathsf{lCount}_{i+l} = 0$$

$$\forall_{\xi=\varphi\mathbf{U}[\ldots]\psi} : \mathsf{allPhi}_{\xi} \Leftrightarrow \forall_{i\in[n-\hat{l},n-1]} : \mathsf{lCount}_i = 0 \Rightarrow \varphi \in \mathsf{lbl}_i$$

$$\forall \varphi \mathbf{U}[\sum_{j=0}^{m} k_j\zeta_j \oplus k]\psi = \xi \in \mathsf{sub}(\Phi) : \forall_{i\in[0,\hat{l}-1]} :$$
$$\mathsf{lType}_{n-\hat{l}+i} = - \Rightarrow \mathsf{ulDelta}^{\xi}_{i} = 0 \wedge \big(\mathsf{lCount}_{n-\hat{l}+i} = 0 \Rightarrow$$
$$\mathsf{ite}(\mathsf{lType}_{n-\hat{l}+i} = \triangleright, \mathsf{ulDelta}^{\xi}_{i} = \sum_{\{j|\zeta_j\in\mathsf{lbl}_{n-\hat{l}+i}\}} k_j,$$
$$\mathsf{ulDelta}^{\xi}_{i} = \mathsf{ulDelta}^{\xi}_{i-1} + \sum_{\{j|\zeta_j\in\mathsf{lbl}_{n-\hat{l}+i}\}} k_j)\big)$$

$$\forall \varphi \mathbf{U}[\sum_{j=0}^{m} k_j\zeta_j \oplus k]\psi = \xi \in \mathsf{sub}(\Phi) : \forall_{i\in[0,n-2]} : \mathsf{lType}_i = - \Rightarrow$$
$$\Big(\xi \in \mathsf{lbl}_i \ \Leftrightarrow\ \varphi\mathbf{U}\psi \in \mathsf{lbl}_i \wedge \Big(\big(\psi \in \mathsf{lbl}_i\ \wedge 0 \oplus k\big)$$
$$\vee\ \big(\varphi \in \mathsf{lbl}_i\ \wedge\ (\mathsf{uBest}^{\xi}_{i+1} \oplus k + \mathsf{uwCtrs}^{\xi}_{i}) \vee (\mathsf{lLast}_i \wedge \mathsf{allPhi}_{\xi} \wedge \mathsf{ulDelta}^{\xi}_{\hat{l}-1} \hat{\oplus}\ 0)\big)\Big)\Big)$$

$$\forall (k_c, \oplus, k) = g \in \bigcup_{e\in E} \delta_g(e) : \forall_{i\in[0,\hat{l}-1]} :$$
$$\mathsf{ite}\Big(\mathsf{lCount}_{n-\hat{l}+i} > 0,\ \mathsf{glDelta}^{g}_{i} = 0,$$
$$\mathsf{ite}\big(\mathsf{lType}_{n-\hat{l}+i} = \triangleright, \mathsf{glDelta}^{g}_{i} = \sum_{c\in C} k_c(c)\cdot\delta_{u(c)}(\mathsf{id}_{n-\hat{l}+i-1}, \mathsf{id}_{n-\hat{l}+i}),$$
$$\mathsf{glDelta}^{g}_{i} = \mathsf{glDelta}^{g}_{i-1} + \sum_{c\in C} k_c(c)\cdot\delta_{u(c)}(\mathsf{id}_{n-\hat{l}+i-1}, \mathsf{id}_{n-\hat{l}+i})\big)\Big)$$

$$\forall_{c\in C} : \mathsf{gCtrs}_0^c = 0 \ \wedge \ \forall_{i\in[1,n-1]} : \forall_{c\in C} : \mathsf{gCtrs}_i^c = \mathsf{gCtrs}_{i-1}^c + \mathsf{lCount}_{i-1} \cdot \delta_{u(c)}(\mathsf{id}_{i-1}, \mathsf{id}_i)$$
$$\wedge \ (\mathsf{lType}_i = - \Rightarrow \bigwedge_{(k_c,\oplus,k)=g\in\delta_g(id_{i-1},id_i)} \sum_{c\in C} k_c(c) \cdot \mathsf{gVals}_i^c \oplus k)$$
$$\wedge \ (\mathsf{lCount}_i = 0 \Rightarrow \bigwedge_{g\in\delta_g(id_{i-1},id_i)} \mathsf{glDelta}_{l-1}^g \check{\oplus}\ 0)$$

The main results of this chapter are summarized in the following theorems.

Theorem 3.2 (Soundness)**.** *Let $\mathcal{S}$ be a counter system, Φ an* lcLTL *formula and $n \in \mathbb{N}$. Then*

$$\Psi(\mathcal{S}, \Phi, n) \in \mathsf{QPA\text{-}SAT} \quad \Rightarrow \quad \mathcal{S} \models_\exists \Phi.$$

This follows directly from the equivalence of satisfiability of the constructed formula with the existence of a path schema and a run in it that satisfies the formula, because, by definition, the same run is also valid in the original system.

As already discussed in the previous chapter, no finite set of path schemas is sufficient to represent all runs in an arbitrary counter system. So there is no finite n big enough that one can be sure that there is no run witnessing the formula Φ.

On the other hand, for flat systems we do know that such a finite set of path schemas exists, hence, if a satisfying run is possible at all, it can be witnessed by one of the path schemas. The results in [DHL+17] show that each such path schema can be transformed into a bigger path schema that lends itself to a labelling like the one presented in this chapter. Also, there is a theoretical upper bound on the size of the resulting path schema that depends on some fixed polynomial we will call p here.

As the path schemas used in our encoding are equivalent to the ones constructed in [DHL+17], these results apply to our encoded path schemas as well, when restricting ourselves to the fLTL fragment of lcLTL and flat Kripke structures. Hence, we can state the following result:

Theorem 3.3. *Let $\mathcal{S}$ be a flat Kripke structure, Φ an* fLTL *formula and $n \in \mathbb{N}$ with $n \geq 2^{p(|\Phi|+|\mathcal{S}|)}$. Then*

$$\mathcal{S} \models_\exists \Phi \quad \Leftrightarrow \quad \Psi(\mathcal{S}, \Phi, n) \in \mathsf{QPA\text{-}SAT}.$$

Notice, that as discussed in the last chapter, it should be possible to generalize the proofs in [DHL+17] to flat counter systems and lcLTL, so that the result would apply in our slightly extended setting as well. This means, that in principle, for flat systems and a sufficiently large n, the satisfiability of the constructed formula precisely captures our existential model checking problem, as this n is large enough to represent all possible path schemas in the system and all shorter path schemas can be extended to this size by unrolling the infinite loop.

3.7 Upper Bounds

First let us calculate the number of variables and size of the formula in dependence on the different input parameters. The number of unique subformulas $|\mathsf{sub}(\Phi)|$ can be assumed to be proportional to the formula size $|\Phi|$, considering atomic propositions, integer numbers and operators as units. Now let $u = |\{\xi \mid \varphi \mathbf{U}[\ldots]\psi = \xi \in \mathsf{sub}(\Phi)\}|$ indicate the number of $\varphi\mathbf{U}[\ldots]\psi$ subformulas present in the supplied formula Φ, $\hat{l} = \min(n, \max(L))$ as above the size of the largest simple loop in $\mathcal{S}$, $g = \sum_{e \in E} |\delta_g(e)|$ the total number of individual guards present in $\mathcal{S}$ and $c = |C|$ the number of counters in $\mathcal{S}$.

There are four kinds of variable types present in the formula—booleans, integers, loop types and state ids—but as discussed in the beginning, we can encode all of these using natural numbers, hence we do not need to differentiate them here.

We need $8n$ variables to encode the path schema with individual states, looping positions, number of iterations, loop lengths, etc. which are properties independent of the supplied lcLTL formula. Additionally we need a label variable for every position and subformula, hence adding another $|\Phi| \cdot n$ variables. For the correct labelling of the $\mathbf{U}[\ldots]$ subformulas we use $3n \cdot u$ variables to encode the counters, another u variables indicating whether the infinite loop is able to accumulate the counter value and $u \cdot \hat{l}$ variables to calculate the total effect of one iteration of the last loop. Similarly, we use $c \cdot n$ variables to encode counters of $\mathcal{S}$ and $g \cdot \hat{l}$ variables to calculate the loop effect on every counter system guard.

All together, the number of variables is:

$$(8 + |\Phi| + 3u + c)n + u(\hat{l} + 1) + g\hat{l} \overset{\hat{l} \leq n}{\leq} (8 + |\Phi| + 4u + c + g)n + u \overset{u \leq |\Phi|}{\leq} (8 + 6 \cdot |\Phi| + c + g) \cdot n$$

This is obviously linear in each parameter individually. One can reasonably assume that a realistic counter system has some small and fixed number of counters and guards and that the formula we are checking is succinctly expressing some property in a human-readable way, hence $|\Phi|$ is also small. So the main message here is that the number of variables grows linear in the path schema size.

Now we want to estimate the size of the resulting formula. We consider the presented formula pieces as constant units and analyze the growth due to the elimination of all explicit and hidden quantifiers.

The formula ensuring that neighboring states have an existing edge expands to a formula of the size $\mathcal{O}(n \cdot |E|)$, as each possible edge must be enumerated in each position. The formulas ensuring a valid loop encoding are all linear in n. The formula ensuring correct labels has the size $\mathcal{O}(n \cdot |\Phi|)$, as we need constraints for every position and subformula. The formula enforcing the left and right unrollings has the size $\mathcal{O}(n \cdot |L|)$. The formulas enforcing correct counter propagation for $\mathbf{U}[\ldots]$ constraints have a total size of $\mathcal{O}(n \cdot u)$. Similar counters for the guards need a formula of size $\mathcal{O}(n \cdot c)$. The formulas checking whether the last loop is accumulating have the size $\mathcal{O}(\hat{l} \cdot u)$. The formulas calculating the total loop effect on counters for the $\mathbf{U}[\ldots]$ have a size of $\mathcal{O}(\hat{l} \cdot u)$ and the similar formula for counter system guards has a size of $\mathcal{O}(\hat{l} \cdot g)$. The valid counter updates and satisfied guards in each position require additional $\mathcal{O}(n \cdot (g + c))$. Added together, we have:

$$\mathcal{O}(n \cdot (|E| + |\Phi| + |L| + u + c + g) + \hat{l} \cdot (u + g)) \overset{\hat{l} \leq n, u \leq |\Phi|}{\subseteq} \mathcal{O}(n \cdot (|E| + |\Phi| + |L| + c + g))$$

Hence, all other things being equal, the formula grows only linear in the specified path schema size. Again, one can reasonably assume some fixed set of counters in the counter system and a number of atomic guards which is at most proportional to the number of edges.

Under these conditions, we can further simplify the upper bound to:

$$\mathcal{O}(n \cdot (|E| + |\Phi| + |L|))$$

Notice, that if the lengths of simple loops in $\mathcal{S}$ are not known and every possible length needs to be accounted for, this degenerates to:

$$\mathcal{O}(n^2 \cdot (|E| + |\Phi|))$$

After our detailed analysis now we can substitute c, g and $|E|$ in the bounds with $|S|$ to obtain the following result:

Theorem 3.4 (Formula Size)**.** *Let $\mathcal{S}$ be a flat counter system, Φ an* lcLTL *formula and $n \in \mathbb{N}$. Then the following holds:*

- *The number of variables in Ψ is in $\mathcal{O}((|S| + |\Phi|) \cdot n)$.*
- *The size of the formula Ψ is in $\mathcal{O}((|S| + |\Phi| + |L|) \cdot n)$.*

In the next chapter we will briefly discuss the trade-off of the need to obtain the set L and the size of the formula.

3.8 Remarks

Notice that we only allow inequality constraints in the $\mathbf{U}[\ldots]$ operator. Theoretically it would be possible to adapt this scheme to equality constraints, too, but this would not be possible without a quadratic blow-up in the number of variables. The reason for this is that with equality constraints it is not possible to use a single sequence of variables to propagate some optimum value. The equality constraint is not monotone regarding satisfaction and if some exact value is satisfactory from the point of view of one position, it can be just as well not be satisfactory from some neighboring position. Therefore, it would be necessary to introduce an individual propagation sequence for every position of the schema.

Also notice that we have neither conjunction nor disjunction of constraints for the $\mathbf{U}[\ldots]$ operator. The disjunction is not necessary, because $\varphi\mathbf{U}[c_1 \vee c_2]\psi \Leftrightarrow \varphi\mathbf{U}[c_1]\psi \vee \varphi\mathbf{U}[c_2]\psi$, where c_1, c_2 denote linear constraints. This is easy to see from

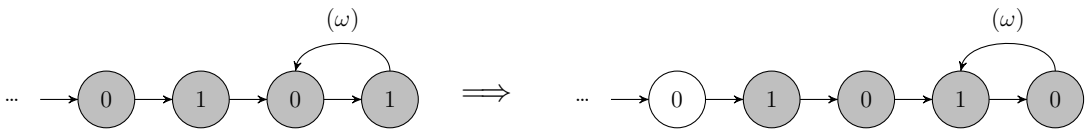
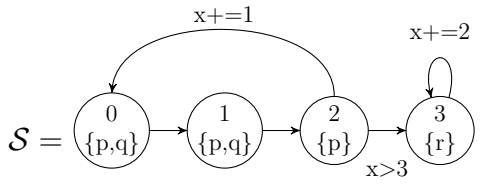


Figure 3.9: Example of a single-state unrolling to extend the schema size.

the semantics, as this would mean that we are happy with some future ψ-position which satisfies just one of the constraints. Conjunction, on the other hand, cannot be easily represented using some propagation in the vein of the uBest counters, because we would need information about the position of the ψ where each value originated from to recognize when the same position is sufficient for both constraints.

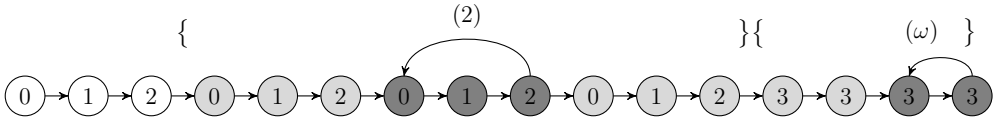
A useful fact we will use in the next chapter is that for every size n for which a satisfying schema and run exist, there exists a schema with size $n + 1$ representing the same run. The bigger schema can be obtained by unrolling a single state of the last loop (see figure 3.9). Hence, if a satisfying run exists which has a minimal representation of size n in our encoding, in theory we are able to find it for an arbitrarily large schema size $n + k$.

Now having fully developed a representation of formula witnesses consisting of a path schema and an embedded run, we will leave this chapter with figure 3.10 that shows a complete example of the developed encoding. In the next chapter we will see how this can be put into practice as a verification technique.



$\mathcal{S} =$ $\qquad L = \{1, 2, 3\} \qquad \Phi = p\mathbf{U}[q > 8]r$

$\mathcal{R} =$



i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
id_i	0	1	2	0	1	2	0	1	2	0	1	2	3	3	3	3
lType_i	–	–	–	–	–	–	$\triangleright$	$+$	$\triangleleft$	–	–	–	–	–	$\triangleright$	$\triangleleft$
lCount_i	1	1	1	1	1	1	2	2	2	1	1	1	1	1	0	0
lStart_i							$0 \to$	$0 \to$	0						$2 \to$	2
lCtr_i							$1 \to$	$2 \to$	3						$1 \to$	2
lLen_i							$3 \leftarrow$	$3 \leftarrow$	3						$2 \leftarrow$	2
lLast_i													$\top$	$\top$		
steps_i	1	2	3	4	5	6	8	10	12	13	14	15	16	17	17	17
udCtrs_i^Φ	0	1	2	2	3	4	4	6	8	8	9	10	10	10	10	10
uwCtrs_i^Φ	0	1	2	2	3	4	4	5	6	8	9	10	10	10	10	10
uBest_i^Φ	$\bot^\Phi$	10	10	10	10	10	10	10	10	10	10	10	10	$\bot^\Phi$	$\bot^\Phi$	10
gCtrs_i^x	0	0	0	1	1	1	2	2	2	4	4	4	4	6	8	8
lbls_i^p	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$				
lbls_i^q	$\top$	$\top$		$\top$	$\top$		$\top$	$\top$		$\top$	$\top$					
lbls_i^r													$\top$	$\top$	$\top$	$\top$
$\mathsf{lbls}_i^{p\mathbf{U}r}$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
lbls_i^Φ	$\top$	$\top$														

$\mathsf{ulDelta}_2^\Phi = 0 \quad \mathsf{allPhi}_\Phi = \bot \quad \mathsf{glDelta}_2^{x>3} = 4$

Figure 3.10: A complete example of a counter system $\mathcal{S}$, formula Φ and a satisfying run $\mathcal{R}$ in a path schema with size $n = 16$, encoded in a matrix of variables, irrelevant values and false booleans omitted for clarity.

4 Implementation

In the previous chapters we have seen the logic lcLTL, learned about flat counter systems and finally developed a formula in Presburger arithmetic that is satisfiable only if a run in some flat underapproximation satisfies some lcLTL formula Φ. Also, such a run can be easily reconstructed from the valuation of the variables in the formula.

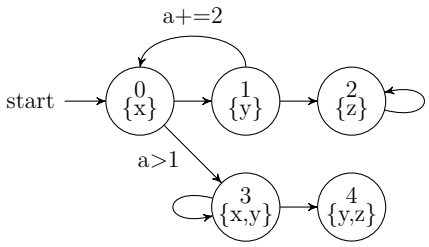
In this chapter we will put the described approach in practice, discuss some aspects of the implementation and some deviations from the theory which was laid out in the last chapter. Finally we will evaluate the developed software tool.

The software can be obtained under the following URL:
`https://github.com/apirogov/flat-checker`

4.1 Choice of Technology

The checker is implemented in Haskell. Haskell is a well-known functional programming language with a focus on expressiveness and correctness. It has a strong static type system, good control of side-effects and helps the developer to express himself by its support for advanced abstractions. Expressiveness means also less redundancy and less code to maintain, leaving less room for mistakes that do not concern the actual program logic itself.

While it is possible to obtain compiled Haskell programs with competitive performance to C, it is not as easy to reason about time and space complexity for idiomatic code as it is with imperative languages. With the task at hand this is not a concern, as most of the computational work is delegated to an SMT-solver. Z3 [DMB08] is used as the solver, as it is currently one of the best-performing SMT-solvers (e.g. see [CDW14]).




```
digraph {
  0[props="x"];
  1[props="y"];
  2[props="z"];
  3[props="x,y"];
  4[props="y,z"];
  0 -> 1 -> 2 -> 2;
  1 -> 0[updates="a+=2"];
  0 -> 3[guards="[a>1]"];
  3 -> 3 -> 4;
}
```

Figure 4.1: A counter system and its representation in the customized DOT-format.

4.2 Overview and Interface

The checker is controlled via the command line. The mandatory parameters are a counter system, an lcLTL formula and a path schema size. Some additional flags can modify the behaviour.

The counter system must be provided as a file in a subset of the DOT-format, either over the standard input or via an argument containing the file name. DOT is a well-known format to represent graphs and has the benefit of being renderable with tools from the well-known GraphViz suite [GN00], as it is its native format. A library is used to parse the file containing the graph.

The restrictions on accepted graphs are the following—only a directed graph (started with `digraph`) with self-loops but no multiple edges between vertices is allowed. All node declarations must precede all edge declarations. Nodes must be identified with natural numbers and 0 must identify the initial state. DOT allows for attaching custom attributes to nodes and edges. The checker parses the `props` attribute from nodes and the `guards` and `updates` attributes from edges. All other attributes are ignored. An example is presented in figure 4.1.

The formula must be passed directly as a command line argument and corresponds to the grammar defined in chapter 2, including syntactic sugar for $\mathbf{F}$, $\mathbf{G}$ and other useful operators. The only restriction is that every binary operator must be parenthesized and propositions must consist of lowercase strings only, as uppercase letters are

reserved for operators. For example, the formula

$$\mathbf{F}[\mathsf{connected}\mathbf{U}[\mathsf{error} > 3]\mathsf{close} \geq 5]\mathsf{shutdown}$$

that we have seen in chapter 2 looks like the following in the accepted syntax:

$$\mathtt{F}[(\mathtt{connectedU}[\mathtt{error} > \mathtt{3}]\mathtt{close}) >= \mathtt{5}]\mathtt{shutdown}$$

The lengths of simple loops present in the counter system can optionally be provided. If they are omitted, in the first step the checker calculates those.

Then the checker performs the translation described in chapter 3 and asks Z3 to find a satisfying valuation for the formula. The found solution (if any) is translated back from the variables into a human-readable form and printed out.

4.3 Calculation of the Simple Loop Lengths

To calculate a list of all simple loop lengths, Johnson's algorithm [Joh75] has been implemented, which seems to be the best algorithm for this task (e.g. see [MD76]). Literature research yielded no theoretically or practically faster algorithm.

The algorithm is implemented directly from the pseudocode in the paper. So while it is not idiomatic Haskell code, it is as close to the original algorithm as possible. The only modification to the algorithm is that instead of reporting a found cycle, just the length of the loop is preserved. Self-loops are eliminated in a preprocessing step, as Johnson's algorithm does not work with self-loops.

The runtime of the algorithm is bounded by $\mathcal{O}((n+e)(c+1))$, where n is the number of vertices, e the number of edges and c the number of simple loops in the graph. It is clear that c can become very large in arbitrary graphs and the related Hamilton-Cycle problem is a well-known **NP**-complete problem.

While realistic counter systems modelling some program flow can be assumed to have a not very big number of different loops, of course such cases can still appear or be constructed. If such a case is known up-front and the user has at least some qualified idea about the lengths of loops, one can supply a list of loop lengths and

skip the invocation of Johnson's algorithm, if an unacceptably high running time is expected.

If the user decides to supply the list of loop lengths manually, of course he has the full responsibility for the consequences, which are:

- If the list is redundant, i.e. contains lengths that are not even present in the system, the size of the formula is unnecessarily increased and the solving could take much longer.
- If the list is incomplete, i.e. some lengths of simple loops are missing, then runs containing these loops cannot be represented and therefore it is possible that no solution can be found for a formula, even though a satisfying run does exist in the system and could be represented by a schema of size n.

A responsible user would let the checker calculate the lengths once and supply the found lengths in subsequent invocations of the checker on the same counter system. This prevents redundant recalculations but does not inhibit the search for runs unnecessarily.

Should a situation arise where it is known that the counter system contains a very large number of distinct simple loops so that the calculation of the loop lengths is expected to not terminate in a reasonable time, there is an approach to be able to use this anyway, in the case that the largest length $\hat{l}$ of such a loop is rather small and known (or at least a good upper bound is known). In these circumstances, the user could provide the list $1, 2, \ldots, \hat{l}$ as the list of loop lengths, which would for sure contain all existing loop lengths and probably some redundant lengths that just increase the formula size. So for small maximal loops, a "brute force" approach concerning loop lengths might be worth a try.

If the largest loop length is not known or not small, one could assume that in case Johnson's algorithm fails to terminate in reasonable time, the system under scrutiny is probably larger than the maximum viable path schema size that can be solved successfully in reasonable time by Z3, hence trying to provide the list $1, 2, \ldots, |E|$ would effectively lead to a quadratic formula size, as discussed in the last chapter, because likely $|E| \gg n$.

4.4 Choice of the Path Schema Size

In general one cannot assume the user to have a good idea of an appropriate path schema size to use for his problem. As discussed, a run can be always unrolled to fit any larger size in theory, so in principle, a user could just provide a big number and hope to get a solution, if one exists.

In practice this does not always work. Z3 (and other similar software) is a generic SMT solver without domain specific knowledge and cannot be expected to find this unrolling strategy to generate big runs, so the best bet in general is to use the smallest path schema possible, which means a smaller formula, less variables and hence a smaller search space.

A modification of the encoding that marks the end of a run explicitly and hence allows for encoding of a shorter run using just a subset of the variables has been implemented. It turned out to be much worse than the fixed-size path schema encoding. At least a part of the problem is of course the fact that such a flexible encoding does neither reduce the formula size nor the number of variables, making the task of Z3 even harder due to more degrees of freedom which are not effectively relevant for the solution, in the sense that non-satisfactory valuations can be tried multiple times. Hence, this generalization is not fruitful and was dropped quickly.

Instead, the checker provides a flag that changes the semantics of the path schema size parameter n. In the default mode, it means a fixed path schema size and one invocation of the SMT-solver. With the flag enabled, the meaning changes to maximum path schema size and the checker begins with a small value, successively doubling the schema size on failure, up to the provided maximum value. Starting with small values increases the chance that Z3 terminates and reports success or failure, so if a short run exists, it is more likely that it will be found. At the same time, the user is freed from the burden of choosing a suitable schema size and can just provide some arbitrary upper bound after which the checker will give up. In principle, as the formula produced by the encoding scales linearly in n (theorem 3.4), increasing n is not unnecessarily expensive, enabling us to try such search tactics.

It might be desirable to get the smallest witness of the formula. The successive doubling does not guarantee the smallest satisfying run, but the checker provides

another flag which modifies the behaviour so that after finding the first satisfying run, a binary search in the interval between the last unsuccessful and the first successful path schema size is used to obtain the smallest run which is possible in the presented encoding.

Hence, we still harness the aforementioned theoretical property, because it ensures that we do not have to try every single possible path schema size and only need to try a logarithmic number of sizes to find the solution, if the chosen n is large enough.

4.5 Theory Versus Practice

While in the last chapter we argued that the encoding can be stated using a minimal syntactical core of quantifier-free Presburger arithmetic and just integer variables, this is in fact not the optimal input for an SMT-solver.

SMT-solvers provide a much richer language to state a satisfiability problem, supporting different data types like booleans and integers and also allowing to define own data types. Also, for each data type there is a rich set of basic supported operations. Such general-purpose solvers use different strategies and heuristics to find solutions, depending on the data types and structural information between the variables which is encoded by the constraints. Hence, using appropriate types and operations that capture the intended semantics more precisely helps the solver to choose the hopefully most suitable tactic.

So contrary to the theory, in practice the formulas were implemented as close to the presentation in the last chapter as possible, instead of the theoretical minimum presented in the beginning. There are custom finite enumeration types for loop types and state ids, booleans for the labels and flags. Integers are only used for variables that in fact are used as such (i.e. counters and other numeric values). This for example prevents the application of arithmetic-based tactics to find valid state sequences, as obviously any arithmetic operation on state ids is meaningless, even if we incidentally refer to states using natural numbers.

The use of atomic operations for constructions like ite instead of the logically equivalent expansion also improved the performance of the solver on the formulas. It is likely that expressing the if-then-else semantics explicitly with $\mathsf{ite}(x, y, z)$ triggers the application of a better tactic which is not used if the logical relationship is obscured by the more generic $(x \wedge y) \vee (\neg x \wedge z)$. After this discovery, special care was taken to use the most expressive operator in each situation.

The checker offers two flags that change the representation of loop types from a custom data type to an explicit pair of booleans and from the custom finite state id type to regular integers. While in some experiments these representations performed better for some path schema sizes, overall it seems that the custom data types are better suited for the task because of the aforementioned reasons, hence this is the default setting in the checker.

4.6 Additional Features

Additionally to the theory presented in the last chapters, the actual implementation has two extensions that will just be mentioned here briefly.

4.6.1 Disjunctions in Counter System Guards

The tool supports not just conjunctions of guards attached to labels, but allows for annotation of the edges in disjunctive normal form. The disjunctions get eliminated in a preprocessing step by duplicating the target node a sufficient number of times and splitting the monomes for each to target one of the node copies. An illustration is presented in figure 4.2. Notice that due to this preprocessing, in total the counter system grows proportionally to the number of disjunctions multiplied by the outdegree of the affected nodes.

Of course, a loop in the path schema can only represent the case that the same monome is satisfied in each iteration. If the run is alternating between the satisfied monomes while traversing the same edge, it can only be represented by splitting the loop iterations accordingly. Hence, the compact representation works only if the constraint invariance holds in the loop on the level of monomes, i.e. without such alternation.

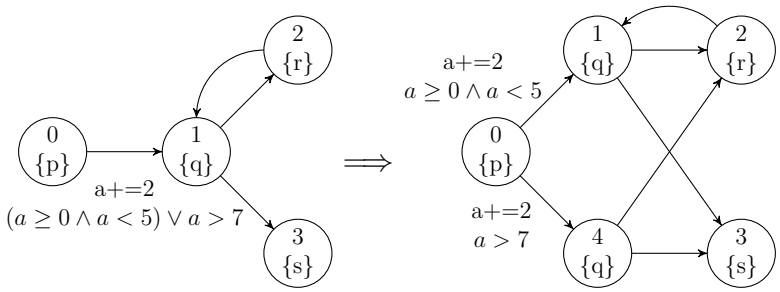


Figure 4.2: The edge with disjunctive constraints is split to target copies of the original target node sharing the same propositions and outgoing edges.

By construction, each run that is found in the transformed counter system can be simulated equivalently in the original system when traversing the disjunctive edges, because no LTL formula can differentiate between such copies of a state.

4.6.2 Counter Systems with Resets

Another useful extension that is implemented in the tool is the possibility to not only increment counters in the counter system, but also set them to fixed values. While it is possible to model the system accordingly to simulate a counter reset with a guarded increment or decrement construction, this possibly adds a nested loop inside a different loop, eliminating the possibility to apply the compact loop representation with just the simple loop lengths (see figure 4.3). Also, direct assignment to variables is a much used feature in any realistic piece of code that might be model-checked. Hence, implementing this directly enables the application of the tool to a much wider pool of problems.

For this to work, instead of one left unrolling of a loop in the path schema two unrollings are required. This is necessary, because in the first loop iteration the counters potentially get reset for the first time and some guards must be passed with the previous value. Only in the subsequent iterations the counters which get reset inside the loop exhibit a consistent behaviour. The calculation of the loop deltas for the guards in the infinite loop are adapted accordingly to ignore counters that are reset there, as they do not accumulate value over the iterations due to the reset.

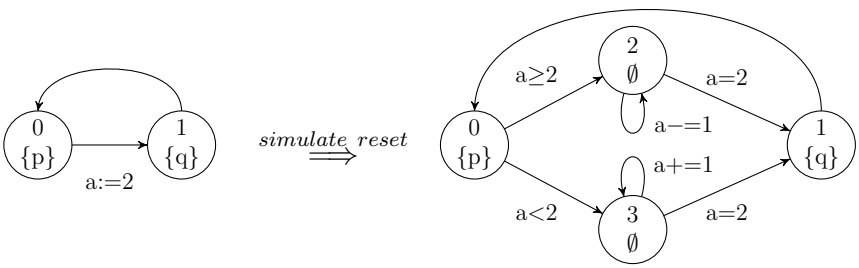


Figure 4.3: Without direct support for counter resets, an otherwise flat part of a system (left) may only be modelled as non-flat (right), inhibiting a compact run representation in the presented encoding.

4.7 A Small Usage Example

Assume that the graph in figure 4.1 is stored in a file `graph.dot` and we want to search for a minimal representable run satisfying the formula $\mathbf{F}[x > 100]z$ with an upper bound on the path schema length of 20. The tool would be invoked with:

```
flat-checker -g graph.dot -f "F[x>100]z" -n 20 -m
```

A possible output of the program would look like this:

```
Solution:
N  | ID LT LS LL LC F[1x > 100]z  GC: a      Labels:
1  | 0              (0,0,-206)        0      x, Fz, true, F[1x > 100]z
2  | 1              (1,1,101)         0      Fz, true
3  | 0              (1,1,101)         2      x, Fz, true
4  | 1              (2,2,101)         2      Fz, true
5  | 0  <  0  2  98 (2,2,101)         4      x, Fz, true
6  | 1  -  0  2  98 (100,3,101)       4      Fz, true
7  | 0              (100,100,101)     200    x, Fz, true
8  | 1              (101,101,101)     200    Fz, true
9  | 2              (101,101,101)     200    z, Fz, true
10 | 2              (101,101,101)     200    z, Fz, true
11 | 2  +           (101,101,101)     200    z, Fz, true
12 | 2  +           (101,101,101)     200    z, Fz, true
13 | 2  <  2  2  0  (101,101,101)     200    z, Fz, true
14 | 2  -  2  2  0  (101,101,101)     200    z, Fz, true
                    True/=
```

Contrary to the usual presentation in this thesis, the runs in the output are aligned vertically. The leftmost column just numbers the positions of the path schema. The `ID` column presents the state ids of the counter system. The next four columns represent the loop structure of the path schema, i.e. loop type, loop start node id, loop length and loop count.

Notice that the formula is represented as $\mathsf{true}\mathbf{U}[x > 100]z$ in the program. Hence, the following column shows the values of the $\mathbf{U}[\ldots]$ counter ($\mathsf{udCtrs}, \mathsf{uwCtrs}, \mathsf{uBest}$ values) and indicates on the bottom the allPhi value and whether the last loop is positive, negative or neutral. In this example, the last loop is neutral and allPhi is trivially true. After columns for the $\mathbf{U}[\ldots]$ operators the values of all system counters and finally all subformula labels which are true in that position follow.

Having seen how `flat-checker` can be used, in the next chapter we want to see how well it can perform on different more realistically looking problems.

5 Application

In this chapter we will see how `flat-checker` performs on a series of test cases of varying complexity. The different problems were evaluated on regular computers, ranging from 2 to 4 cores with $2.2 - 2.7$ GHz. As the runtime is determined by the performance of the SMT-solver and not `flat-checker` itself, the times listed should be just seen as a rough qualitative guide and not as rigorous benchmark results. They depend not only on the problem at hand and the hardware available, but also on factors like the version and configuration of Z3 that is used.

5.1 Evaluating Scalability

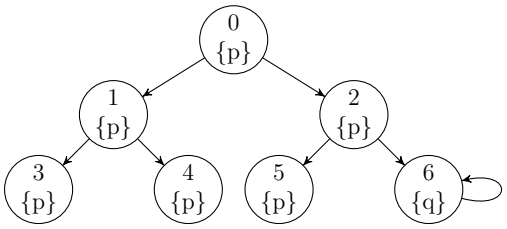
5.1.1 Enumeration of the Cycle Lengths

First the implementation of the cycle length enumeration algorithm has been evaluated. For this purpose, the algorithm has been applied to a series of fully connected graphs with n nodes, as those are a worst-case input that increases the number of different cycles by a large amount. The graph size in terms of edges grows quadratically in n, while the growth of the number of distinct simple cycles grows in $\mathcal{O}(n!)$ due to the different permutations of the node visiting sequence. The runtimes are summarized in the table on the right. The runtime for $n \leq 6$ is actually below the threshold of meaningful measurement and can be seen as neglectable. The cycle enumeration should be fast enough for all practical purposes, as fully connected subgraphs are likely not a common substructure in realistic inputs that represent a meaningful program flow.

n	**time**
≤ 6	~ 0.005 s
7	0.013 s
8	0.077 s
9	0.66 s
10	6.85 s
11	74 s

$\varphi_n = \mathbf{X}^{\lceil \log(7) \rceil}\mathbf{G}(\neg p \wedge q) \quad \hat{\varphi_n} = \mathbf{X}^{\lceil \log(7) \rceil}\mathbf{G}(\neg p \wedge r)$

$\mathcal{S}_n =$



n	**time**(φ_n)	**time**($\hat{\varphi_n}$)
16	0.11 s	0.092 s
32	0.18 s	0.14 s
64	0.48 s	0.24 s
128	1.3 s	0.55 s
256	10 s	0.98 s
512	68 s	2.82 s
1024	6 m 22 s	6.41 s

Figure 5.1: Example of the binary tree benchmark for $n = 7$ on the left and table of the results on the right. $\mathbf{X}^n$ indicates a sequence of n $\mathbf{X}$ operators.

A related substructure that can require a long loop length computation time is given by deeply nested conditional branches contained in program loops. To get about the same number of distinct loops as a full graph with 10 nodes, which contains $\sum_{k=2}^{10} \frac{1}{k} \cdot \frac{10!}{(10-k)!} = 1112073$ unique simple cycles, one requires a full binary subgraph with a height of about 20 which is contained within a loop, giving $2^{20} = 1048576$ distinct loops. A branching of such depth is highly unlikely for realistic program code.

5.1.2 A Simple Growable Benchmark

Next, `flat-checker` was evaluated on a benchmark that was constructed to be both simple and well parametrizable. The benchmark consists of a binary tree with n nodes, in the case of $n = 2^x - 1$ it is a full binary tree. The last node contains a self-loop. The formula schema used on this tree grows logarithmically with the number of nodes by adding $\mathbf{X}$ operators and describes a simple constraint on the only valid run that is possible in the graph. The benchmark structure and results are illustrated in figure 5.1. Additionally, the graph was tested with a unsatisfiable modification of the formula. The path schema size was chosen sufficiently high in all cases.

The solving time of the satisfiable formula multiplies by a factor of approximately $6 - 7$ for each doubling of the number of nodes in the graph, indicating a polynomial growth. Notice that for this graph the number of edges also grows linear with the

number of nodes, so this behaviour is not due to some hidden increase of the encoding due to the number of edges. As the formulas get solved by Z3, which is a general purpose solver, such performance is not surprising. `flat-checker` has no influence on the runtime other than providing the most suitable representation as the input. Fortunately, it seems that Z3 is able to detect unsatisfiability quickly in some cases.

5.2 Combining Most Features

Now we will look at a test case specifically constructed to use as many features supported by `flat-checker` as possible, illustrated in figure 5.2. It consists of a binary tree with a height of 4, each leaf leads back up to the root and has a unique proposition. Each pair of neighboring leafs provides an increment and a decrement update for a different counter, each guarded with a constraint over the previous counter. The formula used for this construction specifies that for each counter the increment associated with it is used a sufficient number of times, while the guards restrict the maximum value of the previous counter. Hence, a valid run must use both increments and decrements to satisfy both the constraints in the formula and the constraints at the edges of the counter system.

n	**time**	$\mathcal{S} \overset{?}{\models_\exists} \varphi$
4	0.025s	?
8	0.083s	?
16	0.325s	?
32	4.87s	?
64	53.81s	?
96	8m 28s	?
104	2h 16m	?
106	1h 32m	?
107	1h 16m	?
108	11m 18s	✓
112	3m 11s	✓
128	9m 32s	✓

The table shows the runtimes that were obtained for the different path schema sizes during an automatic search for a minimal representable run witnessing the given formula in the illustrated counter system. Notice that while there is a growth tendency, unfortunately the time requirements do not increase in a predictable or monotonic way. In this case, Z3 seems to exhibit a spike in the required runtime for path schema sizes that are just below the minimal size that can encode a valid run successfully.

$$\varphi = (\neg \mathsf{i2}\mathbf{U}[\mathsf{i1} \geq 100]\mathsf{i2}) \wedge (\neg \mathsf{i3}\mathbf{U}[\mathsf{i2} \geq 100]\mathsf{i3}) \wedge (\neg \mathsf{i4}\mathbf{U}[\mathsf{i3} \geq 100]\mathsf{i4}) \wedge \mathbf{GF}[\mathsf{i4} \geq 100]\mathsf{i4}$$

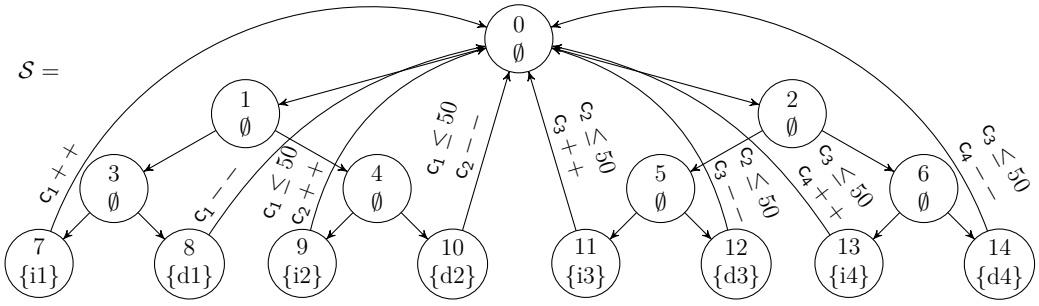


Figure 5.2: A test case for `flat-checker` that uses most of the supported features.

5.3 The RERS Challenge 2017

What the RERS Challenge [HIM+12] is about is best said by the organizers themselves on the homepage (`http://www.rers-challenge.org/2017/`):

> **Rigorous Examination of Reactive Systems (RERS)**
>
> RERS is designed to encourage software developers and researchers to apply and combine their tools and approaches [...] to answer evaluation questions for reachability and LTL formulas on specifically designed benchmarks. The goal of this challenge is to provide a basis for the comparison of verification techniques and available tools.
>
> The benchmarks are automatically synthesized to exhibit chosen properties and then enhanced to include dedicated dimensions of difficulty, ranging from conceptual complexity of the properties (e.g. reachability, full safety, liveness), over size of the reactive systems (a few hundred to tens of thousands lines of code), to contained language features (arrays and index arithmetic). They are therefore especially suited for community-overlapping tool comparisons. What distinguishes RERS from other challenges is that the challenge problems can be approached in a free-style manner: it is highly encouraged to combine and exploit all known (even unusual) approaches to software verification. [...] RERS is the only challenge with a special track for LTL analysis on synthesized benchmarks.

Many other benchmark suites and challenges are not well suited for `flat-checker` on the one hand due to prohibitive language features or program sizes, on the other by the fact that they often focus on pure reachability problems. While checking reachability is possible with `flat-checker`, it does use just a rather small subset of the features exposed by it, explicit LTL verification problems are more interesting to evaluate `flat-checker`.

The RERS problems are posed as a C99 and Java programs that always follow the same general structure. A program has an internal state defined by a set of global variables and contains one main loop that reads an input symbol and then calls a function that branches out in a tree of if-then-else statements to decide how to update its internal state and which output symbol to print. Depending on the chosen challenge track, for each program either a set of LTL formulas or reachability assertions is provided that are to be checked.

From the point of view of control flow the RERS Challenge problems are almost ideal in the sense that disregarding the inner state, they are flat systems without nested loops, recursion, etc. The sequential LTL problem track in the RERS Challenge 2017 contains one problem, Problem 1, that looked like the best fit to evaluate `flat-checker`. The problem contains assignments to integer variables as the state update, which was the reason to implement counter resets in `flat-checker` to preserve flatness of the system. The source file of the problem consists of ≈ 400 lines of code.

The next problem of the same complexity, Problem 4, has already a size of ≈ 9000 lines of code and is not feasible with `flat-checker` using restricted time and hardware resources, hence a solution was not attempted.

Unfortunately the next complexity category of the RERS problems that has a feasible size already contains multiplication and division, which can not be handled by `flat-checker`. In [SIN+13], where the problem generation algorithm for the RERS Challenges is presented, it is claimed that at some point a public web-service shall appear that allows for generating problem sets that can be customized in the complexity of the internal state updates. Unfortunately, no such web service or downloadable framework is available yet. It would be interesting to generate similar benchmark problems that use arithmetic state updates that include only the features

supported in the counter systems for `flat-checker`, i.e. resets, increments and decrements.

5.3.1 Modelling the Counter System

The code of Problem 1 of the RERS Challenge 2017 has been transformed to a suitable counter system for `flat-checker` by hand. While an automatic conversion tool based on the AST of the source code could be written in principle, this would have introduced a layer of unnecessary complexity and another source of mistakes into this task. The manual transformation was straight-forward due to the regularity and simplicity of the code and the humanly manageable problem size.

For each input and output symbol, a proposition was introduced that holds exactly in the moment when it is read or written. The input symbol is also used by the program to determine the next state and is represented using a counter for this purpose. All global variables that determine the program state are also represented as integer counters. The conditional statements over the state are modelled by using guards over those counters to enforce the progression to the correct branch. Figure 5.3 illustrates the general approach of the transformation from C program to `flat-checker` counter system side-by-side.

5.3.2 Adapting the LTL Formulas

The 100 LTL-formulas supplied with every problem specify universal claims about possible input/output interaction sequences, i.e. the observable behaviour of the program. In the counter system described above and hence from the perspective of `flat-checker` the in- and outputs are interspersed with many states and thereby points in time between the in- and output propositions. These must be ignored to reflect the correct meaning of the formulas.

A solution consists of using the additionally introduced `inout` proposition that holds in every step where an in- or output symbol is produced. Each formula Φ was first made syntactically compatible with `flat-checker` and then semantically

```
...

//global state variables
int a52=9; int a62=32; ...
int main() {

  //main loop:
  while(1) {
    int i; scanf("%d", &i);
    if((i != 1) ... && (i != 5))
      return -2;
    calculate_output(i);
  }
}

void calculate_output(int i) {
  if (a166 == 33) {
    if      (a176 == 6)
     calculate_outputm1(i);
     ...
  }
  ...
}
...
void calculate_outputm1(int i) {
  if(a166 == 33 && i==3) {
    a166 = 32;
    a52  = 11;
    printf("%d\n", 25);
    fflush(stdout);
  }
  ...
}
...
```

```
digraph {
0[label="init"];
1[label="main"];
0 -> 1[updates="a52:=9, a62:=32, ..."];

2[props="inout,ia",label="in=A(1)"];
3[props="inout,ib",label="in=B(2)"];
...
6[props="inout,ie",label="in=E(5)"];
7[props="inout,os",label="out=S(19)"];
8[props="inout,ot",label="out=T(20)"];
...
13[props="inout,oy",label="out=Y(25)"];
14[props="inout,oz",label="out=Z(26)"];

//entry state for calculate_output
15 [label="calc_output"];

// main loop:
// guess valid input symbol
1 -> 2[updates="i:=1"]; ...
1 -> 6[updates="i:=5"];
//jump to calculate_output
2 -> 15; 3 -> 15; ...; 6 -> 15;
// backjumps output -> next iteration
7 -> 1; 8 -> 1; ...; 13 -> 1; 14 -> 1;

// entry states for subfunctions
16; 17[label="m1"]; 18[label="m2"];
...  40[label="m20"]; 41[label="m21"];


15 -> 16[guards="[a166=33]"];
  16 -> 17[guards="[a176=6]"];
...


17 -> 42;
  42 -> 13[guards="[i=3]",
           updates="a52:=11,a166:=32"];
...
```

Figure 5.3: Aspects of the encoding of the C program source code as a counter system in DOT-format that is used with `flat-checker`.

transformed to $\hat{\Phi}$ in the following way to skip over the positions where no interaction takes place:

$$\begin{aligned}
\hat{\Phi} &= \neg\mathsf{inout}\mathbf{U}(\mathsf{inout} \wedge \mathsf{transform}(\Phi)) \\
\mathsf{transform}(\mathbf{X}\varphi) &= \mathbf{X}(\neg\mathsf{inout}\mathbf{U}(\mathsf{inout} \wedge \mathsf{transform}(\varphi))) \\
\mathsf{transform}(\varphi\mathbf{U}\psi) &= (\neg\mathsf{inout} \vee \mathsf{transform}(\varphi))\mathbf{U}(\mathsf{inout} \wedge \mathsf{transform}(\varphi)) \\
\mathsf{transform}(\varphi \wedge \psi) &= \mathsf{transform}(\varphi) \wedge \mathsf{transform}(\psi) \\
\mathsf{transform}(\varphi \vee \psi) &= \mathsf{transform}(\varphi) \vee \mathsf{transform}(\psi) \\
\mathsf{transform}(\neg\varphi) &= \neg\mathsf{transform}(\varphi) \\
\mathsf{transform}(\varphi) &= \varphi \quad \text{otherwise}
\end{aligned}$$

5.3.3 Setting the Other Parameters

As flat model-checking is in general an incomplete method, the success depends heavily on the parameters discussed in the last chapter—the path schema size n and the list of supplied loop lengths L.

To decide on a good upper bound value for n, the RERS Challenge 2016 solutions for the according problem type were consulted. For each falsified formula there was given a counterexample sequence in the form of a non-repeated prefix and an infinitely repeated suffix, for example:

```
Formula: (false R (iE & (! ! oY | ((! iA | (! oY U (oV & ! oY))) WU oY))))
"output V responds to input A after input E until output Y"
Formula is not satisfied! An error path is
[iB, oY] ([iC, oZ, iC, oW, iA, oY, iD, oY, iC, oW, iE, oY])*
---------------
```

Both parts consisted in any case of no more than about 7 input-output pairs that make up a program iteration cycle, in fact most formulas have counterexample sizes much below this upper bound. After the initial state each program iteration requires 7 steps in the modelled counter system and the sequence that is to be iterated as a loop needs two left unrollings. Hence, to represent all presented counterexamples, one would need at most $1 + 7 \cdot 7 + 3 \cdot 7 \cdot 7 = 197$ positions in the path schema. Therefore, $n = 200$ was chosen as a sufficiently high upper bound.

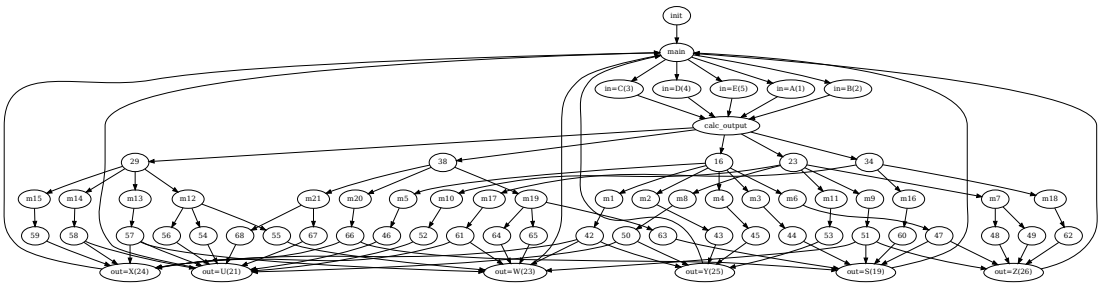


Figure 5.4: Graph rendered from the hand-crafted DOT-file representing the behaviour of the program of Problem 1 of the RERS 2017 Challenge. One can see the diamond construction for the non-deterministic choice of the next input symbol and the conditional branching tree leading to the output symbol nodes, which finally lead back to the beginning for the next iteration of the program loop after exactly 7 steps.

This problem could not be solved by `flat-checker`, when restricted to the automatically deduced simple loop lengths, namely $L = \{2, 7\}$. The reason for this is that what is needed is not a representation of multiple identical iterations of the main loop (this would only work for cases where the internal state does not change), but rather a longer loop that consists of multiple iterations of the program loop. Hence, multiples of the main program loop size are required. These were provided by hand as $L = \{2, 7, 14, 21, 28, 35, 49, 56, 63, 70, 77, 84, 91, 98\}$. This allows for encoding a looping sequence of 14 input-output pairs, which should be more than enough, given the results for the corresponding problem in 2016.

5.3.4 Results

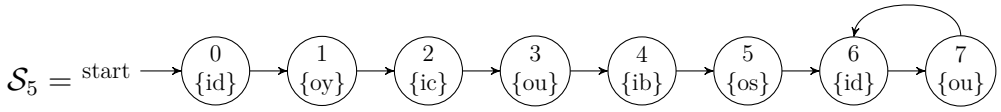
After the discussed preparations, `flat-checker` was finally used to search for counterexamples of the formulas (i.e., the negation of the formulas was given as input). The runs returned by `flat-checker` were post-processed to obtain the input-output sequence violating a formula from the run representation by filtering for the according states that represent different inputs and outputs. To rule out possible mistakes in the counter system model or bugs in `flat-checker`, all runs were reproduced in the actual unmodified program, by supplying it with the inputs from the counterexamples and comparing the outputs.

#	hh:mm:ss	$\mathcal{S} \stackrel{?}{\models} \varphi$	#	hh:mm:ss	$\mathcal{S} \stackrel{?}{\models} \varphi$	#	hh:mm:ss	$\mathcal{S} \stackrel{?}{\models} \varphi$
0	01:42:19	?	33	00:02:04	**no**	66	00:02:57	**no**
1	01:48:30	?	34	00:01:43	**no**	67	04:02:07	?
2	00:26:21	**no**	35	04:09:02	?	68	01:52:45	?
3	00:02:32	**no**	36	01:32:09	?	69	01:24:18	?
4	00:02:50	**no**	37	00:30:55	**no**	70	01:52:45	?
5	00:01:26	**no**	38	00:02:27	**no**	71	04:34:41	?
6	00:15:08	?	39	00:02:27	**no**	72	04:15:46	?
7	01:44:26	?	40	01:33:45	?	73	00:01:33	**no**
8	00:13:19	?	41	01:56:05	?	74	01:33:33	?
9	04:53:19	?	42	00:31:09	**no**	75	00:01:54	**no**
10	00:30:36	?	43	01:34:25	?	76	01:47:36	?
11	04:03:13	?	44	00:01:42	**no**	77	00:14:48	?
12	00:02:40	**no**	45	06:15:39	?	78	04:07:20	?
13	00:15:57	?	46	00:25:25	?	79	01:51:41	?
14	01:03:34	?	47	00:20:14	**no**	80	01:50:58	?
15	00:02:35	**no**	48	00:01:53	**no**	81	00:13:32	**no**
16	02:15:22	?	49	00:01:37	**no**	82	01:28:36	?
17	00:10:49	**no**	50	00:01:56	**no**	83	01:18:28	?
18	00:02:45	**no**	51	00:15:10	?	84	00:01:42	**no**
19	00:02:19	**no**	52	00:15:47	?	85	02:17:39	?
20	00:26:27	**no**	53	00:01:52	**no**	86	00:24:04	?
21	01:44:25	?	54	00:02:19	**no**	87	00:01:49	**no**
22	02:05:54	?	55	00:19:48	?	88	00:02:03	**no**
23	00:01:42	**no**	56	00:02:19	**no**	89	00:01:46	**no**
24	00:11:27	?	57	00:01:40	**no**	90	00:01:53	**no**
25	00:01:48	**no**	58	00:02:20	**no**	91	00:02:16	**no**
26	00:02:30	**no**	59	00:01:51	**no**	92	00:21:18	?
27	00:31:59	**no**	60	01:43:54	?	93	00:02:33	**no**
28	04:12:35	?	61	00:01:46	**no**	94	00:27:15	**no**
29	00:02:00	**no**	62	00:01:48	**no**	95	01:46:10	?
30	04:00:31	?	63	00:02:26	**no**	96	00:15:31	?
31	00:02:14	**no**	64	00:02:19	**no**	97	01:13:10	?
32	00:01:54	**no**	65	00:02:13	**no**	98	00:01:49	**no**
33	00:02:04	**no**	66	00:02:57	**no**	99	01:29:54	?
34	00:01:43	**no**	67	04:02:07	?			

Total time: 4d 0h 13m 45s **# Falsified:** 52

Figure 5.5: Obtained results for RERS 2017 Problem 1. "No" means that a formula has been falsified, "?" means that the limit was reached without success and probably the formula is satisfied.

$\varphi_5 = \mathbf{G}(\mathsf{oy} \wedge (\mathsf{ow} \vee (\mathbf{G}\neg\mathsf{ou} \vee (\neg\mathsf{ou}\mathbf{U}(\mathsf{ox} \vee \mathsf{ow})))))$ $\quad w_5 = \mathsf{id}, \mathsf{oy}, \mathsf{ic}, \mathsf{ou}, \mathsf{ib}, \mathsf{os}, (\mathsf{id}, \mathsf{ou})^\omega$



Promela code:

```
//Alphabet:
byte ib=1; byte ic=2; byte id=3;
byte os=4; byte ou=5; byte ow=6; byte ox=7; byte oy=8;
// w = (["id","oy","ic","ou","ib","os"],["id","ou"])
byte _prop = id; //Initial state
active proctype runproc() {
        _prop = oy;
        _prop = ic;
        _prop = ou;
        _prop = ib;
        _prop = os;
        do :: {
        _prop = id;
        _prop = ou;
        } od
}
// formula = G(oy&((G~ou|(~ouU(ow|ox)))|ow))
ltl formula {([]((_prop==oy) && ((([](!(_prop==ou)))
  || ((!(_prop==ou))U((_prop==ow) || (_prop==ox)))) || (_prop==ow))))}
```

Figure 5.6: Formula φ_5 and a violating sequence candidate w_5, transformed to a Promela program that represents the corresponding counter system $\mathcal{S}_5$ to be checked with SPIN.

Figure 5.5 presents a summary of the obtained results. After a total running time of four days, 52 of the 100 formulas have been successfully falsified. Of the falsified formulas, 43 were falsified after less than 200 seconds for each formula, using a path schema of the size 64. The other 9 were falsified after at most 32 minutes for each formula, using a path schema of size 128. The longest non-repeated prefix of an obtained counterexample run consists of 7 input-output pairs and the longest infinitely repeated suffix consists of 5 input-output pairs.

This means that most of the computation time was spent on the formulas that are probably satisfied, trying to find counterexamples with the specified upper bound of 200. No actually obtained counterexample required this path schema size, affirming the choice of $n = 200$ as probably high enough. To be even more sure, the non-

falsified formulas have been checked again with a fixed schema size of $n = 253$ and this yielded no new counterexamples after an additional calculation time of 6 days and 8 hours.

To ensure that the formulas are indeed violated by those program runs, a verification of the validity of the obtained counterexamples was attempted using the well-known model-checker SPIN [Hol97]. This was done by transforming the counterexample sequences into simple programs in Promela, the language used in SPIN, and checking the accordingly modified LTL formulas on them.

Figure 5.6 shows a Promela program used for verification with SPIN that is equivalent to formula φ_5 and the counterexample sequence w_5 obtained from the counter system representing the original program. Spin quickly verified 41 of the 52 counterexamples, another 7 were verified within about 15 minutes and the remaining 4 counterexamples did not seem to terminate and finally were aborted after 24 hours. This varying behaviour is surprising, as the resulting Promela programs are just a sequence of variable assignments that correspond to the input and output symbols and the non-terminating cases do not seem to look any more complex than the cases that took mere seconds.

Because of this not quite satisfactory behaviour of SPIN, the counterexamples additionally were checked with `flat-checker` itself, but now checking the unmodified formulas (i.e. without the transformation using the inout-proposition) on a minimal counter system just encoding the counterexample sequence, i.e. for formula φ_5 and counterexample w_5 in figure 5.6, `flat-checker` was applied again on the counter system $\mathcal{S}_5$. Using such a simplified minimal representation, `flat-checker` successfully verified each counterexample in seconds. Finally the results were submitted to the RERS Challenge 2017.

The results showed that `flat-checker` correctly identified all falsifiable formulas for the considered problem and thereby earned a bronze medal for the plain LTL category.

6 Discussion

In this thesis we have seen a new approach to existential model-checking that builds on the notion of flat underapproximations. We have seen lcLTL, an extension of LTL with optional linear constraints over the satisfying positions for different subformulas. We have learned about the relationship of lcLTL to other related logics, fLTL and CCTL*. After understanding the shortcomings of bounded model-checking applied to logics with counting constraints and learning how path schemas can help keeping the witness representation small, we have seen how this approach can be implemented by translation to satisfiability of quantifier-free Presburger arithmetic. After a discussion of the implemented prototype that uses this approach, `flat-checker`, we have seen it perform on a selection of different test cases. One of those test cases was part of the RERS Challenge 2017, where `flat-checker` was able to participate successfully in a selected category, using its direct support for counters effectively.

6.1 Critical Reflection

This thesis shows that the described approach does indeed work, both theoretically and practically, if the underlying conditions are met. Unfortunately this seems to be rarely the case and appropriate realistic inputs are hard to come by. Especially the new $\mathbf{U}[\ldots]$ operator, while more convenient and powerful that the $\mathbf{U}^{\frac{m}{n}}$ operator of fLTL, still has a rather limited utility in practical applications. Apart from that, it is difficult to find programs that can be modelled using just the supported capabilities of counter systems (i.e. just constant counter updates) without breaking the flatness, e.g. to be able to solve the first problem of the RERS Challenge it was necessary to extend `flat-checker` with counter resets. Also, without a restriction of possible loop lengths and a modelling that lends itself for compactification in terms of path schemas the approach basically degenerates into a variant of bounded model-checking.

The reliance on an external SMT-solver makes the runtime more or less unpredictable, as could be observed in one of the test cases (of course this drawback applies to all SMT-based approaches). Even when unexpected time spikes do not happen, Z3 does seem to exhibit a polynomial time increase in the size of the path schema and extrapolating from the observed times in chapter 5, `flat-checker` probably can not scale to counter systems larger than maybe a few hundred nodes under realistic constraints on the computation resources.

Nevertheless, in cases where `flat-checker` *is* applicable, it works well and in some circumstances can be even faster than sophisticated tools like SPIN, as seen in the verification of the minimal counterexample systems for the RERS Challenge.

6.2 Outlook

There are multiple directions in which this work could be developed. On the theoretical side, it is left to prove formally (as sketched in chapter 2) that the complexity and satisfiability results of fLTL are indeed applicable to lcLTL. Additionally, it would be interesting to find other counting extensions or operators for LTL that can be used in model-checking with the approach developed in this thesis. Especially finding new viable extensions regarding the capabilities of the counter system could make the approach applicable to a greater range of problems.

On the practical side, it might be possible to improve the performance of the approach, if custom solving strategies for the SMT solver are implemented. If it was possible to make Z3 aware of concepts like loop unrollings by applying some kind of pattern-matching on the formula, it might lead to faster solving times. Maybe the idea of using flat underapproximations even could be separated from the SMT-approach and incorporated in some other tool as a subroutine that is used in appropriate circumstances. For example, maybe the theory could be applied on partial paths in the modular framework *CPAChecker* [BK11] to detect spurious counterexamples.

Regarding the translation to quantifier-free Presburger arithmetic, there is in fact an unexplored alternative encoding to represent loops and their required unrollings. Instead of unrolling the loops horizontally, i.e. on the left and on the right of the loop, one could encode a vertical unrolling, i.e. that all copies of the loop are located

above each other. A propagation row could forward the values after one iteration back to the beginning of the loop in the next unrolling. Aside of requiring a more complex formula, the probably biggest drawback of this encoding approach is that the number of necessary variables for each position increases by about a factor of 4, possibly further damaging scalability, as every position must be able to encode both left unrollings, the right unrolling and the actual loop position, together with at least the necessary labels and counters for $\mathbf{U}[\ldots]$ subformulas, etc. So while the implemented encoding in some sense is one dimensional and can use positions flexibly to encode loops as necessary, the vertical approach is two dimensional and rigid, due to the enforced layering of loop positions.

A reason to try this approach nevertheless is that it elegantly resolves the problem of enforcing the consistency of the loop unrollings. In the vertical encoding, all possible copies of the loop sequence are local (i.e. above each other) and hence no loop length calculation and no blowup due to possible loop lengths is required. This would completely eliminate the need for a list of allowed loop lengths and allow for completely arbitrary loop sequences without apriori knowledge about the underlying counter system, allowing for grouping loop iteration patterns into larger loops without resorting to manual interventions like in the application to the RERS challenge problem.

Unfortunately the idea for this alternative loop encoding came late in the development process of both this thesis and the prototype implementation and had to be discarded due to its questionable practical scalability.

Bibliography

[AMP06] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. *Bounded Model Checking of Software Using SMT Solvers Instead of SAT Solvers*, pages 146–162. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[Ape66] Harry Apelt. Axiomatische untersuchungen über einige mit der presburgerschen arithmetik verwandte systeme. *Mathematical Logic Quarterly*, 12(1):131–168, 1966.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, 1999.

[BCD+11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[BCM+92] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 1020 states and beyond. *Information and computation*, 98(2):142–170, 1992.

[BDL12] B. Bollig, N. Decker, and M. Leucker. Frequency linear-time temporal logic. In *2012 Sixth International Symposium on Theoretical Aspects of Software Engineering*, pages 85–92, July 2012.

[BFLS05] Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Philippe Schnoebelen. Flat acceleration in symbolic model checking. In *International Symposium on Automated Technology for Verification and Analysis*, pages 474–488. Springer, 2005.

[BHMR07] Dirk Beyer, Thomas A Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In *Acm Sigplan Notices*, volume 42, pages 300–309. ACM, 2007.

[BK11] Dirk Beyer and M Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In *International Conference on Computer Aided Verification*, pages 184–190. Springer, 2011.

[Bra13] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05):552–593, 2013.

[BT76] Itshak Borosh and Leon Bruce Treybig. Bounds on positive integral solutions of linear diophantine equations. *Proceedings of the American Mathematical Society*, 55(2):299–304, 1976.

[CBRZ01] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[CDW14] David R. Cok, David Déharbe, and Tjark Weber. The 2014 SMT competition. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:207–242, 2014.

[CE82] Edmund Clarke and E Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Logics of programs*, pages 52–71, 1982.

[CFLZ08] Nicolas Caniart, Emmanuel Fleury, Jérôme Leroux, and Marc Zeitoun. Accelerating interpolation-based model-checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 428–442. Springer, 2008.

[CFM09] L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. *ArXiv e-prints*, July 2009.

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer aided verification*, pages 154–169. Springer, 2000.

[CGP99] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and computation*, 76(2-3):95–120, 1988.

[DDS15] Stéphane Demri, Amit Kumar Dhar, and Arnaud Sangnier. Taming past ltl and flat counter systems. *Information and Computation*, 242:306–339, 2015.

[DFH+04] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Benjamin Werner, and Christine Paulin-Mohring. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[DHL+17] Normann Decker, Peter Habermehl, Martin Leucker, Arnaud Sangnier, and Daniel Thoma. Model-checking counting temporal logics on flat structures. In *CONCUR 2017*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[FHR13] Ylies Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. *Engineering Dependable Software Systems*, 34:141–175, 2013.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[HIK+12] Hossein Hojjat, Radu Iosif, Filip Konečnỳ, Viktor Kuncak, and Philipp Rümmer. Accelerating interpolants. In *International Symposium on Automated Technology for Verification and Analysis*, pages 187–202. Springer, 2012.

[HIM+12] Falk Howar, Malte Isberner, Maik Merten, Bernhard Steffen, and Dirk Beyer. The rers grey-box challenge 2012: analysis of event-condition-action systems. *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pages 608–614, 2012.

[HKPV91] Heinrich Herre, Michał Krynicki, Alexandr Pinus, and Jouko Väänänen. The härtig quantifier: A survey. *The Journal of Symbolic Logic*, 56(4):1153–1183, 1991.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[How80] William A. Howard. The formulae-as-types notion of construction. pages 479–490, 1980.

[Joh75] Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[JSS14] Bertrand Jeannet, Peter Schrammel, and Sriram Sankaranarayanan. Abstract acceleration of general linear loops. In *ACM SIGPLAN Notices*, volume 49, pages 529–540. ACM, 2014.

[KLW15] Daniel Kroening, Matt Lewis, and Georg Weissenbacher. Under-approximating loops in c programs for fast counterexample detection. *Formal methods in system design*, 47(1):75–92, 2015.

[KW10] Daniel Kroening and Georg Weissenbacher. Verification and falsification of programs with loops using predicate abstraction. *Formal Aspects of Computing*, 22(2):105–128, 2010.

[LMP10] François Laroussinie, Antoine Meyer, and Eudes Petonnet. Counting ltl. In *Temporal Representation and Reasoning (TIME), 2010 17th International Symposium on*, pages 51–58. IEEE, 2010.

[LMP12] François Laroussinie, Antoine Meyer, and Eudes Petonnet. Counting CTL. *Logical Methods in Computer Science*, 9(1), 2012.

[LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[MD76] Prabhaker Mateti and Narsingh Deo. On algorithms for enumerating all circuits of a graph. *SIAM Journal on Computing*, 5(1):90–99, 1976.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*, volume 32. Chalmers University of Technology, 2007.

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[Pre29] M. Presburger. Über die vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. *Comptes Rendus du Premier Congrès Mathématicienes des Pays Slaves*, pages 92–101, 1929.

[SIN+13] Bernhard Steffen, Malte Isberner, Stefan Naujokat, Tiziana Margaria, and Maren Geske. Property-driven benchmark generation. In *International SPIN Workshop on Model Checking of Software*, pages 341–357. Springer, 2013.

[Val91] Antti Valmari. Stubborn sets for reduced state space generation. *Advances in Petri Nets 1990*, pages 491–515, 1991.