國立臺灣大學管理學院資訊管理學研究所

博士論文

Department of Information Management
College of Management
National Taiwan University

Doctoral Dissertation

基於自動機理論的模型檢測演算法與工具之改善

Improved Algorithms and Tools for Automata-Theoretic Model Checking

蔡明憲

Ming-Hsien Tsai

指導教授：蔡益坤 博士、王柏堯 博士

Advisers: Yih-Kuen Tsay, Ph.D., Bow-Yaw Wang, Ph.D

中華民國102年7月

July 2013

論文摘要

以自動機理論爲基礎的模型檢測方法已經發展近三十年，此方法被認爲具有效率且容易使用，在確保軟體或硬體設計的正確性上，已成爲業界測試與模擬之外的另一項選擇。在這個方法中，一個待測的系統會用一個Büchi自動機來表示，而系統所需滿足的規格則用一個線性時態邏輯（PTL）式來表示。在線性時態邏輯中，比較常用到的是只能描述未來的版本（LTL）。此方法首先將規格的否定轉換成一個Büchi自動機，然後和表示系統的自動機作交集，最後測試交集部分是否不接受任何行爲。在這些步驟中，時態邏輯式的轉換扮演一個重要的角色，因爲一般來說，轉換出來的自動機越小，則與系統的交集也越小，而模型檢測可以比較快完成。雖然目前已經有很多LTL轉換演算法，然而在其他時態邏輯的部分，仍有改善的空間，例如比LTL更容易書寫較短規格的PTL。

系統規格也可以直接用Büchi自動機來表示，在某些情況下會比時態邏輯式更自然與直接。如此一來，在與系統自動機作交集之前，需要先計算規格自動機的補集。Büchi自動機的補集計算與有限字串長度的傳統自動機相比更加複雜許多，所以一些最佳化方法對補集計算的效能與效率有很大的幫助。因爲Büchi 自動機補集的高複雜度，最近許多研究都跳往不需完整建置補集的漸進式行爲包含測試。在行爲包含測試中，以Ramsey理論爲基礎的方法已被證明相當有效率，不過在補集計算的部分卻相當不具競爭力，反而確定性方法是最好的。同樣地，爲了加速模型檢測，我們除了改進時態邏輯轉換演算法之外，也可以改進補集計算與包含測試的演算法。

在時態邏輯轉換的部分，我們改善了五個漸進式演算法，利用一個回朔的程序讓這些演算法可以支援過去時態運算子，同時保持漸進式轉換的好處。對於狀態基礎演算法中的涵蓋計算程序，我們也利用質含項（Prime Implicant）加以改善。此外，我們也實作了過去與未來的分離程序，使得一個PTL式子可以被轉換成另一個相等的LTL式子，然後再利用目前最有效率的LTL轉換演算法（例如LTL2BA或是LTL3BA）來轉換。這使得我們可以比較我們所改善的演算法，以及目前最有效率的LTL轉換演算法。

在Büchi補集計算的部分，我們審視四個主要方法，並透過實驗加以比較。雖然傳統的看法是非確定性方法比確定性方法更好，因爲非確定性方法有比較低的理論複雜度。然而我們的實驗顯示在Büchi補集計算中，確定性方法是最好的。此外，我們也對其中三個方法提出數個改善的點子，使得這三個方法的效率與效能大爲改善。

在包含測試中，我們提出一個基於確定性方法的漸進式包含測試演算法。雖然確定性方法的執行一般被認爲需要分離成幾個階段，不過我們展示這些階段其實可以合併成一個，而且可以被漸進式地執行。實驗顯示包含關係成立時，我們的方法比以Ramsey理論爲基礎的方法來得好。不過包含關係不成立時，以Ramsey理論爲基礎的方法卻比我們的方法來得好。

最後，我們要應付的問題是自動機與時態邏輯在教育與研究的工具支援。雖然目前已有許多以自動機理論爲基礎的模型檢測工具，不過都沒有支援自動機或時態邏輯式視覺化呈現與操作。這激發了我們的第一個工具：GOAL，第一個可以用來學習無限自動機與時態邏輯的視覺化互動工具。除此之外，GOAL也提供功能幫助研究學者開發和測試新演算法、執行實驗、與收集統計資料。除了GOAL以外，我們還建制一個網頁形式的工具：Büchi Store。這個工具是許多常見時態邏輯式與其最小相等自動機的集合，可以被搜尋、瀏覽與擴展。這樣的集合不僅可以當成一個最佳時態邏輯式轉換演算法，同時也可以作爲實驗的比較基礎。GOAL與Büchi Store已經幫助我們開發許多改進的演算法，讓模型檢測更

加快速。我們相信這兩個工具可以提供研究學者一個好的平台，幫助未來開發相關演算法與進行實驗。

ABSTRACT

Improved Algorithms and Tools for Automata-Theoretic Model Checking

by Ming-Hsien Tsai
Graduate Institute of Information Management
National Taiwan University
Advisers: Yih-Kuen Tsay, Ph.D. and Bow-Yaw Wang, Ph.D.

The automata-theoretic approach to model checking has been developed for near three decades. Because of its proven effectiveness and ease of use, the approach has become a viable alternative to testing and simulation in industry. In the approach, a system is typically represented by a Büchi automaton, while a specification is encoded by a formula in propositional linear temporal logic (PTL), of which the future fragment (usually referred to as LTL) is more often used. The approach proceeds by translating the negation of the formula to a Büchi automaton, which is later intersected with the system automaton for testing emptiness. Such translation plays an important role in the approach because in general, the smaller the negated-specification automaton is, the sooner the model checking process may be completed. Although there has been a long line of research on LTL translation algorithms aiming to produce smaller automata, there are still opportunities of improving translation algorithms for other temporal logics such as PTL, which is expressively more succinct than LTL.

Specifications may also be directly encoded by Büchi automata which in certain cases are more natural and easier than temporal formulae. In such cases, complementation of a specification automaton is performed before taking the intersection with the system automaton. The complementation of Büchi automata is significantly more complicated than that of classic finite automata on finite words. Optimization heuristics are critical to good performance. Due to the high complexity of Büchi complementation, much recent emphasis has been shifted to containment testing without constructing the complement. The Ramsey-based approach has been proven to be quite effective in such containment testing, although it is not competitive in complementation where the determinization-based approach is the best in general. Again, to expedite the model checking process, we can improve not only the translation algorithm but also the complementation algorithm and the containment testing algorithm.

For translation of temporal formulae, we extend five existing incremental algorithms with a backtrace procedure to support past operators of PTL, while maintaining the advantages of incremental automata construction. The cover computation

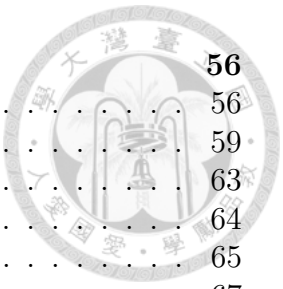
common to the state-based incremental algorithms is improved based on prime implicants to obtain smaller automata. Besides, we also implement the separation of past and future separators. The separation procedure can convert a PTL formula to an equivalent LTL formula, which can later be translated by an efficient LTL translation algorithm such as LTL2BA or LTL3BA. This allows us to compare our extended algorithms with those existing efficient LTL translation algorithms.

For Büchi complementation, we review four approaches and perform a comparative experimentation on the best construction in each approach. Although the conventional wisdom is that the nondeterministic approaches are better than the deterministic approach because of better worse-case bounds, our experimental results show that the deterministic construction is the best for complementation in general. We also propose optimization heuristics for three of the four best constructions. Our experiment shows that the optimization heuristics substantially improve the three constructions.

For containment testing, we propose an incremental containment testing approach based on the determinization-based constructions, of which Safra-Piterman construction is the best in Büchi complementation. Although the determinization-based constructions are typically performed in stages, we show that the stages can be merged such that the containment testing can be performed incrementally. Our experimental results show that for cases where the containment relation holds, our incremental Safra-Piterman approach is much better then the Ramsey-based approach. For other cases where the containment relation does not hold, the Ramsey-based outperforms our incremental Safra-Piterman approach.

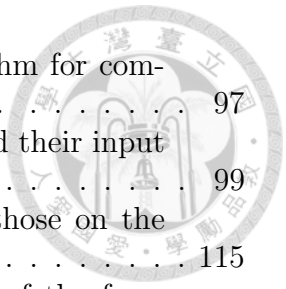
Finally, we address the issue of tool support for education and research on ω-automata and temporal logics. Although there are various tools for automata-theoretic model checking, none of the tools provide facilities for visually manipulating automata and temporal formulae. This motivated our GOAL tool, which is the first interactive graphical tool for learning ω-automata and temporal logics. Besides, GOAL also provides facilities for helping researchers develop and test new algorithms, perform experiments, and collect statistical data. We also build a Web-based tool called Büchi Store, which is an extensible collection of temporal formulae and their equivalent automata. Such a collection can be used not only as a new translation tool that always returns the smallest automata available but also as benchmark cases for experiments. Both GOAL and Büchi Store already helped us develop our improved algorithms, which expedite the model checking process. We believe the two tools provide researchers with a good environement for future development and experimentation of related algorithms.

Contents



List of Figures

List of Tables

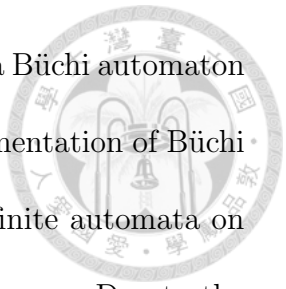


Chapter 1

Introduction

The automata-theoretic approach to model checking has been developed for near three decades [102]. Because of its proven effectiveness and ease of use, the approach has been a viable alternative to testing and simulation in industry. In the approach, a system is typically represented by a Büchi automaton M, while a specification is encoded by a formula f in propositional linear temporal logic (PTL), of which the future fragment (usually referred to as LTL) is more often used. Given a Büchi automaton B, $L(B)$ captures the behavior accepted by the automaton B. The system M satisfies the specification f if $L(M \times A_{\neg f}) = \emptyset$ where $M \times A_{\neg f}$ is the intersection of M and $A_{\neg f}$, and $A_{\neg f}$ is a Büchi automaton obtained by translating the negation of the specification formula. Such translation plays an important role in the approach because in general, the smaller the negated-specification automaton is, the sooner the model checking process may be completed. Although there has been a long line of research on LTL translation algorithms aiming to produce smaller automata, there are still opportunities of improving translation algorithms for other temporal logics such as PTL, which is expressively more succinct than LTL [62].

Specifications may also be directly encoded by Büchi automata, a type of ω-automata, which in certain cases are more natural and easier than temporal formu-



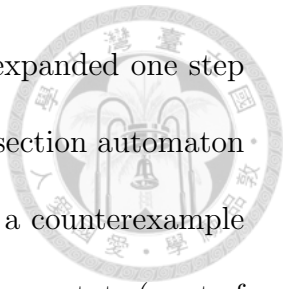
lae. In such cases, a system M satisfies a specification encoded by a Büchi automaton A if $L(M \times \overline{A}) = \emptyset$ where $\overline{A}$ is the complement of A. The complementation of Büchi automata is significantly more complicated than that of classic finite automata on finite words. Optimization heuristics are critical to good performance. Due to the high complexity of Büchi complementation, much recent emphasis has been shifted to containment testing without constructing the complement [20, 25, 3, 1]. Again, to expedite the model checking process, we can improve not only the translation algorithm but also the complementation algorithm and the containment testing algorithm.

In this thesis, we focus on improving translation of temporal formulae, complementation of Büchi automata, and incremental containment testing. We also address the issue of tool support for education and research on ω-automata and temporal logics.

1.1 Translation of Temporal Formulae

The traditional automaton construction [66] for a PTL specification formula works by building a tableau with the full state space; that is, it creates states for every maximally consistent (healthy) sets of subformulae of the specification formula. It then adds transitions according to subformulae information on each state. This kind of constructions can be easily understood, but they are clearly not suitable for practice as they reach the worst case bound every time.

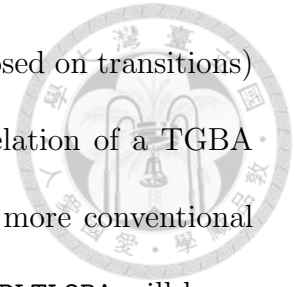
This obvious drawback motivated incremental algorithms [34, 17] which do not generate the whole tableau at once, but rather start from the initial states and build the state transition graph incrementally. The incremental generation permits truely on-the-fly model checking in that the intersection automaton (of the system and the



negated specification) is constructed from the initial states and expanded one step further only if no accepting word is found by examining the intersection automaton constructed so far. On-the-fly model checking therefore can find a counterexample early without generating the whole intersection automaton. Besides, a state (a set of subformulae) in the tableau can be represented symbolically such that the translated automaton is even smaller. Though current practice of on-the-fly model checking generates the entire specification automaton before constructing the intersection automaton, the incremental translation algorithms are still preferred, as they in general produce smaller automata. These algorithms are the kernel of temporal formulae to Büchi automata translation in the well-known model checker `Spin` [42]. Unfortunately, these incremental algorithms cannot be applied to formulae containing past operators.

It has been shown that PTL and LTL have the same expressive power [29, 30]. However, adding past operators brings additional expressivity from a practical view [64]. For example, it is easy to state that "only event q can trigger a series of event p in the next time" in PTL as $\Box(p \rightarrow (p \,\mathcal{S}\, q))$. This specification formula with a past operator is easy to understand and to write, but it is not the case in LTL. The equivalent formula in LTL is $\Box(p \vee ((\neg p) \,\mathcal{W}\, q)) \wedge ((\neg p) \,\mathcal{W}\, q)$, where $(\neg p) \,\mathcal{W}\, q$ appears twice. Moreover, in [62], it has been proven that PTL can be exponentially more succinct than LTL.

To support past operators while attempting to maintain efficiency, the two-step algorithm by Gastin and Oddoux [33] generalized the efficient translation algorithm `LTL2BA` [32]. The generalized algorithm, referred to as `PLTL2BA`, uses two-way very weak alternating co-Büchi automata (2VWAA) and transition-based generalized



Büchi automata (TGBA, where the acceptance condition is imposed on transitions) as the intermediate representations. However, the transition relation of a TGBA in `PLTL2BA` is not defined in a conventional way. If one uses a more conventional representation of automata, the size of the TGBA produced by `PLTL2BA` will be as large as $2^{n+1} \times 2^{n+1}$ states where n is the number of temporal subformulae of the input PTL formula. Besides, `PLTL2BA` has the following two drawbacks during the conversion from a 2VWAA into a TGBA. The first drawback is that the conversion is fully automaton-based, and thus it misses the chance to simplify the TGBA based on the formulae in the states of the 2VWAA. The second drawback is that the saturation loop in the conversion iterates over the whole set of states currently constructed in the TGBA. Even if a timestamp mechanism is used to avoid some redundant computations, the algorithm still spends time in checking the backward consistency between many consistent states.

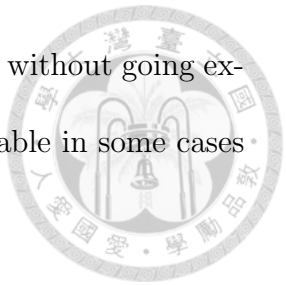
In this thesis, we propose a PTL translation algorithm, which extends the incremental algorithms with a backtrace procedure to support past operators, while maintaining the advantages of incremental automata construction. The backtrace is only performed when a backward inconsistency is found. The cover computation common to those state-based incremental algorithms is improved based on prime implicants to obtain smaller automata. We have implemented the extended incremental algorithms in the GOAL tool. Experimental results show that one of the extended algorithms produces in most cases smaller automata than other algorithms that support past operators. Besides, we have also implemented the separation of past and future operators proposed by Gabbay [29] (with adaptation since the definition of temporal operators differs). The separation procedure can convert a PTL

formula to an equivalent LTL formula, which can later be translated by an efficient LTL translation algorithm such as LTL2BA or LTL3BA. This allows us to compare our extended algorithms with those existing efficient LTL translation algorithms. The experimental results show that our extended algorithms perform much better than LTL translation algorithms plus the separation procedure.

1.2 Complementation of Büchi Automata

Büchi automata are nondeterministic finite automata on infinite words. They recognize ω-regular languages and are closed under Boolean operations, namely union, intersection, and complementation. The formalism was first proposed and studied by Büchi in 1960 as part of a decision procedure for second-order logic [8]. Complementation of Büchi automata is significantly more complicated than that of nondeterministic finite automata on finite words. Given a nondeterministic finite automaton on finite words with n states, complementation yields an automaton with 2^n states through the subset construction [75]. Indeed, for nondeterministic Büchi automata, the subset construction is insufficient for complementation. In fact, Michel showed in 1988 that blow-up of Büchi complementation is at least $n!$ (approximately $(n/e)^n$ or $(0.36n)^n$), which is much higher than 2^n [69]. This lower bound was later sharpened by Yan to $(0.76n)^n$ [103], which was matched by an upper bound by Schewe [79].

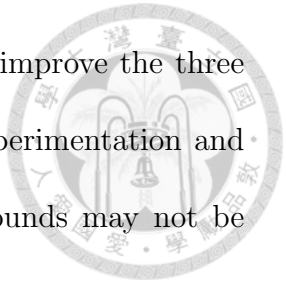
There are several applications of Büchi complementation in formal verification. For example, whether a system satisfies a property can be verified by checking if the intersection of the system automaton and the complement of the specification automaton is empty [98]. Another example is that the correctness of an LTL translation algorithm can be tested with a reference algorithm [39]. Although recently



many works have focused on universality and containment testing without going explicitly through complementation [20, 25, 3, 1], it is still unavoidable in some cases [58].

There have been quite a few complementation constructions, which can be classified into four approaches: Ramsey-based approach [8, 83], determinization-based approach [77, 70, 4, 74], rank-based approach [89, 57, 54], and slice-based approach [47, 101]. The second approach is a deterministic approach while the last two are nondeterministic approaches. The first three approaches were reviewed in [100]. Due to the high complexity of Büchi complementaton, optimization heuristics are critical to good performance [39, 27, 79, 48, 55]. However, with much recent emphasis shifted to universality and containment, empirical studies of Büchi complementation have been scarce [55, 39, 48, 94] in contrast with the rich theoretical developments. A comprehensive empirical study would allow us to evaluate the performance of these complementation approaches that has so far been characterized only by theoretical bounds.

In this thesis, we review the four complementation approaches and perform comparative experimentation on the best construction in each approach. Although the conventional wisdom is that the nondeterministic approaches are better than the deterministic approach because of better worse-case bounds, our experimental results show that the deterministic construction is the best for complementation in general. At the same time the Ramsey-based approach, which is competitive in universality and containment testing [3, 25, 26], performs rather poorly in our complementation experiments. We also propose optimization heuristics for the determinization-based construction, the rank-based construction, and the slice-based construction. Our



experiment shows that the optimization heuristics substantially improve the three constructions. Overall, our work confirms the importance of experimentation and heuristics in studying Büchi complementation, as worst-case bounds may not be accurate indicators of performance.

1.3 Incremental Containment Testing

Model checking a system against a specification can be performed in another way by testing if the behavior of the system is contained in the behavior allowed by the specification without taking the complementation of the specification. Recently, there were many works focus on universality and containment testing based on the Ramsey-based approach and the rank-based approach without going explicitly through complementation [25, 26, 21, 3, 1, 2]. Among the existing approaches, the Ramsey-based approach is shown to be quite competitive [3, 25, 26].

Although the Ramsey-based approach is competitive in universality and containment testing, it performs rather poorly in our complementation experiments, which show that the determinization-based approach is the best for complementation in general. It is then interesting to see the comparison of the determinization-based approach and the Ramsey-based approach in containment testing.

The determinization-based approach is typically performed in three stages: determinizing a Büchi automaton into a deterministic automaton with another acceptance condition, complementing the deterministic automaton, and converting the complement into a Büchi automaton. Such three-stage complementation has an advantage of applying various optimizations in each stage but is considered not suitable for incremental containment testing. In fact, the three stages of the determinization-based approach can be merged. On the one hand, this makes us able to perform

incremental containment testing based on the determinization-based approach. On the other hand, optimization heuristics for complementation may become inapplicable.

In this thesis, we propose an incremental containment testing approach based on the determinization-based approach. We also compare our determinization-based approach with the Ramsey-based approach and the naive approach that takes a full complement first, then intersection, and finally an emptiness test. The experimental results showed that the Ramsey-based approach performs much better than the incremental determinization-based approach and the naive approach in the cases where the containment relation holds but rather poorly in all other cases. The incremental determinization-based approach performs much better than the naive approach when the containment relation does not hold but worse than the naive approach when the containment relation holds.

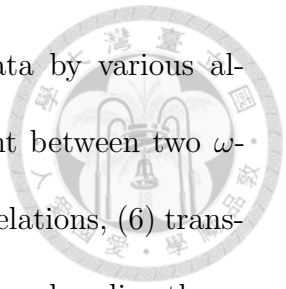
1.4 Tool Support

Besides of improved algorithms for model checking, we also address the issue of tool support for education and research on ω-automata and temporal logics. Two tools, namely GOAL and Büchi Store, are built for this purpose.

1.4.1 GOAL

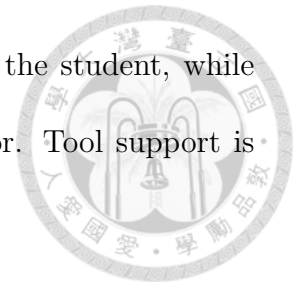
The acronym GOAL[1] was originally derived from "**G**raphical Tool for **O**mega-**A**utomata and **L**ogics". It also stands for "**G**ames, **O**mega-**A**utomata, and **L**ogics". The main functions of GOAL include (1) drawing ω-automata, alternating automata, two-way alternating automata, and games, (2) testing and converting ω-

[1]The tool is available at http://goal.im.ntu.edu.tw/.



automata, (3) determinizing and complementing Büchi automata by various algorithms, (4) checking the language equivalence and containment between two ω-automata, (5) simplifying Büchi automata by various simulation relations, (6) translating QPTL (Quantified Propositional Temporal Logic) [83] formulae directly or step-by-step into equivalent Büchi automata by various algorithms, (7) exporting Büchi automata as Promela code, (8) generating random ω-automata and PTL formulae, (9) solving reachability, Büchi, and parity games.

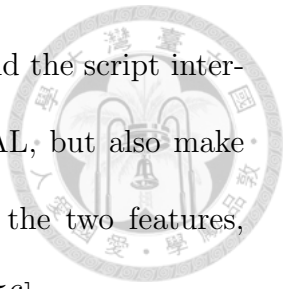
GOAL was originally designed with a graphical interface for learning and teaching Büchi automata and linear temporal logics. As shown in [83], Büchi automata and QPTL are expressively equivalent, though translation between the two formalisms is highly complex [51]. For PTL, practically feasible algorithms exist for translating a PTL formula into an equivalent Büchi automaton [49, 34, 17, 32], though not vice versa. Despite the possibility of mechanical translation, a temporal formula and its equivalent Büchi automaton are two very different artifacts and their correspondence is not easy to grasp. Temporal formulae describe temporal dependency without explicit references to time points and are in general more abstract, while Büchi automata "localize" temporal dependency to relations between states and tend to be of lower level. Understanding their relation is instrumental in discovering algorithmic solutions to model checking problems or simply in using those solutions, e.g., specifying a temporal property directly by an automaton rather than a temporal formula so that the property can be verified by an algorithm that operates on automata. To enhance this understanding, it helps to go through several translation algorithms with different input temporal formulae or simply by examining more examples of temporal formulae and their equivalent Büchi automata. This



learning process, however, is tedious and prone to mistakes for the student, while preparing the material is very time-consuming for the instructor. Tool support is needed.

Moreover, in model checking a system M against a specification f in temporal logic, as the emptiness checking of $M \times A_{\neg f}$ requires time proportional to the size of the system automaton M and to that of the specification automaton $A_{\neg f}$, a larger $A_{\neg f}$ would mean a longer verification time. To reduce verification time, it may be worthwhile to construct a smaller $A_{\neg f}$ manually. But a way for checking the correctness of a user-defined $A_{\neg f}$ is needed. With various translation algorithms and language equivalence testing implemented in GOAL, the correctness of a user-defined specification automaton can be checked against an easier-to-understand QPTL formula, by translating the specification formula into an equivalent automaton and testing the equivalence between the user-defined and the machine-translated automata. Once the specification automaton of an ideal size has been successfully constructed and checked, it can be exported as Promela code which can then be fed into the `Spin` model checker.

Besides the support for education and learning, GOAL can also support research in various aspects. For example, with the graphical interface, translation algorithms, complementation algorithms, language equivalence testing, and other utility functions implemented, GOAL can help researchers develop new algorithms with cross-checking of correctness, generation of random tests, collection of statistical data from comparison experiments, More importantly, when researchers found some results are different to their expectation, they can use GOAL to "see" what the problem is. Besides the graphical interface, GOAL also provides a command-line



interface and a script interpreter. The command-line interface and the script interpreter not only make experiments easy to be performed in GOAL, but also make other tools easy to access functions provided by GOAL. With the two features, GOAL has been used by other researchers in their work [11, 12, 56].

To the best of our knowledge, GOAL is the first graphical interactive tool designed for ω-automata and temporal logics. There are other tools that provide translation of LTL into Büchi automata, e.g., `Spin` [42], LTL2BA [32], Wring [86], MoDeLLa [81], and LTL2Buchi [35]. `Spin` in particular is an automata-theoretic model checker that has been widely used both in practice and in education. None of these tools provide facilities for visually manipulating automata and the temporal logics they support are less expressive than QPTL supported by GOAL. GOAL also supports past temporal operators which make some specifications easier to write. The operations and tests on Büchi automata provided by GOAL are also more comprehensive than those by other tools.

1.4.2 Büchi Store

To apply the automata-theoretic approach, an algorithm for translating a temporal formula into an equivalent Büchi automaton is essential. There has been a long line of research on such translation algorithms, aiming to produce smaller automata. According to our experiments, none of the proposed algorithms outperforms the others for every temporal formula tested. Table 7.5 shows a comparison of some of the well-known translation algorithms for selected temporal formulae. From the table, we observe that, although one algorithm (LTL2BA) is better than the other algorithms in most cases, quite a few of the automata that it produces are still larger than the best ones that we know (in the second column of the table). Following [13],

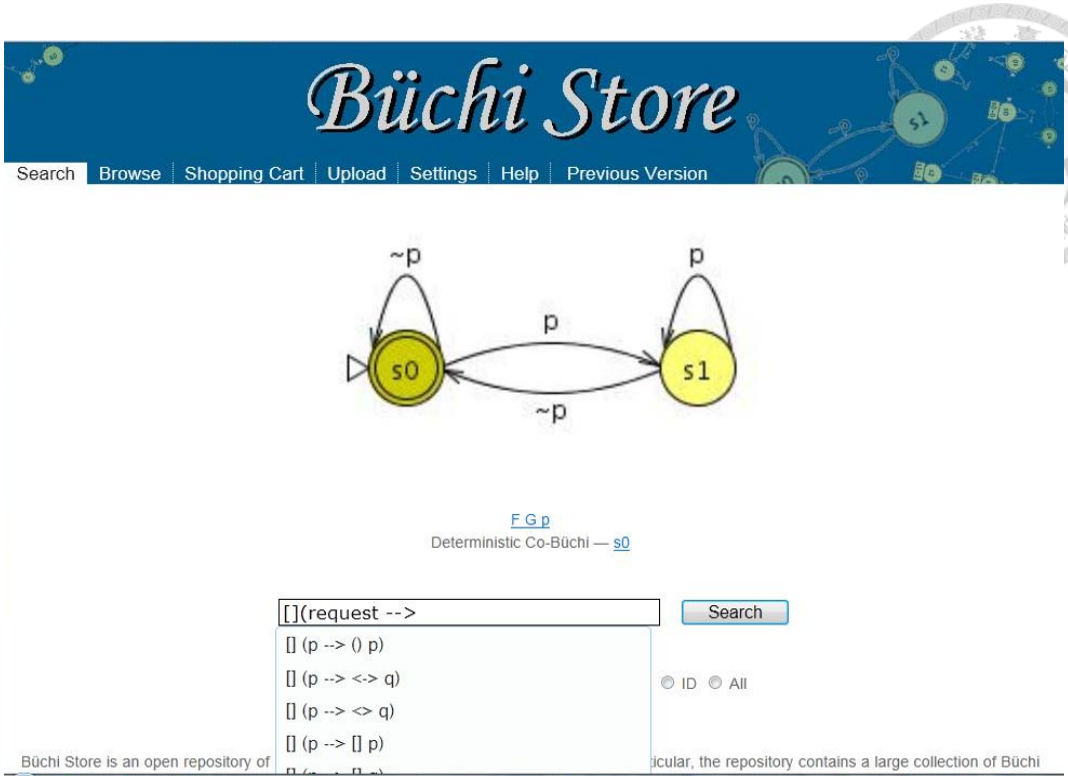
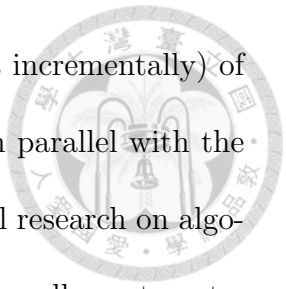


Figure 1.1: Main page of Büchi Store and search of $\Box$(request $\rightarrow \Diamond$response) in progress.

we also experimented with temporal formulae of the form $\Diamond(p_1 \wedge \Diamond(p_2 \wedge \Diamond(\cdots \wedge \Diamond(p_{n-1} \wedge \Diamond p_n)\cdots))) \wedge \Diamond(q_1 \wedge \Diamond(q_2 \wedge \Diamond(\cdots \wedge \Diamond(q_{n-1} \wedge \Diamond q_n)\cdots)))$ as shown in Table 7.4. In this experiment, `Spin` is the best, but again many of the results may be further improved.

Given that smaller automata usually expedite the model-checking process, it is certainly desirable that one is always guaranteed to get the smallest possible automaton for (the negation of) a specification formula. One way to provide the guarantee is to try all algorithms or even manual constructions and take the best result. This simple-minded technique turns out to be feasible, as most specifications use formulae of the same patterns [22] and the tedious work of trying all alternatives needs only be done once for a particular pattern instance.

Sometimes it may be hard, if not impossible (using quantification over propositions), to give the specification as a temporal formula. When the specification is

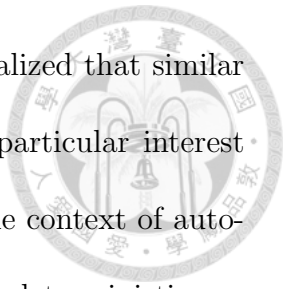


given directly as an automaton, taking the complement (perhaps incrementally) of the specification automaton becomes necessary. Consequently, in parallel with the research on translation algorithms, there has also been substantial research on algorithms for Büchi complementation. The aim again is to produce smaller automata.

Several Büchi complementation algorithms have been proposed that achieve the lower bound of $2^{\Omega(n \log n)}$ [69]. However, the performances of these "optimal" algorithms differ from case to case, sometimes quite dramatically. Table 7.6 shows a comparison of some of the complementation algorithms for selected Büchi automata (identified by equivalent temporal formulae). From the table, we again observe that, although one algorithm (Rank-Based) is better than the other algorithms in most cases, many of the results that it produces are still larger (some much larger) than the known best ones. In the literature, evaluations of these algorithms usually stop at a theoretical-analysis level, partly due to the lack of or inaccessibility to actual implementations. This may be remedied if a suitable set of benchmark cases becomes available and subsequent evaluations are conducted using the same benchmark.

Büchi Store was thus motivated and implemented as a website[2], whose main page is shown in Figure 1.1. One advantage for the Store to be on the Web is that the user always gets the most recent collection of automata. Another advantage is that it is easily made open for the user to contribute better (smaller) automata. In the current implementation of the Store, automata are grouped into equivalence classes induced by the languages that they recognize. The current collection contains over two hundred equivalence classes. Most automata are associated with their equivalent temporal formulae. Additional properties such as classification into the Manna-Pnueli temporal hierarchy [65] are also provided.

[2]URL of Büchi Store: `http://buchi.im.ntu.edu.tw/`.



With the collection of Büchi automata as a start, we soon realized that similar collections are also desired for other types of ω-automata. Of particular interest are deterministic Büchi and deterministic parity automata. In the context of automatic synthesis of reactive systems from temporal specifications, deterministic automata are useful for battling high complexity [74, 85]. Though deterministic Büchi (word) automata (DBWs) are less expressive than nondeterministic Büchi automata (NBWs) for encoding temporal properties, DBWs wherever adequate are favored for they allow more efficient synthesis algorithms and produce better results. DBWs are also useful in the classification of temporal properties. Deterministic parity automata (DPWs), on the other hand, are as expressive as NBWs. When used for synthesis, DPWs induce two-player games with the parity condition that are easier to solve than games induced by other types of deterministic ω-automata. However, algorithms for obtaining deterministic automata introduce exponential blow-ups in the number of states and their performances vary for different inputs. Büchi Store can be a good source of small deterministic automata for common temporal properties.

1.5 Overview

This thesis proposal is organized as the following.

Chapter 2 shows related work on temporal formulae translation, Büchi complementation, and containment testing.

Chapter 3 gives the preliminaries where PTL is introduced in Section 3.2, ω-automata in Section 3.3, and the Manna-Pnueli temporal hierarchy in Section 3.4.

Chapter 4 introduces the proposed incremental translation algorithm for full PTL formulae. The prerequisites are Sections 3.2 and 3.3.

Chapter 5 introduces the optimization heuristics for Büchi complementation algorithms and a comparative experimentation on the best construction in each complementation approach. The prerequisite is Section 3.3.

Chapter 6 introduces an on-the-fly containment testing based on the determinization-based constructions. The prerequisites are Section 3.3 and the improved Safra-Piterman construction introduced in Chapter 5.

Chapter 7 introduces the tools GOAL and Büchi Store. The prerequisites are Sections 3.2, 3.3, and 3.4.

Chapter 8 concludes and describes the future work.

Chapter 2

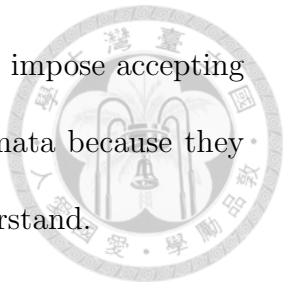
Related Work

We review in this chapter related work on translation of propositional temporal formulae, complementation of Büchi automata, and containment testing of Büchi automata.

2.1 Translation of Propositional Temporal Formulae

The traditional automaton construction for a PTL specification works by creating states for every maximally consistent (healthy) sets of subformulae of the specification. For example, the algorithm in [66], referred to as `Tableau` here, expands formula φ to compute the closure of φ, which is then combined with classification rules to create atoms. The edges between atoms are created by the information on every pair of atoms. Next, the accepting states are set according to the promising formulae and fulfilling requirements. Finally, the constructed temporal tableau is converted into a Büchi automaton.

Of the same type as `Tableau`, the algorithm in [52, 50], referred to as `Tester` here, uses the intermediate structure Fair Discrete System (FDS), a variant of Fair Transition System (FTS). Both of these two algorithms first compute all states (atoms), which contain subformulae of the given PTL formula and then build transitions by



evaluating the information on each states (atoms). The rest is to impose accepting conditions on states. These two algorithms generate large automata because they always create the full state space first, but they are easy to understand.

The algorithm in [49], referred to as `IncTableau` here, is another tableau construction which incrementally creates only reachable states from valid initial states. The construction starts with an automaton containing a set of initial states and a forged final state, and it then refines the automaton by creating and removing states and transitions until there are no inconsistent transitions. This construction produces smaller automata than `Tableau` and `Tester`.

All the above three translation algorithms support full PTL formulae. `Tableau` and `Tester` are considered not efficient because they always create full state space first. Although `IncTableau` constructs the automaton incrementally and supports past operators as well, a global view is maintained such that it cannot be used in truely on-the-fly model checking. Moreover, `IncTableau` has to fix both forward inconsistencies and backward inconsistencies. A forward inconsistency from a state s to a state t is resolved by computing the satisfactory successors of s as the cover of $\{f : \bigcirc f \in s\} \cup \{\neg f : \neg\bigcirc f \in s\} \cup t$ or $\{f : \bigcirc f \in s\} \cup t$ if the input formula is in negation normal form. However, our algorithm computes satisfactory successors from scratch as the cover of $\{f : \bigcirc f \in s\}$ which contains less formulae than `IncTableau`. Since only basic formulae are stored in a state under the setting of `LTL2AUT`, it is earier in our `Extended LTL2AUT+` to find a satisfactory successor from existing states and thus result in a smaller automaton.

Subsequently, several incremental algorithms were developed to enable truely on-the-fly model checking. Only reachable states are computed from initial states

during model checking. The automata representing the system, specification, and product can be created as needed during emptiness checking. We will recall three such incremental algorithms using the same convention as [17] to designate the three algorithms.

`GPVW` is the "simple on-the-fly" algorithm which uses a cover computation based on the tableau rules in Table 3.1 to expand the successors of a state [34]. An improved version (`GPVW+`) is suggested in the same paper. Later on, `LTL2AUT` [17] improves `GPVW+` by syntactical implication. The difference among these algorithms lies on how information is stored in a state and how contradiction and redundancy are detected during a cover computation. For example, `GPVW` always stores a processed formula while `LTL2AUT` stores only basic formulae. A formula is redundant in `GPVW` if it has been processed, while in `LTL2AUT`, it is redundant if it can be syntactically implied by the stored basic formulae.

`EQLTL` [23] and `Wring` [86] proceed in three stages, namely formulae rewriting, translation, and optimization. The two constructions both use incremental algorithms as their core translation algorithm, but apply different techniques in the rewriting and optimization stages and actually produce smaller automata. All the above incremental translation algorithms can produce smaller automata than `Tableau`, `Tester`, and `IncTableau` and are faster, but they do not support past operators.

Another kind of translation aims at a two-step translation which translates a PTL formula to an intermediate representation and then converts it into a Büchi automaton for model checking. Of this kind, `LTL2BA` [32] utilizes very-weak alternating co-Büchi automata (VWAA) and TGBA as its intermediate representations.

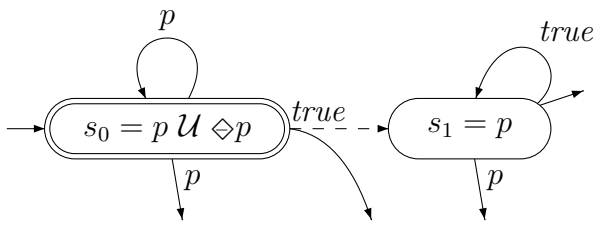
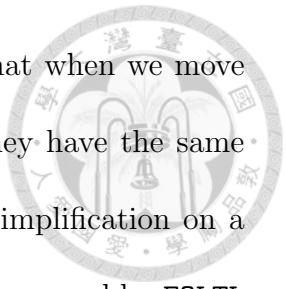


Figure 2.1: A two-way very weak alternating automaton equivalent to $p \,\mathcal{U}\, \diamondsuit p$. The solid transitions go forward while the dashed transition goes backward.

Although a two-step translation is used, `LTL2BA` can produce even smaller automata than incremental algorithms and is also faster. However, `LTL2BA` does not support past operators until its authors propose its generalization `PLTL2BA`.

`PLTL2BA` uses 2VWAA and TGBA as the intermediate representations. It has the following drawbacks in the conversion from a 2VWAA into a TGBA: (1) the conversion is fully automaton-based, and thus it misses the change of using the formulae on the states of the 2VWW to produce smaller TGBA, and (2) the saturation loop in the conversion iterates over the whole set of states currently constructed in the TGBA and thus many consistent states are checked for backward inconsistency during each iteration even if a timestamp mechanism is used to avoid some redundant computations. Consider the 2VWAA in Figure 2.1 as an example demonstrating the first drawback. During the conversion from the 2VWAA into a TGBA, it may create a state $\{p \,\mathcal{U}\, \diamondsuit p, p\}$ in the TGBA. From the view of PTL formulae, $p \,\mathcal{U}\, \diamondsuit p$ is implied by p and thus $p \,\mathcal{U}\, \diamondsuit p$ can be discarded if p is present. However, from the view of automata, the conversion is not aware of this simplification.

Both Couvreur's translation algorithm [16], referred to as `Couvreur` here, and `LTL2BUCHI` [35] use transition-based generalized Büchi automata (TGBA) as the only intermediate representation. While `Couvreur` uses BDD to represent an expanded formula and applies prime implicants to simplify the formula, `LTL2BUCHI` adopts



the core translation algorithm of LTL2AUT. The observation is that when we move labels from states to transitions, two states can be merged if they have the same successors before moving labels. Besides the direct simulation simplification on a TGBA proposed in LTL2BUCHI, several simplification techniques proposed by EQLTL and Wring are also implemented in LTL2BUCHI.

The algorithm proposed by Kesten and Pnueli in [51], denoted by KP02 here, can translate a QPTL formula into a congruent Büchi automaton by an inductive construction. In the congruent automaton, a new proposition is introduced to represent the time of the evaluaiton of the formula. A Büchi automaton equivalent to the formula is then obtained by taking the intersection of the congruemtn automaton and another Büchi automaton where the new proposition is true initially. This translation algorithm does not deal with $\Box$, $\boxminus$, $\mathcal{U}$, and $\mathcal{S}$ formulae but instead converts these formulae into longer formulae with quantifications, $\bigcirc$, $\Diamond$, $\ominus$, $\diamondminus$, and negations. For the case of a negated formula, complementation of Büchi automata is required. Thus, the algorithm produces large automata in general.

2.2 Complementation of Büchi Automata

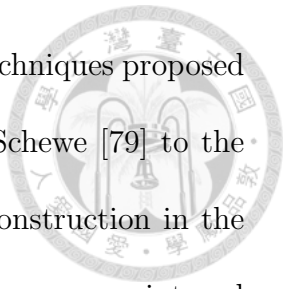
Ramsey-based approach. The very first complementation construction introduced by Büchi in 1960 involves a Ramsey-based combinatorial argument and results in a $2^{2^{O(n)}}$ blow-up in the state size [8]. This construction was later improved by Sistla, Vardi, and Wolper to reach a single-exponential complexity $2^{O(n^2)}$ [83]. In the improved construction, referred to as Ramsey in this paper, the complement is obtained by composing certain automata among a set of Büchi automata which form a partition of Σ^ω, based on Ramsey's Theorem. Various optimization heuristics for the Ramsey-based approach are described in [3, 26], but the focus in these

works is on universality and containment. In spite of the quadratic exponent of the Ramsey-based approach, it is shown in [3, 25, 26] to be quite competitive for universality and containment.

Determinization-based approach. Safra's $2^{O(n \log n)}$ construction is the first complementation construction that matches the $\Omega(n!)$ lower bound [77]. Later on, Muller and Schupp introduced a similar determinization construction which records more information and yields larger complements in most cases, but can be understood more easily [70, 4]. In [74], Piterman improved Safra's construction by using a more compact structure and using parity automata as the intermediate deterministic automata, which yields an upper bound of n^{2n}. (See also [80].) Piterman's construction, referred to as `SP` in this paper, performs complementation in stages: from NBW to DPW, from DPW to complement DPW, and finally from complement DPW to complement NBW. The idea is the use of (1) a compact Safra tree to capture the history of all runs on a word and (2) marks to indicate whether a run passes an accepting state again or dies.

Since the determinization-based approach performs complementation in stages, different optimization techniques can be applied separately to the different stages. For instance, several optimization heuristics on Safra's determinization and on simplifying the intermediate DRW were proposed by Klein and Baier [55].

Rank-based approach. The rank-based approach, proposed by Kupferman and Vardi, uses rank functions to measure the progress made by a node of a run tree towards fair termination [57]. The basic idea of this approach may be traced back to Klarlund's construction with a more complex measure [54]. Both constructions



have complexity $2^{O(n \log n)}$. There were also several optimization techniques proposed in [39, 27, 48]. A final improvement was proposed recently by Schewe [79] to the construction in [27]. The later construction performs a subset construction in the first phase. In the second phase, it continually guesses ranks from some point and verifies the guesses. Schewe proposed doing this verification in a piecemeal fashion. This yields a complement with $O((0.76n)^n)$ states, which matches the known lower bound modulo an $O(n^2)$ factor. We refer to the construction with Schewe's improvement as `Rank` in this paper.

Unlike the determinization-based approach that collects information from the history, the rank-based approach guesses ranks bounded by $2(n - |\mathcal{F}|)$ and results in many nondeterministic choices. This nondeterminism means that the rank-based construction often creates more useless states because many guesses may be verified later to be incorrect.

Slice-based approach. The slice-based construction was proposed by Kähler and Wilke in 2008 [47]. The blow-up of the construction is $4(3n)^n$ while its preliminary version in [101], referred to as `Slice` here, has a $(3n)^n$ blow-up[1]. Unlike the previous two approaches that analyze run trees, the slice-based approach analyzes reduced split trees. The construction `Slice` uses slices as states of the complement and performs a construction based on the evolution of reduced split trees in the first phase. By decorating vertices in slices at some point, it guesses whether a vertex belongs to an infinite branch of a reduced split tree or the vertex has a finite number of descendants. In the second phase, it verifies the guesses and enforces that accepting states will not occur infinitely often.

[1]The construction in [47] has a higher complexity than its preliminary version because it treats complementation and disambiguation in a uniform way.

The first phase of `Slice` in general creates more states than the first phase of `Rank` because of an ordering of vertices in the reduced split trees. Similar to `Rank`, `Slice` also introduces nondeterministic choices in guessing the decorations. While `Rank` guesses ranks bounded by $2(n - |\mathcal{F}|)$ and continually guesses ranks in the second phase, `Slice` guesses only once the decorations from a fixed set of size 3 at some point.

2.3 Containment Testing of Büchi Automata

Most recent work on incremental containment testing for Büchi automata was motivated by an implicit algorithm with a subsumption relation proposed by Doyen and Raskin [20, 21], which is an extension of the antichain algorithm for classic finite automata [18]. During a test of whether a Büchi automaton A is contained in another Büchi automaton B, this implicit algorithm computes incrementally the product states of $A \times \overline{B}$ based on the rank-based approach until a counterexample is found, while keeping only a minimal set of product states according to a subsumption relation on the product states. Such a subsumption relation allows the discard of product states whose behaviors are subsumed by other states and thus, avoids the exploration of the full product automaton.

Later in [26], Fogarty and Vardi proposed an incremental algorithm based on the Ramsey-based approach for universality testing, which is a special case of containment testing. Rather than computing product states incrementally in the rank-based containment testing algorithm, this Ramsey-based algorithm divides a universality testing of a Büchi automaton into smaller tests on proper pairs of elements in the transition monoid of the automaton. Each element in the transition monoid characterizes a set of finite strings that behave exactly the same in the automaton while

each proper pair characterizes a set of infinite strings that behave exactly the same in the automaton. The algorithm stops as long as it finds a proper pair that characterizes a set of infinite strings rejected by the automaton. They also extended the subsumption relation in [5] to reduce the elements need to take into account. Their experimental results showed that neither the rank-based algorithm nor the Ramsey-based algorithm to Büchi universality testing is superior.

Another important work in containment testing is the application of simulation relations in obtaining larger subsumption relations, which make containment testing even faster. Such a simulation subsumption was first proposed by Abdulla *et al.* for the universality and containment testing of classic finite automata and tree automata [3]. Later, Abdulla *et al.* proposed a subsumption with a forward simulation relation for the Ramsey-based containment testing algorithm [1]. Their experimental results showed that their approach consistently outperforms the approach in [26]. The simulation subsumption for the Ramsey-based containment testing algorithm was then extended by Abdulla *et al.* by using also backward simulation relations and simulation relations between the two automata under test [2].

Chapter 3

Preliminaries

In this section, we give some common notations and then describe basic definitions of propositional temporal logic (PTL), ω-automata, and the Manna-Pnueli temporal hierarchy.

3.1 Common Notations

Let S be a finite set and $\rho = s_0 s_1 \cdots$ a sequence where $s_i \in S$ for all $i \geq 0$. The empty sequence is denoted by ϵ. The length of ρ is denoted by $|\rho|$. If the sequence ρ is infinite, then $|\rho| = \omega$. The i-th element of the sequence ρ is denoted by $\rho(i) = s_i$ while the i-th suffix of the sequence is denoted by $\rho_i = s_i s_{i+1} \cdots$. We use the superscript $*$ and ω over a set to denote respectively finite and infinite sequences of elements of the set. The union of finite sequences and infinite sequences is denoted by the superscript $\circ$.

3.2 Propositional Temporal Logic

A propositional temporal logic (PTL) formula is inductively constructed from a set AP of atomic propositions where we apply (a) boolean operators, (b) the future operators $\bigcirc$ (next) and $\mathcal{U}$ (until), and (c) the past operators $\ominus$ (previous) and $\mathcal{S}$ (since) as follows:

- Every atomic proposition $p \in AP$ is a PTL formula.
- If φ, φ_1, and φ_2 are PTL formulae, then so are $\neg\varphi$, $\varphi_1 \vee \varphi_2$, $\bigcirc\varphi$, $\varphi_1 \,\mathcal{U}\, \varphi_2$, $\ominus\varphi$, and $\varphi_1 \,\mathcal{S}\, \varphi_2$.

Let φ, φ_1, and φ_2 be PTL formulae and p an atomic proposition in AP. Other shorthands and operators can be defined as below.

$$
\begin{array}{rclrcl}
\top & \equiv & p \vee \neg p & \bot & \equiv & \neg\top \\
\varphi_1 \wedge \varphi_2 & \equiv & \neg((\neg\varphi_1) \vee (\neg\varphi_2)) & & & \\
\varphi_1 \rightarrow \varphi_2 & \equiv & (\neg\varphi_1) \vee \varphi_2 & \varphi_1 \leftrightarrow \varphi_2 & \equiv & (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) \\
\Diamond\varphi & \equiv & \top \,\mathcal{U}\, \varphi & \Box\varphi & \equiv & \neg\Diamond\neg\varphi \\
\varphi_1 \,\mathcal{W}\, \varphi_2 & \equiv & (\Box\varphi_1) \vee (\varphi_1 \,\mathcal{U}\, \varphi_2) & \varphi_1 \,\mathcal{R}\, \varphi_2 & \equiv & \neg((\neg\varphi_1) \,\mathcal{U}\, (\neg\varphi_2)) \\
\circleddash\varphi & \equiv & \neg\ominus\neg\varphi & \mathit{first} & \equiv & \circleddash\bot \\
\diamondminus\varphi & \equiv & \top \,\mathcal{S}\, \varphi & \boxminus\varphi & \equiv & \neg\diamondminus\neg\varphi \\
\varphi_1 \,\mathcal{B}\, \varphi_2 & \equiv & (\boxminus\varphi_1) \vee (\varphi_1 \,\mathcal{S}\, \varphi_2) & \varphi_1 \,\mathcal{T}\, \varphi_2 & \equiv & \neg((\neg\varphi_1) \,\mathcal{S}\, (\neg\varphi_2))
\end{array}
$$

A *literal* is an atomic proposition or its negation. A *future formula* is a formula without past operators while a *past formula* is a formula without future operators. A *basic formula* is either `True`, `False`, a literal, a $\bigcirc$-formula, a $\ominus$-formula, or a $\circleddash$-formula. A set of formulae is basic if all its formulae are. A non-basic formula φ can be decomposed by the tableau rules in Table 3.1 such that

$$\varphi \leftrightarrow \bigwedge_{\psi_1 \in \alpha_1(\varphi)} \psi_1 \vee \bigwedge_{\psi_2 \in \alpha_2(\varphi)} \psi_2.$$

A formula is in *negation normal form* (NNF) if negation only appears right before atomic propositions. For every PTL formula f, there is a PTL formula g in NNF that is congruent (to be described later) to f.

Let $w = a_0 a_1 \cdots$ be an infinite word where each symbol $a_i \subseteq AP$ for all i. A symbol a can be viewed as an evaluation of atomic propositions where an atomic proposition p is *true* in s if and only if $p \in s$. Given an infinite word w, an index $i \geq 0$, and a PTL formula φ, the word satisfies the formula at position i if and only if $(w, i) \models \varphi$. An infinite word w satisfies a PTL formula φ (equivalently w

φ	$\alpha_1(\varphi)$	$\alpha_2(\varphi)$
$f \wedge g$	f, g	*false*
$f \vee g$	f	g
$f \,\mathcal{U}\, g$	g	f, $\bigcirc(f \,\mathcal{U}\, g)$
$f \,\mathcal{W}\, g$	g	f, $\bigcirc(f \,\mathcal{W}\, g)$
$f \,\mathcal{S}\, g$	g	f, $\ominus(f \,\mathcal{S}\, g)$
$f \,\mathcal{B}\, g$	g	f, $\circledcirc(f \,\mathcal{B}\, g)$

Table 3.1: Tableau rules.

is a model of φ, or φ holds on w), denoted by $w \models \varphi$, if and only if $(w, 0) \models \varphi$. The *language* of a PTL formula φ is the set of all models of φ, denoted by $L(\varphi)$. Let p be an atomic proposition and φ, φ_1, and φ_2 PTL formulae. The semantics of $(w, i) \models \varphi$ is given below.

- $(w, i) \models p \Longleftrightarrow p \in w(i)$,
- $(w, i) \models \neg\varphi \Longleftrightarrow (w, i) \not\models \varphi$,
- $(w, i) \models \varphi_1 \vee \varphi_2 \Longleftrightarrow (w, i) \models \varphi_1$ or $(w, i) \models \varphi_2$,
- $(w, i) \models \bigcirc\varphi \Longleftrightarrow (w, i + 1) \models \varphi$,
- $(w, i) \models \varphi_1 \,\mathcal{U}\, \varphi_2 \Longleftrightarrow$ for some $k \geq i, (w, k) \models \varphi_2$, and for all $j, i \leq j < k, (w, j) \models \varphi_1$
- $(w, i) \models \ominus\varphi \Longleftrightarrow i > 0$ and $(w, i - 1) \models \varphi$,
- $(w, i) \models \varphi_1 \,\mathcal{S}\, \varphi_2 \Longleftrightarrow$ for some $k \leq i, (w, k) \models \varphi_2$, and for all $j, k < j \leq i, (w, j) \models \varphi_1$

A formula φ is *satisfiable* if it holds on some model while the formula is called *valid*, denoted by $\models \varphi$, if it holds on all models. Formulae φ_1 and φ_2 are *equivalent*, denoted by $\varphi_1 \sim \varphi_2$ if the formula $\varphi_1 \leftrightarrow \varphi_2$ is valid. Formulae φ_1 and φ_2 are

congruent, denoted by $\varphi_1 \approx \varphi_2$ if the formula $\Box(\varphi_1 \leftrightarrow \varphi_2)$ is valid. Obviously, every formula φ has a formula in NNF that is congruent to φ.

3.3 ω-Automata

A *nondeterministic* ω-automaton A is a tuple $(\Sigma, Q, q_0, \delta, \mathcal{F})$ where Σ is a finite alphabet, Q a finite set of states, $q_0 \in Q$ the initial states, $\delta : Q \times \Sigma \to 2^Q$ a transition function, and $\mathcal{F}$ an acceptance condition, to be described subsequently. A is *deterministic* if $|\delta(q, a)| = 1$ for all $q \in Q$ and $a \in \Sigma$. Note that sometimes we may restrict the automaton to have only one initial state for simplicity.

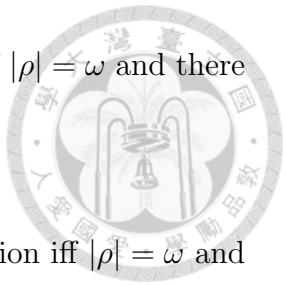
Given an automaton $A = (\Sigma, Q, q_0, \delta, \mathcal{F})$ and an infinite word $w = a_0 a_1 \cdots \in \Sigma^\omega$, a *run* ρ of A on w is a sequence $q_0 q_1 \cdots \in Q^\omega$ satisfying $q_{i+1} \in \delta(q_i, a_i)$ for all $i \geq 0$. A run is *accepting* if it satisfies the acceptance condition and is *rejecting* otherwise. A word is *accepted* by an automaton if there is an accepting run of the automaton on the word and is *rejected* otherwise. The *language* of an automaton A, denoted by $L(A)$, is the set of words accepted by A.

Let ρ be a run and $inf(\rho)$ be the set of states that occur infinitely often in ρ. Various automata can be defined by assigning different acceptance conditions below.

- *Büchi condition* where $\mathcal{F} \subseteq Q$: ρ satisfies the condition iff $|\rho| = \omega$ and $inf(\rho) \cap \mathcal{F} \neq \emptyset$.

- *Generalized Büchi condition* where $\mathcal{F} \subseteq 2^Q$: ρ satisfies the condition iff $|\rho| = \omega$ and for all $F \in \mathcal{F}$, $inf(\rho) \cap F \neq \emptyset$.

- *Co-Büchi condition* where $\mathcal{F} \subseteq Q$: ρ satisfies the condition iff $|\rho| = \omega$ and $inf(\rho) \cap \mathcal{F} = \emptyset$.



- *Muller condition* where $\mathcal{F} \subseteq 2^Q$: ρ satisfies the condition iff $|\rho| = \omega$ and there exists $F \in \mathcal{F}$ such that $inf(\rho) = F$.

- *Rabin condition*: where $\mathcal{F} \subseteq 2^Q \times 2^Q$: ρ satisfies the condition iff $|\rho| = \omega$ and there exists $(E, F) \in \mathcal{F}$ such that $inf(\rho) \cap E = \emptyset$ and $inf(\rho) \cap F \neq \emptyset$.

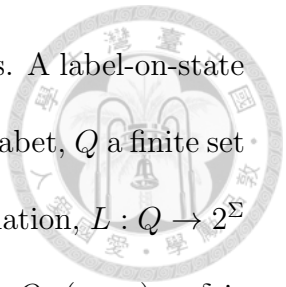
- *Streett condition* where $\mathcal{F} \subseteq 2^Q \times 2^Q$: ρ satisfies the condition iff $|\rho| = \omega$ and for all $(E, F) \in \mathcal{F}$, $inf(\rho) \cap F \neq \emptyset$ implies $inf(\rho) \cap E \neq \emptyset$.

- *Parity condition* where $\mathcal{F} : Q \to \{0, 1, \ldots, n\}$: ρ satisfies the condition iff $|\rho| = \omega$ and $min\{\mathcal{F}(q) \mid q \in inf(\rho)\}$ is even. $\mathcal{F}(q)$ is called the *color* of a state q and $\mathcal{F}$ here is also called a coloring function.

Among automata with these acceptance conditions, only deterministic Büchi automata are less expressive than the others.

We use a system of three-letter acronyms to denote these finite-state automata. The first letter indicates whether the automaton is **n**ondeterministic or **d**eterministic. The second letter indicates whether the acceptance condition is classic **F**inite, **B**üchi, **C**o-Büchi, **G**eneralized Büchi, **M**uller, **R**abin, **S**treett, or **P**arity. The third letter is always a "**W**" indicating the automaton accepts words. For example, NBW stands for a nondeterministic Büchi automaton and DPW stands for a deterministic parity automaton.

An ω-automaton A is *universal* iff $L(A) = \Sigma^\omega$. Two automata A and A' are *equivalent* iff $L(A) = L(A')$. A state is *live* if it occurs in an accepting run on some word, and is *dead* otherwise. Dead states can be discovered using a nonemptiness algorithm, cf. [99], and can be pruned off without affecting the language of the automaton.



Automata may have labels on states rather than on transitions. A label-on-state automaton is a six tuple $(\Sigma, Q, I, \delta, L, \mathcal{F})$ where Σ is the finite alphabet, Q a finite set of states, $I \subseteq Q$ a set of initial states, $\delta \subseteq Q \times Q$ the transition relation, $L : Q \to 2^{\Sigma}$ a labeling function, and $\mathcal{F}$ an acceptance condition. For $s_i, s_j \in Q$, $(s_i, s_j) \in \delta$ is also denoted by $\delta(s_i, s_j)$. Let $w = a_0 a_1 \cdots$ be an infinite word where $a_i \in \Sigma$ for all i. A run ρ of a label-on-state automaton on w is a sequence of states $s_0 s_1 \cdots$ such that $s_0 \in I$ and, for every $i \geq 0$, both $\delta(s_i, s_{i+1})$ and $a_i \in L(s_i)$. The acceptance of a run of a label-on-state automaton is the same as that of an automaton. A label-on-state automaton can be easily converted into an equivalent automaton and vice versa.

3.4 Temporal Hierarchy

In the Manna-Pnueli temporal hierarchy [65], properties specifiable by PTL formulae are classified into six classes shown in Figure 3.1. For each class, there is a canonical form of PTL formulae and every automaton in the class has an equivalent PTL formula in that canonical form. Let p, q, p_i, and q_i be past temporal formulae. The six classes and their canonical forms are described as the following.

- Safety: The canonical form of this class is $\Box p$. A property in this class basically states that something bad will not happen.

- Guarantee: The canonical form of this class is $\Diamond p$. A property in this class basically states that something good will eventually happen.

- Obligation: The canonical form of this class is $\wedge_i(\Box p_i \vee \Diamond q_i)$. A property in this class is a combination of properties in the safety and guarantee classes.

- Recurrence: The canonical form of this class is $\Box\Diamond p$. A property in this class

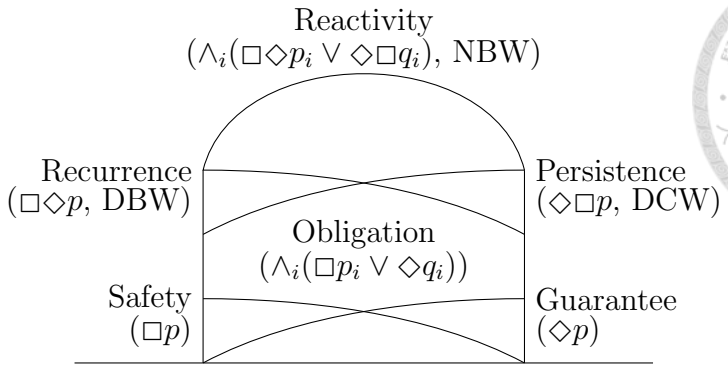


Figure 3.1: The Manna-Pnueli Temporal Hierarchy. Here p, q, p_i, and q_i are past temporal formulae. DCW stands for deterministic co-Büchi (word) automaton.

basically states that some event will occur infinitely often. For each formula in this class, there is a DBW equivalent to the formula.

- Persistence: The canonical form of this class is $\Diamond\Box p$. A property in this class basically states that some event will constantly occur after some time. For each formula in this class, there is a DCW equivalent to the formula.

- Reactivity: The canonical form of this class is $\wedge_i(\Box\Diamond p_i \vee \Diamond\Box q_i)$. Every PTL formula is equivalent to some PTL formula in the reactivity canonical form.

Chapter 4

Incremental Translation of PTL Formulae

In this chapter, we first describe the translation algorithm `LTL2AUT` [17] which can translate an LTL formula to an equivalent label-on-state NGW. The label-on-state NGW can later be converted into an equivalent NBW. We then introduce a backtrace procedure which makes `LTL2AUT` support PTL formulae including past operators. A correctness proof is given. We also propose some optimization heuristics for the extended algorithm and extend other state-based and transition-based translation algorithms in a similar way. At the end of this chapter, we compare our extended algorithms with others that also support past operators. An experimentative comparison between our extended algorithms and other LTL translation algorithms that rely on a separation of past and future operators to handle PTL formulae is also described.

4.1 The Classic Construction

`LTL2AUT` uses a cover computation procedure to calculate the successive states of a state. A *cover* of a set $\mathbb{F}$ of formulae is a possibly empty set of sets of PTL formulae

$C = \{C_0, C_1, \ldots, C_n\}$ such that

$$\bigwedge_{f \in \mathbb{F}} f \leftrightarrow \bigvee_{0 \le i \le n} \bigwedge_{g_i \in C_i} g_i. \tag{4.1}$$

Each element in the cover represents a symbolic state which serves as an alternative way to satisfy all formulae in $\mathbb{F}$. The cover can be computed by the function **Cover** in Algorithms 1 and 2 where

- $SELECT(ToCover)$ returns a formula from $ToCover$,
- $STORE(f)$ always returns false,
- $CONTRADICTION(f, ToCover, Current, Covered)$ returns true iff $SATISFY(ToCover \cup Current, \neg f)$,
- $REDUNDANT(f, ToCover, Current, Covered)$ returns true iff $SATISFY(ToCover \cup Current, f)$ and, if f is $\varphi \,\mathcal{U}\, \psi$, $SATISFY(ToCover \cup Current, \psi)$, and
- $SATISFY(\mathbb{F}, f)$ returns true iff the NNF of f belongs to $SI(\mathbb{F})$.

Given a set $\mathbb{F}$ of formulae, $SI(\mathbb{F})$ is a set of syntactically implied formulae by $\mathbb{F}$ and is defined inductively as follows:

- $true \in SI(\mathbb{F})$,
- $f \in SI(\mathbb{F})$ if $f \in \mathbb{F}$,
- $f \in SI(\mathbb{F})$ if f is non-basic and either $\alpha_1(f) \subseteq SI(\mathbb{F})$ or $\alpha_2(f) \subseteq SI(\mathbb{F})$.

Algorithm 1 Cover$(\mathbb{F})$

1: **return cover**$(\mathbb{F}, \emptyset, \emptyset, \emptyset)$

We define four additional notations: $\overrightarrow{s} = \{f : \bigcirc f \in s\}$, $\overleftarrow{s} = \{f : \ominus f \in s \text{ or } \circledcirc f \in s\}$, $s^+ = s \cap AP$, and $s^- = \{p \in AP : \neg p \in s\}$. For two sets s and t of PTL formulae,

Algorithm 2 cover($ToCover$, $Current$, $Covered$, $Cover$)

```
1: if ToCover = ∅ then
2:   return Cover ∪ {Current}
3: else
4:   f = SELECT(ToCover)
5:   remove f from ToCover and add it to Covered
6:   if STORE(f) then
7:     Current = Current ∪ {f}
8:   if CONTRADICTION(f, ToCover, Current, Covered) then
9:     return Cover
10:  else
11:    if REDUNDANT(f, ToCover, Current, Covered) then
12:      return cover(ToCover, Current, Covered, Cover)
13:    else
14:      if f is basic then
15:        return cover(ToCover, Current ∪ {f}, Covered, Cover)
16:      else
17:        return cover(ToCover ∪ (α1(f) \ Current), Current, Covered,
18:             cover(ToCover ∪ (α2(f) \ Current), Current, Covered, Cover)
```

we say s and t are *forward inconsistent* if $SATISFY(t, \vec{s})$ is false, and are *backward inconsistent* if $SATISFY(s, \overleftarrow{t})$ is false. For a set $\mathbb{G}$ of formulae, $SATISFY(\mathbb{F}, \mathbb{G})$ returns true iff $SATISFY(\mathbb{F}, g)$ returns true for all $g \in \mathbb{G}$.

Given a PTL formula φ in NNF over a set AP of atomic propositions, `LTL2AUT` constructs a label-on-state NGW $A_\varphi = (\Sigma, Q, I, \delta, L, F)$ as follows such that A_φ accepts all models of φ:

- $\Sigma = 2^{AP}$ where AP is the set of atomic propositions occuring in the input formula φ.
- Q, I, and δ are computed by the algorithm **create_automaton_structure**.
- $L(s) = \{a \in \Sigma : s^+ \subseteq a \text{ and } a \cap s^- = \emptyset\}$.
- $F = \{F_{f\ \mathcal{U}\ g} : f\ \mathcal{U}\ g \text{ is a subformula of the input formula } \varphi\}$ where

$$F_{f\ \mathcal{U}\ g} = \{s \in Q : SATISFY(s, f\ \mathcal{U}\ g) \text{ implies } SATISFY(s, g)\}.$$

Algorithm 3 create_automaton_structure(φ)

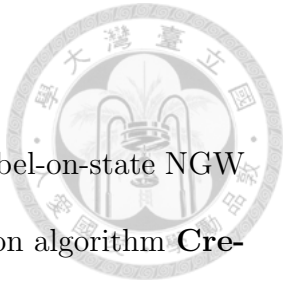
1: $Q = I = U =$ **Cover**$(\varphi), \delta = \emptyset$
2: **while** $U \neq \emptyset$ **do**
3: remove s from U
4: **for** $t \in$ **Cover**$(\vec{s})$ **do**
5: **if** $t \notin Q$ **then**
6: $Q = Q \cup \{t\}$
7: $U = U \cup \{t\}$
8: $\delta = \delta \cup \{(s, t)\}$

4.2 Construction wtih Past Operators

In this section, we present our algorithm that translates, in an incremental fashion, a given PTL formula to a label-on-state NGW. The part of our algorithm that deals with the future fragment of PTL (i.e., LTL) is essentially `LTL2AUT`. This is to be extended with a procedure called *backtrace* to handle past operators. The extended algorithm will be referred to as `Extended LTL2AUT`.

Consider the translation of an LTL formula, it is sufficient to recursively compute successors of existing states until no more states can be created. The translation becomes more complex if past operators are involved. Whenever a state s containing past formulae is created, these past formulae impose additional constraints on the predecessors of s. However, when we created those predecessors, the constraints were not taken into account and thus the predecessors and s may be backward inconsistent. Besides, the predecessors and s may be forward inconsistent if the imposed constraints contain future formulae which in turn impose additional requirements back on s. To resolve backward inconsistency, a backtrace is applied whenever we are trying to add a transition which is backward inconsistent. During the backtrace, states may be created to refine the predecessors. For forward inconsistency on a state, we just recompute its successors.

4.2.1 The Translation Algorithm



Given a PTL formula φ in NNF, our goal is to translate φ to a label-on-state NGW A_φ which accepts exactly all the models of φ. The core translation algorithm **CreateAutomaton** is based on **create_automaton_structure** except that backtrace is done when adding transitions (line 8) and certain states cannot be initial states anymore (lines 9-11).

Algorithm 4 CreateAutomaton(φ)

1: $Q = I = U =$ **Cover**$(\varphi), \delta = \kappa = \gamma = \emptyset$
2: **while** $U \neq \emptyset$ **do**
3: remove s from U
4: **for** $t \in$ **Cover**$(\vec{s})$ **do**
5: **if** $t \notin Q$ **then**
6: $Q = Q \cup \{t\}$
7: $U = U \cup \{t\}$
8: **AddTransition**(s, t)
9: **for** $s \in I$ **do**
10: **if** $\{f : \ominus f \in s\} \neq \emptyset$ **then**
11: $I = I \backslash \{s\}$

There are several global variables: the set U is used to keep unprocessed states, all created states will be added to the set Q, candidates of initial states will be added to the set I, δ is a transition function, κ records the performed backtraces, and γ is a mapping used to track refinement relations.

The algorithm starts by computing the cover of φ which serves as the initial states (line 1). Initially, the transition function δ, the backtrace records κ and the refinement function γ are all empty (line 1).

As long as the set U is not empty, a state s in U is removed (line 3). The forward expansion of s is done by computating its successors as the cover of $\vec{s}$ (line 4). For each successor t, it is added to the set Q and U for further process if it is a new state (lines 5-7). The transition from s to its successor t is added by the function

AddTransition (line 8) which ensures that backward inconsistency is found and resolved.

After creating all needed states and transitions and resolving all inconsistency, the algorithm fixes the set I by removing states containing $\ominus$-formulae (lines 9-11) because $\ominus$-formulae never hold initially.

The resulting generalized Büchi automaton $A_\varphi = (\Sigma, Q, I, \delta L, F)$ can be defined as follows:

- $\Sigma = 2^{AP}$ where AP is the set of atomic propositions occuring in the input formula φ.
- $L(s) = \{a \in \Sigma : s^+ \subseteq a \text{ and } a \cap s^- = \emptyset\}$.
- $F = \{F_{f\ \mathcal{U}\ g} : f\ \mathcal{U}\ g \text{ is a subformula of the input formula } \varphi\}$ where

$$F_{f\ \mathcal{U}\ g} = \{s \in Q : \mathit{SATISFY}(s, f\ \mathcal{U}\ g) \text{ implies } \mathit{SATISFY}(s, g)\}.$$

4.2.2 The Backtrace Procedure

A backtrace from t to s is needed if we are trying to add a transition from s to t but s and t are not backward consistent. The set κ is used to remember the resolved inconsistency so that the backtrace from a state t to a state s is never performed more than once (lines 1-2).

To resolve potential inconsistency, we cannot simply impose the constraint $\overleftarrow{t}$ on s because $\overleftarrow{t}$ may contain formulae which are not required by other successors of s. For example, consider the intermediate translation result of $\Box(p \rightarrow \ominus(q\ \mathcal{U}\ r))$ by `LTL2AUT` in Figure 4.1. We are trying to add a transition from s_0 to one of its successors, s_1. Although state s_1 requires that its predecessor must satisfy $q\ \mathcal{U}\ r$,

Algorithm 5 BackTrace(s, t)

```
1: if (s,t) ∉ κ then
2:    κ = κ ∪ {(s,t)}
3:    for r ∈ Cover(←t ∪ s) do
4:       γ(s) = γ(s) ∪ {r}
5:       if r ∉ Q then
6:          Q = Q ∪ {r}
7:          U = U ∪ {r}
8:       if s ∈ I then
9:          I = I ∪ {r}
10:      for u ∈ {u : δ(u,s)} do
11:         AddTransition(u,r)
```

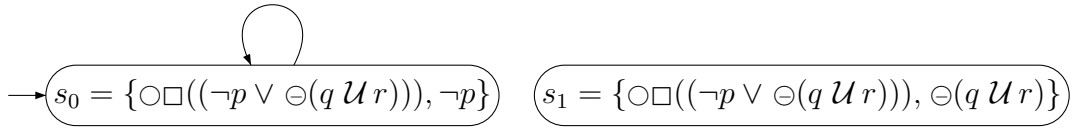


Figure 4.1: Part of an intermediate result of translating $\Box(p \rightarrow \ominus(q\,\mathcal{U}\,r))$.

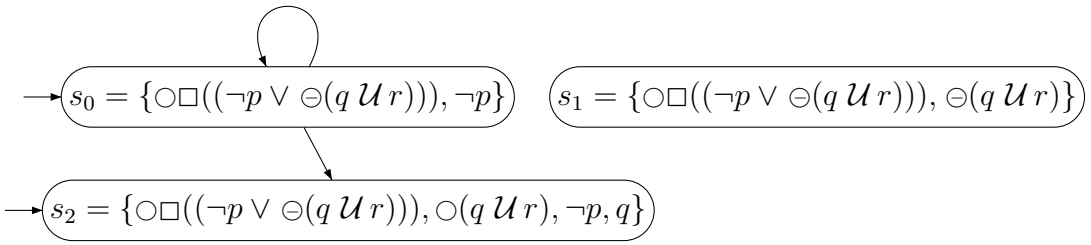
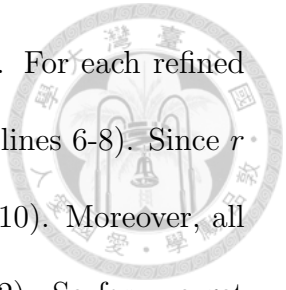


Figure 4.2: Part of an intermediate result of translating $\Box(p \rightarrow \ominus(q\,\mathcal{U}\,r))$ after the first backtrace

we can neither add (1) r nor (2) q and $\bigcirc(q\,\mathcal{U}\,r)$ to s_0 because the infinite word $(\neg p \neg q \neg r)^\omega$, which is a model of $\Box(p \rightarrow \ominus(q\,\mathcal{U}\,r))$, would be rejected.

Note that a symbolic state (i.e., an element in a cover) is used to represent a set of states which all satisfy the set of formulae in the symbolic state. A backward inconsistency here does not necessarily mean all states represented by s fail to satisfy the requirement. Thus the solution is to keep the original predecessor s but compute satisfactory predecessors.

The satisfactory predecessors are refined from the original predecessor s by computing the cover of $\overleftarrow{t} \cup s$ (line 4). The refinement relation is kept in the function



γ where $r \in \gamma(s)$ indicates that r is a refined state of s (line 5). For each refined predecessor r, we add it to the sets Q and U if it is a new state (lines 6-8). Since r is a refined state of s, r should be an initial state if s is (lines 9-10). Moreover, all predecessors of s should have transitions to r as well (lines 11-12). So far, we get the intermediate result of the example shown in Figure 4.2 by processing the refined predecessor s_2 of s_0 first.

After building the history of the refined state r, we need to expand its future. Although r is a refined state of s, we cannot simply add a transition from r to the original successor t of s because this may induce forward inconsistency. For example, if we add a transition from s_2 to s_1 in Figure 4.2, forward inconsistency occurs because s_1 does not satisfy $q \,\mathcal{U}\, r$. Thus, new refined states are added to U to be processed later.

4.2.3 The AddTransition Procedure

Given two states s and t, the function **AddTransition** not only adds transitions from s to t, but also adds other transitions and invokes backtrace if necessary. First, we check if there is a backward inconsistency between s and t (line 1). If there is no backward inconsistency, we add a transition from s to t and try to add a transition from s to every refined state r of t because r satisfies more formulae than t including all constraints from s (lines 2-4). If a backward inconsistency between s and t is found, the backtrace procedure is performed (line 6).

4.3 Correctness

The correctness of `Extended LTL2AUT` is given by the following theorem.

Theorem 4.3.1. *A word w is accepted by the automaton A_φ constructed for the*

Algorithm 6 AddTransition(s, t)

1: **if** SATISFY$(s, \overleftarrow{t})$ **then**
2: $\quad \delta = \delta \cup \{(s, t)\}$
3: $\quad$ **for** $r \in \gamma(t)$ **do**
4: $\quad\quad$ **AddTransition**(s, r)
5: **else**
6: $\quad$ **Backrace**(s, t)

PTL formula φ if and only if $w \models \varphi$.

Proof. This is proven by Lemmas 4.3.7 and 4.3.8 which require Lemmas 1-5. □

First of all, the notion of a run may be extended to start from any state in an automaton.

Definition. *A pseudo-run ρ of a generalized Büchi automaton $A = (\Sigma, Q, \delta, I, L, F)$ on a word w is an infinite sequence $s_0 s_1 \cdots$ of states in Q such that $\delta(s_i, s_{i+1})$ and $w(i) \in L(s_i)$ for all $i \geq 0$. ρ is accepting if for all $F_i \in F$, $inf(\rho) \cap F_i \neq \emptyset$.*

Since our forward expansion is based on the incremental algorithms and the cover computation in them, properties of the cover computation can be adapted for full PTL formulae.

Lemma 4.3.2. *Let $\mathbb{F}$ be a set of formulae and $Cover(\mathbb{F}) = \{C_0, C_1, \ldots, C_n\}$. Then, for all $0 \leq i \leq n$, (a) C_i is basic, (b) $\mathbb{F} \subseteq SI(C_i)$, and (c) $\bigwedge_{f \in \mathbb{F}} f \leftrightarrow \bigvee_{0 \leq i \leq n} \bigwedge_{g \in C_i} g$. Moreover, for a word w and a position j, if $(w, j) \models \mathbb{F} \cup \{g_k : k \in K\}$ and $\{f_k \,\mathcal{U}\, g_k : k \in K\} \subseteq \mathbb{F}$, there exists i such that (d) $(w, j) \models C_i$, and (e) $\{g_k : k \in K\} \subseteq SI(C_i)$.*

Proof. This lemma can be proven by the same technique applied in [17] for the cover computation of LTL formulae. □

In the following, let $A_\varphi = (\Sigma, Q, \delta, I, L, F)$ be the automaton constructed for a PTL formula φ.

Lemma 4.3.3. *For all* $(s,t) \in \delta$, *(a)* $\overrightarrow{s} \subseteq SI(t)$, *and (b)* $\overleftarrow{t} \subseteq SI(s)$.

Proof. The only place of adding (s,t) to δ is line 2 in **AddTransition**. The if-condition in line 1 ensures that $SATISFY(s, \overleftarrow{t})$ is true, which implies that (b) holds. For (a), it is ensured before making a call to **AddTransition**(s,t), which has three occurrences. In **CreateAutomaton**, $t \in Cover(\overrightarrow{s})$ and thus $\overrightarrow{s} \subseteq t$ by Lemma 4.3.2(b). Both in **BackTrace** and in **AddTransition**, s is a predecessor of another state u which is refined by $t \in Cover(Pred \cup u)$ where $Pred$ is some set of formulae. As (s,u) is already in δ, $\overrightarrow{s} \subseteq SI(u)$. Thus, by Lemma 4.3.2(b), $Pred \cup u \subseteq SI(t)$ and $\overrightarrow{s} \subseteq SI(t)$. □

Lemma 4.3.4. *Let* $s_0 s_1 \cdots$ *be a pseudo-run of* A_φ *such that* $f\ \mathcal{U}\ g \in SI(s_i)$ *for some* $i \geq 0$. *Then one of the following holds:*

1. $\forall j \geq i : f, f\ \mathcal{U}\ g \in SI(s_j)$ *and* $g \notin SI(s_j)$.

2. $\exists j \geq i : g \in SI(s_j)$ *and for all* $i \leq k < j$, $f, f\ \mathcal{U}\ g \in SI(s_k)$.

Proof. This lemma is proven by the definition of SI and by Lemma 4.3.3. □

Lemma 4.3.5. *Let* $s_0 s_1 \cdots$ *be a pseudo-run of* A_φ *such that* $f\ \mathcal{U}\ g \in SI(s_i)$ *for some* $i \geq 0$. *Then there is* $0 \leq j \leq i$ *such that* $g \in SI(s_j)$ *and for all* $j < k \leq i$, $f, f\ \mathcal{S}\ g \in SI(s_k)$.

Proof. This lemma is proven by the definition of SI and by Lemma 4.3.3. □

Lemma 4.3.6. *Let* $\rho = s_0 s_1 \cdots$ *be an accepting pseudo-run of* A_φ *on* w *and* s_0 *does not contain any* $\ominus$*-formula. Then,* $(w,i) \models SI(s_i)$ *for all* $i \geq 0$.

Proof. We prove by induction on the structure of the formulae in $SI(s_i)$.

- The base case is a literal $f \in s_i$. By the definition of the labeling function, $L(s_i) = \{a \in \Sigma : s^+ \subseteq a \text{ and } a \cap s^- = \emptyset\}$. As ρ is a run of A_φ on w, $w(i) \in L(s_i)$. Thus, $(w, i) \models f$.

- There are four cases to consider for the induction step:

  - $\bigcirc f \in s_i$: $f \in SI(s_i)$ by Lemma 4.3.3. By the induction hypothesis, $(w, i+1) \models f$ and thus, $(w, i) \models \bigcirc f$.

  - $f \,\mathcal{U}\, g \in SI(s_i)$: By the fact that ρ is accepting, only case 2 in Lemma 4.3.4 is possible. Thus, by the induction hypothesis, $(w, i) \models f \,\mathcal{U}\, g$.

  - $\ominus f \in SI(s_i)$: By the assumption and Lemma 4.3.3, $i > 0$ and $f \in SI(s_{i-1})$. Thus, by the induction hypothesis, $(w, i-1) \models f$ and $(w, i) \models \ominus f$.

  - $f \,\mathcal{S}\, g \in SI(s_i)$: By Lemma 4.3.5 and the induction hypothesis, $(w, i) \models f \,\mathcal{S}\, g$.

□

Lemma 4.3.7. *Let ρ be an accepting run of A_φ on w. Then, $w \models \varphi$.*

Proof. By Lemma 4.3.6, $w \models SI(s_0)$. By the construction, the initial states I can be defined as $I_2 - \{s \in I_2 : \ominus f \in s \text{ for some PTL formula } f\}$ where $I_2 = I_0 \cup I_1$, $I_0 = Cover(\{\varphi\})$, and I_1 is a set of states refined from the states in I_0. Thus, by Lemma 4.3.2(b), $\{\varphi\} \subseteq SI(s_0)$ and $w \models \varphi$. □

Lemma 4.3.8. *Let w be an infinite word and $w \models \varphi$. Then, w is accepted by A_φ.*

Proof. We prove by induction on the number n of forward and backward expansions and constructs a sequence $\rho_n = s_0 s_1 \cdots s_m$ of states ($m \leq n$) such that the following conditions holds.

[C1] $s_0 \in I$.

[C2] $(w, i) \models s_i$ for all $0 \leq i \leq m$.

[C3] $\delta(s_i, s_{i+1})$ for all $0 \leq i < m - 1$.

[C4] $s_{m-1}^{\rightarrow} \subseteq SI(s_m)$.

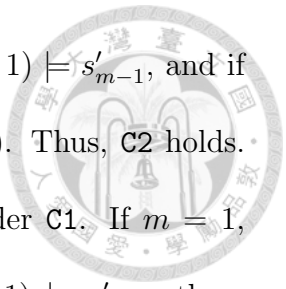
[C5] For all $0 \leq i < m$, if $f \,\mathcal{U}\, g \in s_i^{\rightarrow}$ and $(w, i + 1) \models g$, then $g \in SI(s_{i+1})$.

In the base case, $\rho_0 = s_0$ where s_0 is selected from $Cover(\{\varphi\})$ such that by Lemma 4.3.2, $(w, 0) \models s_0$ and thus C2 holds. By the fact that s_0 has a model, there is no $\ominus$-formula in s_0. Thus, by the definition of initial states, $s_0 \in I$ and C1 holds. C3, C4, and C5 hold in the base case trivially.

In the induction step, consider a sequence $\rho_n = s_0 s_1 \cdots s_{m-1} s_m$ of states. There are two cases depending on whether s_{m-1} and s_m are backward consistent.

- s_{m-1} and s_m are backward consistent. By the induction hypothesis, $(w, m) \models s_m$ and therefore $(w, m+1) \models s_m^{\rightarrow}$. Thus, by Lemma 4.3.2 and the construction, we can construct $\rho_{n+1} = s_0 s_1 \cdots s_m s_{m+1}$ where s_{m+1} is selected from $Cover(s_m^{\rightarrow})$ such that $s_m^{\rightarrow} \subseteq SI(s_{m+1})$, $(w, m + 1) \models s_{m+1}$, and if $(w, m + 1) \models g$ for an $f \,\mathcal{U}\, g \in s_m^{\rightarrow}$, $g \in SI(s_{m+1})$. Thus, C2, C4, and C5 hold. By the induction hypothesis and the fact that s_{m-1} and s_m are backward consistent, C1 and C3 hold.

- s_{m-1} and s_m are backward inconsistent. By the induction hypothesis, $(w, m - 1) \models s_{m-1}$ and $(w, m) \models s_m$. Thus, $(w, m - 1) \models s_m^{\leftarrow}$ and $(w, m - 1) \models s_{m-1} \cup s_m^{\leftarrow}$. By the construction and Lemma 4.3.2, a backtrace is performed and we can construct $\rho_{n+1} = s_0 s_1 \cdots s_{m-2} s'_{m-1}$ where s'_{m-1} is selected from



$Cover(s_{m-1} \cup \overleftarrow{s_m})$ such that $s_{m-1} \cup \overleftarrow{s_m} \subseteq SI(s'_{m-1})$, $(w, m-1) \models s'_{m-1}$, and if $f \,\mathcal{U}\, g \in s_{m-1} \cup \overleftarrow{s_m}$ and $(w, m-1) \models g$, then $g \in SI(s'_{m-1})$. Thus, `C2` holds. By the induction hypothesis, `C3`, `C4`, and `C5` hold. Consider `C1`. If $m = 1$, $s_{m-1} = s_0 \in I$ by the induction hypothesis. As $(w, m-1) \models s'_{m-1}$, there is no $\ominus$-formula in s'_{m-1}. Thus, $s'_{m-1} \in I$ by the construction and `C1` holds. Otherwise, `C1` holds by the induction hypothesis.

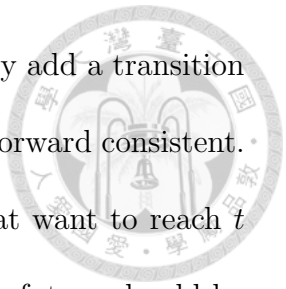
We have to show that the sequence $\rho_0\rho_1\cdots$ converges. Define a partial order $\preceq$ on states by $s_i \preceq s_j$ if $s_i \subseteq s_j$. By the construction, the sequence $\rho_0\rho_1\cdots$ is strictly increasing in the lexicographic order induced by $\preceq$. As the state space is finite, the sequence $\rho_0\rho_1\cdots$ converges to an infinite sequence ρ of states, which forms a run of A_φ because of `C1` and `C3`.

We next show that ρ is accepting. Consider a formula $f \,\mathcal{U}\, g \in SI(s_i)$. If $g \in SI(s_i)$, then $f \,\mathcal{U}\, g$ is fulfilled immediately in s_i. Otherwise, $g \notin SI(s_i)$ and $f, \bigcirc(f \,\mathcal{U}\, g) \in SI(s_i)$. By `C2`, $(w, i) \models \bigcirc(f \,\mathcal{U}\, g)$ and there exists $j > i$ such that $(w, j) \models g$. If $g \in SI(s_k)$ for some $i < k < j$, then $f \,\mathcal{U}\, g$ is fulfilled in s_k. Otherwise, $g \notin SI(s_k)$ and $f, \bigcirc(f \,\mathcal{U}\, g) \in SI(s_k)$ for all $i < k < j$ by `C4`. By `C5`, $g \in SI(s_j)$ and therefore $f \,\mathcal{U}\, g$ is fulfilled in s_j. Thus, ρ is accepting. □

4.4 Optimization Heuristics

Two optimization heuristics are described in this section. The first one is the postponed expansion of refined states while the second one is the application of prime implicants. We will refer `Extended LTL2AUT` combined with the two optimization heuristics to as `Extended LTL2AUT+`.

Originally, when a new refined state r is created during the backtrace from t to s,



t is added to U to expand its future later. Actually, we can simply add a transition from r to t without fully expanding the future of r if r and t are forward consistent. The reason is that so far r is only used to replace s for runs that want to reach t through s. However, if r is reached during forward expansion, its future should be fully expanded.

To achieve the postponed expansion of refined states, we create another global variable R to store refined states whose forward expansions are postponed. In the **CreateAutomaton** procedure, the following code is added after line 7 as the else branch of the if-condition in line 5:

if $t \in R$ **then**

$R = R \backslash \{t\}$

$U = U \cup \{t\}$

In the **BackTrace** procedure, line 7 is replaced by the following code.

if $SATISFY(t, \vec{r})$ **then**

$R = R \cup \{r\}$

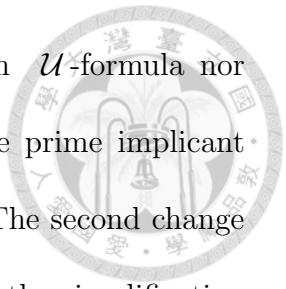
AddTransition(r, t)

else

$U = U \cup \{r\}$

Prime implicants can be used to simplify a Boolean formula in a sum of products. Such simplification has been applied in `Couvreur` [16] which encodes an LTL formula in a Boolean formula based on the tableau rules. For our extended incremental algorithms, we can use the same idea to compute the cover of a set of PTL formulae.

As the acceptance condition is on states in the incremental algorithms but is on transitions in `Couvreur`, two modifications are required to apply prime implicants.



The first change is that if a prime implicant neither fulfills an $\mathcal{U}$-formula nor implies the $\mathcal{U}$-formula, we have to add the $\mathcal{U}$-formula to the prime implicant such that the acceptance condition can be correctly computed. The second change is that syntactical implication has to be weakened because after the simplification with prime implicants, some basic formulae may be discarded such that syntactical implication does not hold as before. For example, the originally computed cover of $p\ \mathcal{B}\neg p$ is $\{\{p, \circledcirc(p\ \mathcal{B}\neg p)\}, \{\neg p\}\}$. With prime implicants, the cover becomes $\{\{\circledcirc(p\ \mathcal{B}\neg p)\}, \{\neg p\}\}$ but the first element in the new cover does not syntactically imply $p\ \mathcal{B}\neg p$. Actually, if we do a case analysis on p, both $\{\circledcirc(p\ \mathcal{B}\neg p), p\}$ and $\{\circledcirc(p\ \mathcal{B}\neg p), \neg p\}$ can syntactically imply $p\ \mathcal{B}\neg p$. Thus, the syntactical implication of a set $\mathbb{F}$ of PTL formulae is weakened by an additional rule:

- $f \in SI(\mathbb{F})$ if there is a formula g such that $f \in SI(\mathbb{F} \cup \{g\})$ and $f \in SI(\mathbb{F} \cup \{\neg g\})$.

4.5 Extension to Other Algorithms

Besides `LTL2AUT`, we have also extended two state-based algorithms, namely `GPVW` and `MoDeLLa`, and two transitions-based algorithms, namely `Couvreur` and `LTL2BUCHI`, to support past operators.

4.5.1 `Extended GPVW`

The extension to `GPVW`, resulting in `Extended GPVW`, is easy as `GPVW` and `LTL2AUT` were presented in a uniform way in [17]. `GPVW+` is also presented in the uniform way but cannot be similarly extended because the requirement for a state s to satisfy a formula f in `GPVW+` is as strict as in `GPVW` but some useful formulae are discarded in the cover computation. That is, in `GPVW+`, $SATISFY(\mathbb{F}, f)$ returns true iff $f \in \mathbb{F}$

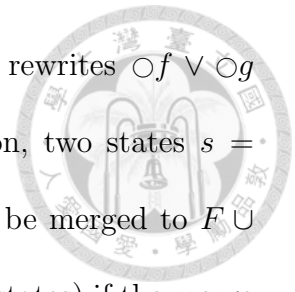
while $STORE(f)$ returns true iff f is an $\mathcal{U}$-formula or f is the right argument of an $\mathcal{U}$-formula. Consider the translation of $\bigcirc\boxminus p$ by `GPVW+`. The successor of the initial state $q_0 = \{\bigcirc\boxminus p\}$ computed by **Cover** is $q_1 = \{p, \circledcirc\boxminus p\}$. Since q_0 does not satisfy $\boxminus p$ imposed by q_1, we have to refine q_0 by $\boxminus p$ and obtain a refined state $q_2 = \{\bigcirc\boxminus p, p, \circledcirc\boxminus p\}$. Then, the successor of q_2 computed by **Cover** is q_1. In `GPVW+`, q_2 does not satisfy $\boxminus p$ imposed by q_1 ($SATISFY(q_2, \boxminus p) = false$) because $\boxminus p \notin q_2$. However, q_2 does satisfy $\boxminus p$ semantically. In this case, `GPVW+` will construct a label-on-state NGW that accepts nothing for $\bigcirc\boxminus p$, which is incorrect.

4.5.2 `Extended MoDeLLa`

`MoDeLLa` basically follows the construction of `LTL2AUT` but uses a different cover computation procedure and three additional improvements. Since the cover computation in `MoDeLLa` also satisfies Formula 4.1, we can extend `MoDeLLa`, resulting in `Extended MoDeLLa`, in the same way with special cares of the additional improvements.

The first improvement in `MoDeLLa` is pruning fair sets, which can be performed after a label-on-state NGW is obtained. Thus, this improvement won't affect our extension.

The second improvement in `MoDeLLa` merges states that have the same $\bigcirc$-formulae and behave the same in the acceptance condition. Thus, a state in `MoDeLLa` may contains several sub-states. In `MoDeLLa`, a state is a To make this improvement work with our backtrace procedure, we merge states if they also have the same $\ominus$- and $\circledcirc$-formulae. Thus, every sub-state in a state have the same backward requirements. Moreover, when refining a state, we refine its sub-states separately as they have different satisfied formulae.



The third improvement is branching postponement, which rewrites $\bigcirc f \vee \bigcirc g$ into $\bigcirc(f \vee g)$ if it is safe. Therefore, in the cover computation, two states $s = F \cup \{\bigcirc f_1, \bigcirc f_2, \ldots, \bigcirc f_n\}$ and $t = G \cup \{\bigcirc g_1, \bigcirc g_2, \ldots, \bigcirc g_m\}$ may be merged to $F \cup \{\bigcirc(f_1 \wedge f_2 \wedge \cdots \wedge f_n \vee g_1 \wedge g_2 \wedge \cdots \wedge g_m)\}$ (with s and t as the sub-states) if the merge is safe and $F = G$. This improvement merges only future formulae and produces larger merged states similar in the first improvement. Thus, we require that states to be merged by branching postponement must have the same backward requirements and we refine sub-states separately.

4.5.3 `Extended Couvreur`

`Couvreur` relies also on tableau rules to expand a formula into several cases that satisfy the formula, which is similar to the cover computation. Unlike those state-based translation algorithms, `Couvreur` puts acceptance conditions on transitions instead of states, produces transition-based NGW (NTGW), and uses BDD (Binary Decision Diagram) [7] to encode automata.

A state in `Couvreur` is a set of formulae to be satisfied by the following transitions and successive states. When computating the successors of a state s, `Couvreur` first expands the formulae in s to a set C, of which an element in C is a possible way to satisfy all formulae in s. Then, for each $c \in C$, `Couvreur` creates a successor $\vec{c}$ and a transition with literals in c as the label. For example, in translating $\Diamond p$, `Couvreur` starts with a single initial state $q_0 = \{\Diamond p\}$ instead of decomposing $\Diamond p$ immediately. Since $\Diamond p$ can be expanded by the tableau rules into $\{\{p\}, \{\bigcirc\Diamond p\}\}$ with two cases to satisfy $\Diamond p$, two transitions t_1 and t_2 are created, of which t_1 has a label p and successor $\{\mathbf{true}\}$ while t_2 has label **true** and successor q_0. The NTGW produced by `Couvreur` is shown in Figure 4.3.

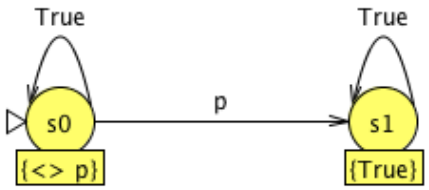


Figure 4.3: The NTGW produced by `Couvreur` for $\Diamond p$

A problem appears immediately is that when doing backtrace, we want to put formulae that should be satisfied in a state. Therefore, a state may contain formulae to be satisfied by the following transitions and successors and formulae to be satisfied by itself. We will be hard to distinguish formulae in a state with different intentions. A solution is to add a $\bigcirc$ operator to every formula to be satisfied next and compute the successors of a state s by decomposing the formulae $\vec{s}$. Besides, after a backtrace, we may add literals to a state. Those literals should be included in all outgoing transitions.

Another problem is that the translation may be incorrect if we backtrace the initial state. For example, consider the translation of $\circleddash p$, which is equivalent to *true*. With the previous solution, the initial state is $q_0 = \{\bigcirc\circleddash p\}$ and the successor candidate of the initial state is $q_1 = \{\circleddash p\}$. Since q_0 does not satisfy p imposed by q_1, we may need to do a backtrace. If we do, we will refine the initial state and put a p in it. Such p will appear in all outgoing transitions of the refined state. As a result, the constructed NTGW will only accept words start with a p, which is incorrect. Thus, when adding an outgoing transition from the initial state to a successor, we always skip the backtrace and really add the transition if the successor contains no $\ominus$-formula. However, self-loop transitions of the initial states may need backtrace because they may appear several steps after the initial state but the backtrace is omitted anyway now. To avoid this problem, we add $\circleddash$**false** to the initial state and

therefore the initial state will has no predecessor.

4.5.4 Extended LTL2BUCHI

Compared to GPVW, MoDeLLa, and Couvreur, the extension to LTL2BUCHI is the most complicated one because it adapts the Node data structures and recursive procedure in GPVW and GPVW+ [34]. A node in LTL2BUCHI contains formulae to be satisfied. In each iteration of the recursive procedure, a formula in a node is expanded by the tableau rules. If the formula is expanded into two disjunctive cases, the node is split into two nodes. If no formula in a node can be expanded, the node is fully expanded. Once a node is fully expanded, if there is no duplicate node, a state will be created for it and a new node will be created to satisfy the $\bigcirc$-formulae of the fully expanded node.

To adapt our backtrace procedure to LTL2BUCHI, we perform backtrace when a node is fully expanded. During the expansion of nodes, the refinement relation are maintained carefully because nodes may not be fully expanded and nodes may be merged into other nodes. When splitting a node n into two nodes n_1 and n_2, if n refines some node m, n_1 and n_2 should also refine m. Moreover, LTL2BUCHI merges nodes that have the same $\bigcirc$-formulae as in MoDeLLa. Unlike MoDeLLa, LTL2BUCHI does not maintain sub-nodes of nodes. Thus, we merge nodes containing past formulae if they are exactly the same.

Both LTL2BUCHI and Couvreur produce NTGWs and put the formula to be translated in the initial state. Therefore, some modifications applied in Couvreur are also required in LTL2BUCHI. These modifications are listed below.

- Incoming transitions of the initial state is forbidded.

- Never refine the initial state.

- Any successor of the initial state cannot contain previous formulae.

4.6 Experimental Results

We have implemented the extended versions of `GPVW`, `LTL2AUT`, `LTL2AUT+`, `MoDeLLa`, `Couvreur`, and `LTL2BUCHI` as part of the GOAL tool [95, 94], which also contains several well-known translation algorithms of other kinds. Totally, there are fourteen translation algorithms, eleven of which are capable of handing past operators. We also implemented an adaptation of the *temporal separation* of past and future operators proposed by Gabbay [29]. Such separation can convert a PTL formula to an equivalent LTL formula, which can later be translated by an LTL translation algorithm such as `LTL2BA`.

In the first experiment, we compared `IncTableau`, `Extended LTL2AUT+`, `Couvreur`, `LTL2BUCHI`, `PLTL2BA`, and `LTL2BA` (with the help of the temporal separation) using 2,500 randomly generated unique formulae with different number of *temporal alternations*. There is a temporal alternation if there is a past (or respectively future) operator immediately inside a future (or respectively past) operator. The number of temporal alternation of a formula is the maximal number of temporal alternation of branches of the parse tree of the formula. For every $i \in \{1, 2, 3, 4, 5\}$, 500 formulae with a length of 20, 2 propositions, and i temporal alternation(s) were generated randomly.

All the automata in the experiment were converted into equivalent Büchi automata. `Tableau`, `Tester`, `Extended GPVW`, and `MoDeLLa` were not selected because preliminary results show that they are not competitive in terms of the state size of

automata. This experiment was run on an Intel Core 2 Duo 2.53 GHz machine with 1 GB of memory allocated to the Java Virtual Machine. A translation process that did not finish in time (10 minutes) was killed. For `LTL2BA`, the time of temporal separation also counts.

Figure 4.4 shows the number of timeout cases of the translation algorithms. While `Extended LTL2AUT+` has the fewest timeout cases, `PLTL2BA` has many more timeout cases than all the others. All our extended algorithms perform better than the others in terms of timeouts. Figure 4.4 also shows that when the number of temporal alternations increase, the translation becomes harder.

Figure 4.5 shows the average number of states of the Büchi automata constructed by the translation algorithms. Same as in the complementation experiments in Section 5.1, we only calculate the state size from effective samples. For 1, 2, 3, 4, and 5 temporal alternations, there are respectively 277, 228, 148, 129, and 86 effective samples. The results show that `Extended Couvreur` performs better than all the others. Although the average length of LTL formulae obtained by temporal separation is 256, which is a big blow up, `LTL2BA` is still quite competitive in terms of state size. However, as the temporal separation has a high complexity and cannot be finished in time in many more cases, it would not be feasible to translate a PTL formula by the combination of the temporal separation and an LTL translation algorithm.

We also conducted a comparative experiment using 5 formulae in real-life benchmarks. The 5 formulae has been appeared in [63] and are shown in Table 4.1. The results are summarized in Table 4.2. All the automata were converted to Büchi automata for comparison. In this small experiment, `LTL2BA` with temporal separation

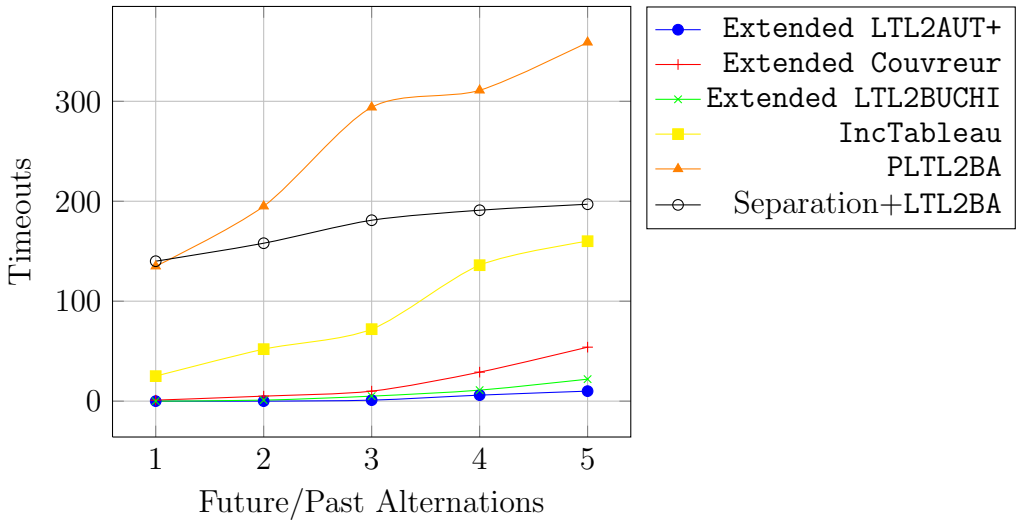


Figure 4.4: A comparison of translation algorithms based on the number of timeouts

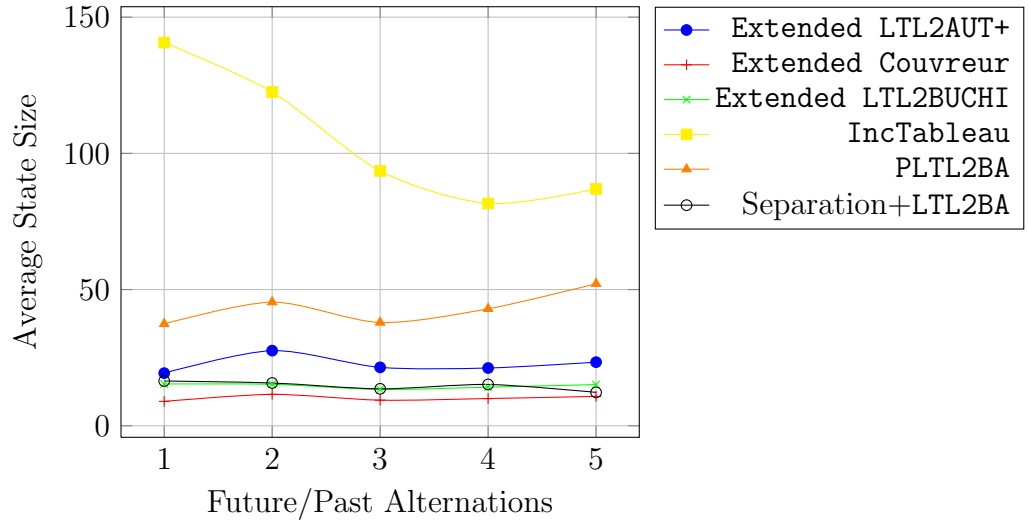


Figure 4.5: A comparison of translation algorithms based on the average state size

No.	Temporal Formula
1	$\neg(\Box(p \rightarrow \ominus\boxminus q))$
2	$\neg(\Diamond\Box(p \rightarrow \diamondminus(q \rightarrow \diamondminus r)))$
3	$\neg(\Box(p \rightarrow p(\ \mathcal{T}\ \neg pq\ \mathcal{T}\ )))$
4	$\neg((\Box p) \rightarrow \Box(q \wedge \ominus(\neg q \wedge \diamondminus(r \wedge \diamondminus(s \wedge \diamondminus t))) \rightarrow \diamondminus(u \wedge \diamondminus(v \wedge \Box w))))$
5	$\neg((\Diamond\Box p \wedge \Box\Diamond q \wedge \Box\Diamond r) \rightarrow \Diamond(s\ \mathcal{S}\ (t\ \mathcal{S}\ (u\ \mathcal{S}\ (v\ \mathcal{S}\ w)))))$

Table 4.1: Five selected formulae for a comparison of translation algorithms

No.	`Extended LTL2AUT+`		`Extended Couvreur`		`Extended LTL2BUCHI`		`IncTableau`		`PLTL2BA`		`LTL2BA`	
	st	tran	st	tran	st	tran	st	tran	st	tran	st	tran
1	6	34	4	24	4	20	9	50	11	86	6	36
2	2	6	2	4	3	7	5	15	4	12	2	4
3	6	27	4	19	3	13	10	49	11	85	3	11
4	-	-	-	-	-	-	-	-	-	-	-	-
5	256	279,936	-	-	4	560	*	*	-	-	4	24
1	4	16	4	16	4	17	4	16	4	16	6	33
2	2	4	2	4	2	4	2	4	3	6	2	4
3	3	10	3	10	3	10	3	10	3	12	3	10
4	-	-	-	-	-	-	-	-	-	-	-	-
5	5	416	-	-	4	352	*	*	-	-	4	22

Table 4.2: Comparison of six translation algorithms without and with simplifications in [86, 24] applied to the final Büchi automata based on the five formulae in Table 4.1. The top 5 formulae are without simplification and the bottom 5 are with simplification. A "-" indicates that the algorithm cannot finish the translation in 10 minutes. A "*" indicates that the algorithm runs out 1 GB memory.

performs quite good, especially in the 5th formula.

4.7 Summary of this Chapter

We presented a way to extend incremental translation algorithms to support full PTL formulae. Our extension preserves the incremental construction of automaton states of the original algorithms such that the resulting automaton is small. Two optimization heuristics, postponed forward expansion of refined states and cover computation with prime implicants, were proposed to make the automaton even smaller. The experimental results showed that our `Extended Couvreur` in average produces smaller automata than (1) the other two algorithms that support past

operators and (2) `LTL2BA` with a temporal sepration procedure.

Chapter 5

Complementation Algorithms

As mentioned in the introduction, the complementation constructions can be classified into four approaches: : Ramsey-based approach [8, 83], determinization-based approach [77, 70, 4, 74], rank-based approach [89, 57, 54], and slice-based approach [47, 101]. In this chapter, we first compare the best construction in each approach based on experimentation. The four representative complementation constructions are `Ramsey` [83], `Safra-Piterman` (`SP` for short) [74], `Rank` [79], and `Slice` [101]. We then propose the following optimization heuristics: simplifying DPWs by simulation (`+S`) and merging equivalent states (`+E`) for `SP`; maximizing the Büchi acceptance set (`+A`) for `Rank`; deterministic decoration (`+D`), reducing transitions (`+R`), and merging adjacent nodes (`+M`) for `Slice`. Experimental results showing the performance of our optimization heuristics and the comparison of the improved constructions are followed by the summary of this chapter.

5.1 Comparison of Complementation Approaches

We chose in the comparison four representative complementation constructions, namely `Ramsey` [83], `SP` [74], `Rank` [79], and `Slice` [101], each of which is considered the most efficient construction in its respective approach. These constructions were implemented in the GOAL tool [94]. We randomly generated 11,000 automata with

Constructions	T	M	Effective Samples	S_R	(Win)	S_L	(Win)	S_L/S_R
$\mathbb{A}_{15}$								
`Ramsey`	4,564	36	2,259	513.85	(0)	30.82	(522.50)	0.060
`SP`	5	0		45.26	(1,843)	2.27	(556.67)	0.050
`Rank`	5,303	0		260.41	(415)	2.79	(649.17)	0.011
`Slice`	3,131	3,213		790.92	(1)	3.03	(530.67)	0.004

Table 5.1: A comparison of the four representative complementation constructions based on $\mathbb{A}_{15}$

an alphabet of size 2 and states of size 15. Among the 11,000 automata of state size 15, denoted by $\mathbb{A}_{15}$, each 100 automata were generated from a combination of 11 transition densities (from 1.0 to 3.0) and 10 acceptance densities (from 0.1 to 1.0). For every generated automaton $A = (\Sigma, Q, q_0, \delta, \mathcal{F})$ with n states, symbol $a \in \Sigma$, transition density r, and acceptance density f, we made $q \in \delta(p, a)$ for $\lceil rn \rceil$ pairs of states $(p, q) \in Q^2$ uniformly chosen at random and added $\lceil fn \rceil$ states to $\mathcal{F}$ uniformly at random. Our parameters were chosen to generate a large set of complementation problems, ranging from easy to hard. The experiment was performed on a cluster at Rice University (http://rcsg.rice.edu/sugar/int/). For each complementation task, we allocated one 2.83-GHz CPU and 1 GB of memory. The timeout of a complementation task was 10 minutes.

The experimental results are summarized in Table 5.1 where T is the total number of timed-out tasks and M is the total number of tasks that run out of memory. Compared to `SP` that has only 5 unfinished tasks, each of `Ramsey`, `Rank`, and `Slice` has around 50% of unfinished tasks. Besides the number of unfinished tasks, we also want to compare the sizes of states of the constructed complements. As these constructions may successfully finish different tasks, a construction may be misleadingly considered to be worse in producing more states because it can finish harder tasks that have much larger complements. Therefore, we only collect state-size informa-

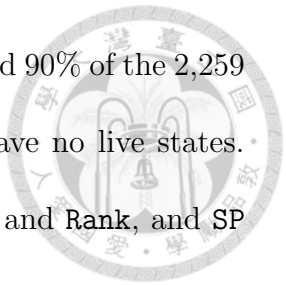
tion from 2,259 *effective samples*, which are tasks finished successfully by all the four constructions. Among the 2,259 effective samples, around 90% of the automata are universal.

The state-size information is shown in the columns S_L and S_R. The column S_R is the average number of reachable states, while S_L is the average number of live states, of the complements. The two columns show that `SP` is the best in average state size. The low S_L/S_R ratio shows that `Rank` and `Slice` create more dead states that can be easily pruned off.

In addition to the number of unfinished tasks and the state-size information, the number of smallest complements produced by a construction among the effective samples is another measure of performance. A construction *wins* in an effective sample w.r.t. S_R (resp., S_L) if the complement produced by the construction is the smallest in terms of reachable states (resp., live states). The Win column of a construction in S_R (resp., S_L) is the fractional share of effective samples where the construction wins w.r.t. S_R (resp., S_L). If k constructions win in an effective sample, each gets $1/k$ shares. The Win columns show that although `Rank` generates more dead states, it produces more complements that are the smallest after pruning dead states.

Constructions	Effective Samples	S_R	(Win)	S_L	(Win)	S_L/S_R
$\mathbb{A}_{15}$						
`Ramsey`	171	1,892.37	(0)	397.10	(0)	0.210
`SP`		36.38	(102.5)	17.77	(34.67)	0.488
`Rank`		156.63	(67.5)	24.61	(127.67)	0.157
`Slice`		422.88	(1)	27.75	(8.67)	0.066

Table 5.2: A comparison of the four representative complementation constructions based on the nonuniversal automata in $\mathbb{A}_{15}$



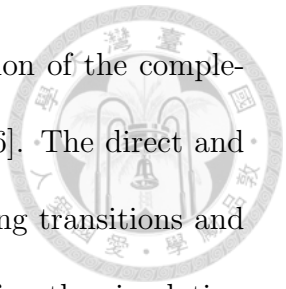
`Rank` and `Slice` become much closer to `SP` in S_L because around 90% of the 2,259 effective samples are universal automata, whose complements have no live states. If we only consider nonuniversal automata, the gaps between `SP` and `Rank`, and `SP` and `Slice` in S_L become larger as shown in Table 5.2.

In a summary, the experimental results show that (1) `SP` is the best in average state size and in the number of finished tasks, (2) `Ramsey` is not competitive in complementation even though it is competitive in universality and containment testing as shown in [3, 25, 26], and (3) `Slice` has the most unfinished tasks (even more than `Ramsey`) and, compared to `SP` and `Rank`, produces many more states. As we will show later, we can improve the performance of `Slice` significantly by employing various optimization heuristics.

Besides the 11,000 automata with 15 states, another 11,000 automata with 10 states and another 11,000 automata with 20 states, denoted by $\mathbb{A}_{10}$ and $\mathbb{A}_{20}$ respectively, were also used in our experiments. We focus on experimental results based on $\mathbb{A}_{15}$ and put the full experimental results in Section 5.6 because $\mathbb{A}_{10}$ contains fewer tough automata especially for `SP` while $\mathbb{A}_{20}$ is too hard to get effective samples. Moreover, the comparisons based on $\mathbb{A}_{10}$ and $\mathbb{A}_{20}$ are basically consistent to those based on $\mathbb{A}_{15}$.

5.2 Safra-Piterman Construction

`SP` performs complementation via several intermediate stages: starting with the given NBW, it computes first an equivalent DPW, then a complement DPW, and finally a complement NBW. We address (1) the simplification of the complement DPW, which results in an NPW, and (2) the conversion from an NPW to an equivalent NBW.



Simplifying DPWs by simulation (`+S`). For the simplification of the complement DPW, we borrow from the ideas of Somenzi and Bloem [86]. The direct and reverse simulation relations they introduced are useful in removing transitions and possibly states of an NBW while retaining its language. We define the simulation relations for an NPW in order to apply the same simplification technique. Given an NPW $(\Sigma, Q, q_0, \delta, \mathcal{F})$ and two states $p, q \in Q$, p is *directly simulated* by q iff (1) for all $p' \in \delta(p, a)$, there is $q' \in \delta(q, a)$ such that p' is directly simulated by q', and (2) $\mathcal{F}(p) = \mathcal{F}(q)$. Similarly, p is *reversely simulated* by q iff (1) for all $p' \in \delta^{-1}(p, a)$, there is $q' \in \delta^{-1}(q, a)$ such that p' is reversely simulated by q', (2) $\mathcal{F}(p) = \mathcal{F}(q)$, and (3) $p = q_0$ implies $q = q_0$. After simplification using simulation relations, as in [86], a DPW may become nondeterministic. Since the complementation of a DPW is much easier than that of an NPW, the simplification by simulation is applied to the complement DPW (in the second stage) but not to the equivalent DPW (in the first stage).

Merging equivalent states (`+E`). As for the conversion from an NPW to an NBW, a typical way in the literature is to direcly apply the conversion from an NRW to an NBW [53, 36] because the parity condition is a special case of the Rabin condition. Here we first review the conversion from an NRW to an NBW adapted for an NPW.

Intuitively, the conversion nondeterministically guesses the minimal even parity passed infinitely often in a run starting from some state. Once a run is guessed to pass a minimal even parity $2k$ infinitely often starting from a state p, every state q in the run after p should have a parity greater than or equal to $2k$ and q is designated as an accepting state in the resulting NBW if it has parity $2k$.

Given an NPW $P = (\Sigma, Q, q_0, \delta, \mathcal{F})$ where $\mathcal{F} : Q \to \{0, 1, \ldots, 2r\}$, the typical conversion constructs the equivalent NBW $A = (\Sigma, S, s_0, \Delta, \mathcal{G})$ where

- $S = Q \times \{0, 2, \ldots, 2r\}$,
- $s_0 = (q_0, 0)$,
- $\Delta : S \times \Sigma \to 2^S$ is the transition function satisfying
  - $(q_j, 2k) \in \Delta((q_i, 0), a)$ iff $q_j \in \delta(q_i, a)$ and
  - $(q_j, 2k) \in \Delta((q_i, 2k), a)$ iff $q_j \in \delta(q_i, a)$ and $\mathcal{F}(q_j) \geq 2k$, and
- $\mathcal{G} = \{(q, 2k) \in S \mid \mathcal{F}(q) = 2k\}$.

A run of A will always look like $(q_0, 0) \cdots (q_{i-1}, 0)(q_i, 2k)(q_{i+1}, 2k) \cdots$ where the transitions from $(q_{i-1}, 0)$ to $(q_i, 2k)$ represent the guess of $2k$ to be the minimal even parity passed infinitely often and from there on $2k$ remains unchanged.

Lemma 5.2.1. *Given an NPW P, the typical conversion constructs an NBW A such that $L(P) = L(A)$.*

Proof. The typical conversion basically follows the conversion in [53] but restricts the Rabin condition to a Rabin chain, which is equivalent to the parity condition of P. Thus, the proof in [53] applies. □

We propose to perform the conversion with states merged and with the start of guessing the minimal even parity $2k$ delayed to a state that has the even parity. Let $P = (\Sigma, Q, q_0, \delta, \mathcal{F})$ be an NPW where $\mathcal{F} : Q \to \{0, 1, \ldots, 2r\}$. We first define an equivalence relation on states with respect to an even parity in order to merge the states in the conversion. Two states p and q are equivalent with respect to an even parity $2k$, denoted by $p \equiv_{2k} q$, iff $\delta(p, a) = \delta(q, a)$ for all $a \in \Sigma$, and either

- $\mathcal{F}(p) = \mathcal{F}(q) = 2k$,
- $\mathcal{F}(p) > 2k$ and $\mathcal{F}(q) > 2k$, or
- $\mathcal{F}(p) < 2k$ and $\mathcal{F}(q) < 2k$.

Let $[q]_{2k} = \{p \mid p \equiv_{2k} q\}$ be the equivalence class of a state q for a parity $2k$. Let $[Q] = \{[q]_{2k} \mid q \in Q \text{ and } 0 \leq k \leq r\}$ denote the set of equivalence classes of states in Q for all even parities.

Given an NPW $P = (\Sigma, Q, q_0, \delta, \mathcal{F})$ where $\mathcal{F} : Q \to \{0, 1, \ldots, 2r\}$, the improved conversion constructs the equivalent NBW $A' = (\Sigma, [Q], [q_0]_0, \Delta, \mathcal{G})$ where

- $\Delta : [Q] \times \Sigma \to 2^{[Q]}$ is the transition function where

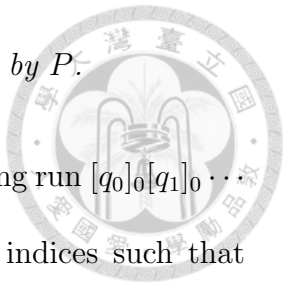
  [TR1] $[q]_{2k} \in \Delta([p]_0, a)$ iff there exists $q' \in [q]_{2k}$ such that $q' \in \delta(p, a)$ and either $k = 0$ or $\mathcal{F}(q') = 2k$,

  [TR2] $[q]_{2k} \in \Delta([p]_{2k}, a)$ iff there exists $q' \in [q]_{2k}$ such that $q' \in \delta(p, a)$ and $\mathcal{F}(q') \geq 2k$, and

- $\mathcal{G} = \{[q]_{2k} \in [Q] \mid \mathcal{F}(q) = 2k\}$.

Lemma 5.2.2. *If a word w is accepted by P, then it is accepted by A'.*

Proof. Let w be a word accepted by P and ρ an accepting run $q_0 q_1 \cdots$ of P on w. Suppose $2k$ is the minimal even parity passed infinitely often in ρ after some state q_i of parity $2k$. By the construction, there is a run $\rho' = [q_0]_0[q_1]_0 \cdots [q_{i-1}]_0[q_i]_{2k}[q_{i+1}]_{2k} \cdots$ of A' on w. The transitions before and after $[q_i]_{2k}$ follow the transition rules TR1 and TR2 respectively. Since $2k$ is the minimal even parity passed infinitely often in ρ, there are infinitely many $[q_m]_{2k}$'s $(i \leq m)$ in ρ' such that $\mathcal{F}(q_m) = 2k$ and $[q_m]_{2k} \in \mathcal{G}$. Thus, ρ' is an accepting run of A' on w. □



Lemma 5.2.3. *If a word w is accepted by A', then it is accepted by P.*

Proof. Let $w = a_0a_1\cdots$ be a word accepted by A' and ρ an accepting run $[q_0]_0[q_1]_0\cdots [q_{i-1}]_0[q_i]_{2k}[q_{i+1}]_{2k}\cdots$ of A' on w. Let M be an infinite set of indices such that $m \in M$ iff $[q_m]_{2k} \in \mathcal{G}$. By the construction, we can find a state $q'_1 \in [q_1]_0$ such that $q'_1 \in \delta(q_0, a_0)$. Starting from q'_1, by the construction and by the definition of equivalence classes, we can find a state $q'_2 \in [q_2]_0$ such that $q'_2 \in \delta(q'_1, a_1)$ and so on. Therefore, there is a run $\rho' = q_0q'_1q'_2\cdots q'_iq'_{i+1}\cdots$ of P on w such that $q'_j \in [q_j]_0$ for $0 < j < i$ and $q'_j \in [q_j]_{2k}$ for $j \geq i$. By the transition function and the equivalence relation, $\mathcal{F}(q'_j) \geq 2k$ for $j \geq i$ and $\mathcal{F}(q'_m) = 2k$ for all $m \in M$. Hence, $2k$ is the minimal even parity passed infinitely often in ρ' after q_i and ρ' is an accepting run of P on w. □

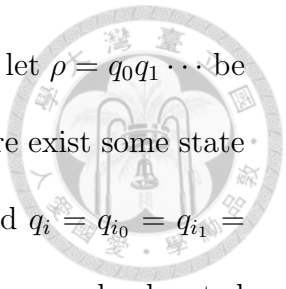
Theorem 5.2.4. $L(P) = L(A')$.

Proof. The result follows directly from Lemmas 5.2.2 and 5.2.3. □

5.3 Rank-Based Construction

Maximizing the Büchi acceptance set (`+A`). As stated in Section 2.2, the ranks for the rank-based approach are bounded by $2(n - |\mathcal{F}|)$. The larger $\mathcal{F}$ is, the fewer the ranks are. Thus, we propose to maximize the acceptance set of the input NBW before complementation based on the following theorem without changing its language, states, or transition function.

Theorem 5.3.1. *Let $A = (\Sigma, Q, q_0, \delta, \mathcal{F})$ and $A' = (\Sigma, Q, q_0, \delta, \mathcal{G})$ be two NBWs where $\mathcal{G} \supseteq \mathcal{F}$. Then, $L(A) = L(A')$ if for all $q \in \mathcal{G}$, every elementary cycle containing q also contains at least one state in $\mathcal{F}$.*



Proof. Since $\mathcal{G} \supseteq \mathcal{F}$, $L(A) \subseteq L(A')$. To prove $L(A) \supseteq L(A')$, first let $\rho = q_0 q_1 \cdots$ be an accepting run of A' on some word w. Since ρ is accepting, there exist some state $q_i \in \rho$ and infinite indices $i_0 < i_1 < i_2 < \cdots$ such that $q_i \in \mathcal{G}$ and $q_i = q_{i_0} = q_{i_1} = q_{i_2} = \cdots$. For all $j \geq 0$, the sequence of states $q_{i_j} q_{i_j+1} \ldots q_{i_{j+1}}$ forms a cycle, denoted by C_j. As $q_i \in \mathcal{G}$ and a cycle is formed by elementary cycles, in each C_j, there is some state $q_{k_j} \in \mathcal{F}$. Since there are infinitely many C_j's but Q is finite, there exists some state $q_k \in \mathcal{F}$ occurring infinitely many times in ρ. Thus, ρ is an accepting of A on w and $L(A) \supseteq L(A')$. □

The elementary cycles of an automaton can be found by the algorithm in [44] with a time complexity $O((n+e)(c+1))$ and a space complexity $O(n+e)$ where n is the number of states, e the number of transitions, and c the number of elementary cycles in the automaton. [1]

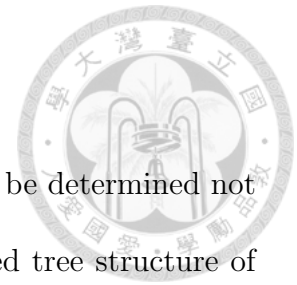
This heuristic can also be applied to other complementation approaches as it maximizes the acceptance set of the input NBW before complementation. We will show the improvement made by this heuristic for `SP`, `Rank`, and `Slice` later in Section 5.5.

5.4 Slice-Based Construction

In this section, we first introduce some definitions used in the slice-based constructions, then the basic `Slice` construction, which is followed by the proposed optimization heuristics and the improved `Slice` construction.

[1] Instead of finding elementary cycles, our implementation makes a state accepting if the state cannot go back to itself without passing an accepting state in $\mathcal{F}$, which is checked by a depth-first search.

5.4.1 Definitions



Consider an NBW A. The acceptance of an infinite word w can be determined not only by the sequential runs of A on w but also by an aggregated tree structure of those runs. Depending on different ways of aggregation, different tree structures can be defined.

The *run tree* of A on w is a tree where a (full) branch corresponds to a run of A on w and there is a corresponding branch for each run of A on w. To determine whether w is accepted by A, one needs only pay attention to the distinction between accepting and non-accepting states, and a run tree may be simplified or abstracted to leave just this much detail. The *split tree* of A on w is a binary tree that abstracts the run tree by grouping accepting children and nonaccepting children of a tree node respectively into a left child and a right child of the node. The word w is accepted by A if there is a *left-recurring branch* in the split tree of A on w while a branch is left-recurring if the branch goes left infinitely many times. Both run trees and split trees suffer the problem of unbounded tree width, which motivates the next tree structure. The *reduced split tree* of A on w is a binary tree obtained from the split tree of A on w by removing a state from a node if it also occurs in a node to the left on the same level; a node is removed if it becomes empty. Similarly, w is accepted by A if there is a left-recurring branch in the reduced split tree of A on w. Each a split tree or a reduced split tree can be represented by a sequence of *slices* where a slice is a sequence of sets of states representing all nodes on a same level of the tree from left to right.

Consider the NBW in Figure 5.1 where the alphabet is $\{p, \neg p\}$, the initial state is q_0, and the acceptance condition is $\{q_1\}$. The split tree and the reduced split tree

of the NBW on the accepted word $p\neg pp^\omega$ are shown in Figure 5.2.

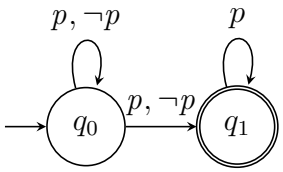
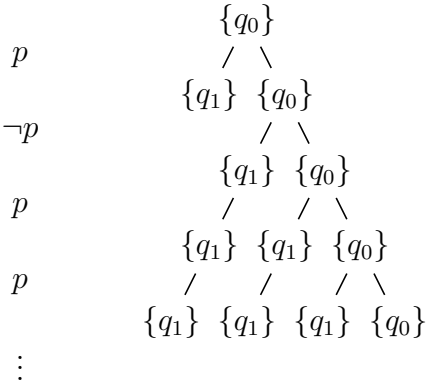
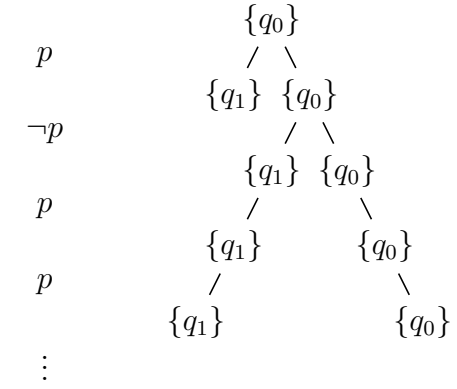


Figure 5.1: An NBW where the alphabet is $\{p, \neg p\}$, the initial state is q_0, and the acceptance condition is $\{q_1\}$



(a) the split tree of the NBW in Figure 5.1 on the accepted word $p\neg pp^\omega$



(b) the reduced split tree of the NBW in Figure 5.1 on the accepted word $p\neg pp^\omega$

Figure 5.2: Examples of a split tree and a reduced split tree

Let $A = (\Sigma, Q, q_0, \delta, \mathcal{F})$ be an NBW and D the set $\{0, *, 1\}$. An *undecorated slice* over Q is a finite, pairwise disjoint, sequence $Q_0 \cdots Q_{n-1}$ of non-empty subsets of Q. A *decorated slice* over Q is a finite sequence $(Q_0, d_0) \cdots (Q_{n-1}, d_{n-1})$ where $Q_0 \cdots Q_{n-1}$ form an undecorated slice and $d_i \in D$ for $i < n$. The i-th node of a slice s is denoted by $s(i)$ and the number of nodes of a slice is denoted by $|s|$. The empty sequence, denoted by $\perp$, is a special slice considered both undecorated and decorated. The set of slices over Q is denoted by $S = S^u \cup S^d$ where S^u is the set of undecorated slices and S^d the set of decorated slices. Let $s = (Q_0, d_0) \cdots (Q_{n-1}, d_{n-1}) \in S^d$. Define $s_{\downarrow Q} = Q_0 \cdots Q_{n-1}$ to be the undecorated version of s and $s_{\downarrow D} = \{d_0, \ldots, d_{n-1}\}$. We say s is a *reset slice* iff $0 \notin s_{\downarrow D}$ and s is *doomed* iff $1 \notin s_{\downarrow D}$. In particular, $\perp$ is a reset slice and is doomed.

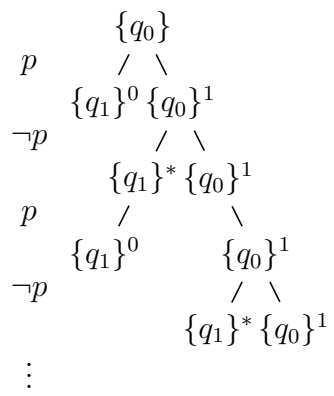


Figure 5.3: A decorated reduced split tree of the NBW in Figure 5.1 on the rejected word $(p\neg p)^\omega$

5.4.2 The Basic `Slice` Construction

The central idea of `Slice` is based on the following lemma [47].

Lemma 5.4.1. *A word w is rejected by an NBW A iff the reduced split tree of A on w has a* cutoff*, which is a level i such that after the i-th level in the reduced split tree, all the left children are in finite branches.*

Note that a reduced split tree may have infinitely many cutoffs.

As an example, the reduced split tree of the NBW in Figure 5.1 on a rejected word $(p\neg p)^\omega$ is shown in Figure 5.3 where the superscripts 0, $*$, and 1 are decorations to explained later. It can be ssen that after the first level of the reduced split tree, all the accepting states of the NBW are in finite branches.

Based on Lemma 5.4.1, `Slice` constructs a complement with slices as states to accept all reduced split trees of an input NBW on rejected words by guessing the cutoffs nondeterministically. A decoration scheme is applied to verify whether a slice is on a cutoff. The transition relation of the complement is divided into two phases. In the first phase, the transition relation is based on the evolution of reduced split trees. When the construction nondeterministically chooses a slice as the slice on

some cutoff, it guesses the decorations of the nodes in the slice and goes from the first phase to the second phase, where the decorations are verified.

A node in a slice can be decorated by 1, 0, or $*$. Intuitively, the decoration 1 indicates that a node is in an infinite branch of a reduced split tree. The decoration 0 indicates that the descendants of a node die out eventually before the next *reset slice*, which is a slice with no node decorated by 0. The decoration $*$ has the same meaning as 0 but the check is put on hold after the next reset slice where the children of $*$-nodes are decorated by 0. Shown in Figure 5.4 are the decoration rules, which enforce that when reset slices are passed infinitely many times, descendants of the left children after decoration will eventually die out.

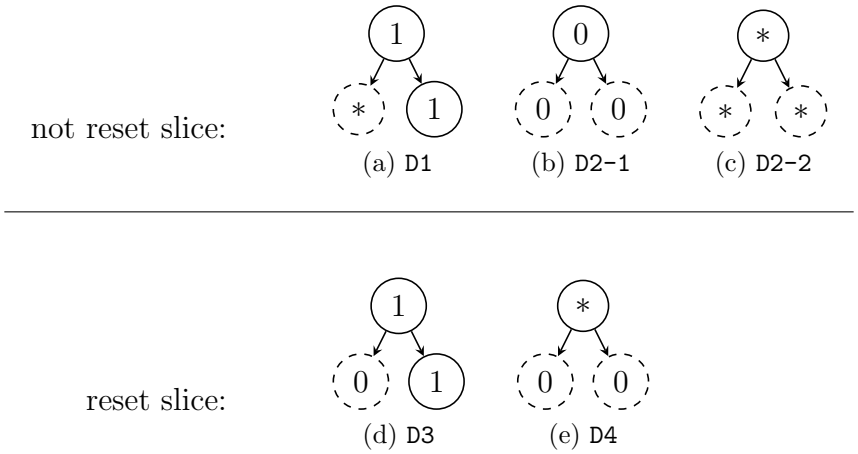


Figure 5.4: The decoration rules `D1`, `D2` (which consists of `D2-1` and `D2-2`), `D3`, and `D4` applied in the basic `Slice` construction. The upper three rules are applied only to non-reset slices while the lower two rules only to reset slices. A node in a slice is represented by a circle in which the label denotes the decoration of the node. As an example, when rule `D3` is applied to a reset slice, the left child of an 1-node is decorated by 0 while the right child is decorated by 1. A child can be absent when applying a rule if it is dashed and otherwise it must exist.

The reason why both 0 and $*$ are used for nodes on finite branches is that when we focus on the sequence of slices that form a reduced split tree, we want to distinguish a node that just died on the previous level between the same node that is just born

on the current level. Otherwise, reset slices may not appear if nodes decorated by 0 are always born on a level immediately after they died out on the previous level. For example, consider the decorated reduced split tree in Figure 5.3. If the decoration $*$ is replaced by 0, the reduced split tree will no longer contain reset slices because $\{q_1\}^0$ appears on every level after the root, but actually $\{q_1\}^0$ dies on every odd level and is born on every even level.

Let $A = (\Sigma, Q, q_0, \delta, \mathcal{F})$ be an NBW. The complement constructed by `Slice` is $A' = (\Sigma, S, s_0, \Delta, \mathcal{G})$ where

- $s_0 = \{q_0\}$,
- $\Delta = S \times \Sigma \to 2^S$ is the transition function described below, and
- $\mathcal{G} = \{s \in S^d \mid s \text{ is a reset slice}\}$.

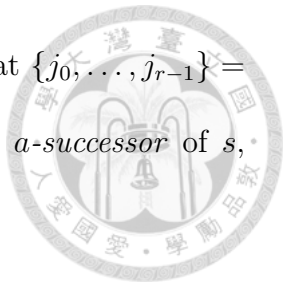
For all $s \in S$ and $a \in \Sigma$, $\Delta(s, a)$ is defined as follows:

- $\Delta(s, a) = \{\delta_u(s, a)\} \cup \delta_g(s, a)$ if $s \in S^u$.
- $\Delta(s, a) = \{\delta_d(s, a)\}$ if $s \in S^d$.

The functions δ_u, δ_g, and δ_d correspond respectively to the transition functions in the first phase, from the first phase to the second phase, and in the second phase.

- The transition function $\delta_u : S^u \times \Sigma \to S^u$ represents the first phase of `Slice` with $\delta_u(s, a)$ giving the next level of s in a reduced split tree with respect to the symbol a. Let $s = Q_0 \cdots Q_{n-1} \in Q^u$ and $a \in \Sigma$. Define $s' = Q'_0 \cdots Q'_{2n-1}$ such that for $i < n$,
  - $Q'_{2i} = (\cup_{q \in Q_i} \delta(q, a) \cap \mathcal{F}) - \bigcup_{j<2i} Q'_j$, and
  - $Q'_{2i+1} = (\cup_{q \in Q_i} \delta(q, a) - \mathcal{F}) - \bigcup_{j<2i} Q'_j$.

By removing $\emptyset$ from s', we can find $j_0 < \cdots < j_{r-1}$ such that $\{j_0, \ldots, j_{r-1}\} = \{j < 2n \mid Q'_j \neq \emptyset\}$. The result $Q_{j_0} \cdots Q_{j_{r-1}}$ is called an *a-successor* of s, denoted by $\delta_u(s, a)$.



- The transition function $\delta_g : S^u \times \Sigma \to 2^{S^d}$ is applied when `Slice` nondeterministically goes from the first phase to the second phase and starts decoration. In this transition, it labels the children of an undecorated slice nondeterministically by 0 or 1. Thus for $s \in S^u$, $a \in \Sigma$, and $s' \in S^d$, $s' \in \delta_g(s, a)$ iff ${s'}_{\downarrow Q} = \delta_u(s, a)$ and ${s'}_{\downarrow D} \subseteq \{0, 1\}$.

- The transition function $\delta_d : S^d \times \Sigma \to S^d$ represents the second phase of `Slice` where it verifies the decorations by evolving decorated slices in the following way. Let $s = (Q_0, d_0) \cdots (Q_{n-1}, d_{n-1}) \in S^d$, $a \in \Sigma$, and $s' = (Q'_{j_0}, d'_{j_0}) \cdots (Q'_{j_r-1}, d'_{j_r-1}) \in S^d$ where j's and Q'_j's are defined as in the definition of δ_u, i.e., ${s'}_{\downarrow Q} = \delta_u(s_{\downarrow Q}, a)$. The decorated slice s' is an a-successor of s, denoted by $\delta_d(s, a)$, iff the following two conditions are satisfied:

  [C1] for all $i < n$ with $d_i = 1$, $Q'_{2i+1} \neq \emptyset$,

  [C2] d'_j's are decorated by the following rules:

    [D1] If s is not a reset slice and $d_i = 1$, then $d'_{2i} = *$ and $d'_{2i+1} = 1$.

    [D2] If s is not a reset slice and $d_i \in \{0, *\}$, then $d'_{2i} = d_i$ and $d'_{2i+1} = d_i$.

    [D3] If s is a reset slice and $d_i = 1$, then $d'_{2i} = 0$ and $d'_{2i+1} = 1$.

    [D4] If s is a reset slice and $d_i = *$, then $d'_{2i} = 0$ and $d'_{2i+1} = 0$.

Theorem 5.4.2. *Given an NBW A, the basic* `Slice` *construction produces an NBW A' with $L(A') = \overline{L(A)}$ [101].*

5.4.3 The Improved `Slice` Construction

We first describe three optimization heuristics applied to the basic `Slice` construction and then the resulting improved `Slice` construction.

Deterministic decoration (`+D`). The first heuristic uses 1 to label nodes that *may* (rather than *must*) be in an infinite branch of a reduced split tree and only verifies the condition `C2` in the second phase. Thus, all nodes could be decorated by 1 in the guesses. However, since the first evolution of the second phase always labels a left (accepting) child by 0 and a right (nonaccepting) child by 1, we actually decorate accepting nodes by 0 and nonaccepting nodes by 1 in the guesses. Formally, let $A = (\Sigma, Q, q_0, \delta, \mathcal{F})$ be an NBW. We define $\delta'_g : S^u \times \Sigma \to S^d$ and $\delta'_d : S^d \times \Sigma \to S^d$, which refine respectively δ_g and δ_d based on this heuristic.

- Let $s = Q_0 \cdots Q_{n-1} \in S^u$, $a \in \Sigma$, and $s' = (Q'_{j_0}, d'_{j_0}) \cdots (Q'_{j_{r-1}}, d'_{j_{r-1}}) \in S^d$ where j's and Q'_j's are defined as in the basic `Slice` construction, i.e., $s'_{\downarrow Q} = \delta_u(s, a)$. Then $s' = \delta'_g(s, a)$ iff for all $i < n$, $d'_{2i} = 0$ and $d'_{2i+1} = 1$.

- The transition function δ'_d is the same as δ_d in the basic `Slice` construction except that the condition C1 of δ_d is not required to be satisfied.

This heuristic results in deterministic decoration. The only nondeterminism comes from choosing when to start decorating.

Reducing transitions (`+R`). The second heuristic relies on the observation that if a run ends up in the empty sequence $\perp$, the run will stay in $\perp$ forever and we never need to decorate the run because it can reach $\perp$ without any decoration. Thus we do not allow transitions from decorated slices other than $\perp$ to $\perp$ or from any slice

to doomed slices other than $\perp$; recall that a slice is doomed if it has no node labeled by 1, i.e., every run through a doomed slice is expected to reach $\perp$.

Merging adjacent nodes (`+M`). The third heuristic recursively merges adjacent nodes decorated all by 0 or all by $*$. The observation is that they are all guessed to have a finite number of descendants and their successors will have the same decoration, either 0 or $*$. Let $s = (Q_0, d_0) \cdots (Q_{n-1}, d_{n-1}) \in S^d$. Based on this heuristic, we can recursively merge adjacent nodes (Q_i, d_i) and (Q_{i+1}, d_{i+1}) in s when $d_i = d_{i+1} = 0$ or $d_i = d_{i+1} = *$. We call the result a *merged slice* of s and denote it by $merge(s)$.

Given an NBW $A = (\Sigma, Q, q_0, \delta, \mathcal{F})$, the improved `Slice` with all optimization heuristics in this section constructs the complement $A' = (\Sigma, S, s_0, \Delta, \mathcal{G})$ where

- $s_0 = \{q_0\}$,
- $\Delta = S \times \Sigma \to 2^S$ is the transition function described below, and
- $\mathcal{G} = \{s \in S^d \mid s \text{ is a reset slice}\}$.

For all $s \in S$ and $a \in \Sigma$, $\Delta(s, a)$ is defined as follows:

- $\Delta(s, a) = \{\delta_u(s, a)\}$ if $s \in S^u$ and $\delta'_g(s, a)$ is doomed.
- $\Delta(s, a) = \{\delta_u(s, a), merge(\delta'_g(s, a))\}$ if $s \in S^u$ and $\delta'_g(s, a)$ is not doomed.
- $\Delta(s, a) = \{merge(\delta'_d(s, a))\}$ if $s \in S^d$ and $\delta'_d(s, a)$ is not doomed.

Before proving the correctness of the improved `Slice` construction, we define another merge function $merge_{i,j}$ that will be used in the proof. For a decorated slice s, $merge_{i,j}(s)$ is a slice obtained from s by merging as many as possible and

at most j consecutive mergible nodes starting from the i-th pair of mergible nodes. For example, if $s = (Q_0, 0)(Q_1, 0)(Q_2, 0)(Q_3, 0)(Q_4, 0)$, then $merge_{1,2}(s) = (Q_0 \cup Q_1, 0)(Q_2, 0)(Q_3, 0)(Q_4, 0)$ and $merge_{2,3}(s) = (Q_0, 0)(Q_1 \cup Q_2 \cup Q_3, 0)(Q_4, 0)$. By this definition, $merge(merge_{i,j}(s)) = merge(s)$ for any i, j, and decorated slice s.

Lemma 5.4.3. *For a decorated slice $s \in S^d$ and a symbol $a \in \Sigma$, there exist some i and j such that $\delta'_d(merge_{1,2}(s), a) = merge_{i,j}(\delta'_d(s, a))$.*

Proof. Let $s = (Q_0, d_0) \cdots (Q_{n-1}, d_{n-1})$ and $t = (Q'_0, d'_0) \cdots (Q'_{2n-1}, d'_{2n-1})$ be the a-successor of s before removing empty nodes. Assume $d_i = d_{i+1} \in \{0, *\}$, and (Q_i, d_i) and (Q_{i+1}, d_{i+1}) are the first mergible pair of nodes in s. By the decoration rules `D2` and `D4`, $d'_{2i} = d'_{2i+1} = d'_{2i+2} = d'_{2i+3} \in \{0, *\}$. Then, $merge_{1,2}(s) = (Q_0, d_0) \cdots (Q_{i-1}, d_{i-1})(Q_i \cup Q_{i+1}, d_i)(Q_{i+2}, d_{i+2}) \cdots (Q_{n-1}, d_{n-1})$ and its a-successor before removing empty nodes is $(Q'_0, d'_0) \cdots (Q'_{2i-1}, d'_{2i-1})(Q_{2i} \cup Q_{2i+1} \cup Q_{2i+2} \cup Q_{2i+3}, d_{2i})(Q'_{2i+4}, d'_{2i+4}) \cdots (Q'_{2n-1}, d'_{2n-1})$, denote by t'. Since $d'_{2i} = d'_{2i+1} = d'_{2i+2} = d'_{2i+3} \in \{0, *\}$, (Q'_{2i}, d'_{2i}), (Q'_{2i+1}, d'_{2i+1}), (Q'_{2i+2}, d'_{2i+2}), and (Q'_{2i+3}, d'_{2i+3}) are mergible. Suppose (Q'_{2i}, d'_{2i}) and (Q'_{2i+1}, d'_{2i+1}) are the k-th mergible pair of nodes. Then, $merge_{k,4}(t) = t'$. As $\delta'_d(s, a)$ and $\delta'_d(merge_{1,2}(s), a)$ are derived respectively from t and t' by removing empty nodes, we can found some i and j ($0 \leq j \leq 4$) such that $\delta'_d(merge_{1,2}(s), a) = merge_{i,j}(\delta'_d(s, a))$. □

Lemma 5.4.4. *Let $s \in S^d$ be a decorated slice and $a \in \Sigma$ a symbol. If $\delta'_d(s, a)$ is not doomed, then $\Delta(merge(s), a) = \{merge(\delta'_d(s, a))\}$.*

Proof. We prove by induction on the number of mergible pairs in s. The base case is that s has no mergible pair, which implies that $merge(s) = s$. Thus,

$$\begin{array}{rcll} \Delta(merge(s), a) & = & \{merge(\delta'_d(merge(s), a))\} & \text{(by the definition of } \Delta) \\ & = & \{merge(\delta'_d(s, a))\} & \text{(by } merge(s) = s) \end{array}$$

Assume the hypothesis holds for any slice that has n mergible pairs and consider a slice s that has $n+1$ mergible pairs. Since $merge_{1,2}(s)$ has n mergible pairs, we know that:

$$\begin{array}{rcll}
\Delta(merge(s), a) & = & \Delta(merge(merge_{1,2}(s)), a) & \text{(by the definition of } merge_{i,j}\text{)} \\
& = & \{merge(\delta'_d(merge_{1,2}(s), a))\} & \text{(by the induction hypothesis)} \\
& = & \{merge(merge_{i,j}(\delta'_d(s, a)))\} & \text{for some } i \text{ and } j \\
& & & \text{(by Lemma 5.4.3)} \\
& = & \{merge(\delta'_d(s, a))\} & \text{(by the definition of } merge_{i,j}\text{)}
\end{array}$$

□

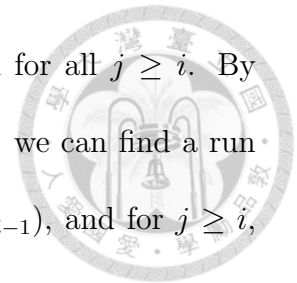
Theorem 5.4.5. *Given an NBW $A = (\Sigma, Q, q_0, \delta, \mathcal{F})$, the improved* `Slice` *construction produces an NBW $A' = (\Sigma, S, s_0, \Delta, \mathcal{G})$ with $L(A') = \overline{L(A)}$.*

Proof. We first prove that if a word $w = a_0 a_1 \cdots$ is rejected by A, w is accepted by A'. Let $T = s_0 s_1 \cdots$ be the reduced split tree of A on w where $s_{j+1} = \delta_u(s_j, a_j)$ for all j.

- Case 1: There is no run of A on w. Then, there exists some i such that $s_j = \perp$ for all $j \geq i$. By the construction of the improved `Slice`, $s_0 s_1 \cdots s_{i-1} \perp^\omega$ is an accepting run of w on A'. Thus, w is accepted by A'.

- Case 2: There is at least one run of A on w. Since w is rejected by A, by Lemma 5.4.1, there exists some cutoff i such that for all $j \geq i$, all accepting states of A in s_j belong to finite branches of T. Then, we can construct a sequence of slices $s_0 s_1 \cdots s_{i-1} t_i t_{i+1} \cdots$ where $t_j \in S^d$ such that

  - $t_i = \delta'_g(s_{i-1}, a_{i-1})$,
  - $t_{j+1} = \delta'_d(t_j, a_j)$ for $j \geq i$, and
  - $t_{j \downarrow Q} = s_j$ for $j \geq i$.



As there is at least one run of A on w, t_j is not doomed for all $j \geq i$. By Lemma 5.4.4 and the construction of the improved `Slice`, we can find a run $\rho = s_0 s_1 \cdots s_{i-1} u_i u_{i+1} \cdots$ of A' on w where $u_i \in \Delta(s_{i-1}, a_{i-1})$, and for $j \geq i$, $u_{j+1} \in \Delta(u_j, a_j)$ and $u_j = \mathit{merge}(t_j)$. Since all accepting states of A in s_j for $j \geq i$ belong to finite branches of T and these states are decorated by either 0 or $*$ in both t_j and u_j, we can find infinitely many reset slices in ρ by the decoration rules. Thus, ρ is accepting and w is accepted by A'.

We then prove that if a word $w = a_0 a_1 \cdots$ is accepted by A', w is rejected by A. Let $\rho = s_0 s_1 \cdots$ be an accepting run of A' on w.

- Case 1: $\perp \in \rho$. In this case, there is some i such that $s_j = \perp$ for all $j \geq i$. Thus, there is no run of A on w and w is rejected by A.

- Case 2: $\perp \notin \rho$. Let $T = t_0 t_1 \cdots$ be the reduced split tree of A on w. Assume s_i is the first decorated slice in ρ. Then, $s_j = t_j$ for $j < i$ and s_j is not doomed for $j \geq i$. By Lemma 5.4.4 and the construction of the improved `Slice`, there is a sequence $\rho' = s_0 s_1 \cdots s_{i-1} u_i u_{i+1} \cdots$ where $u_j \in S^d$ for $j \geq i$ such that

  - $u_i = \delta'_g(s_{i-1}, a_{i-1})$,
  - $u_{j+1} = \delta'_d(u_j, a_j)$ for $j \geq i$, and
  - $\mathit{merge}(u_j) = s_j$ and $u_{j\downarrow Q} = t_j$ for $j \geq i$.

  Since ρ is accepting, there are infinitely many reset slices in ρ as well as in ρ'. Based on the construction of the improved `Slice`, all accepting states of A are decorated by either 0 or $*$ in ρ, the decoration of 0-nodes and $*$-nodes remains unchanged before the next reset slice, and the decoration of $*$-nodes becomes 0 after a reset slice. Thus, after u_i in ρ, all these accepting states belong to finite

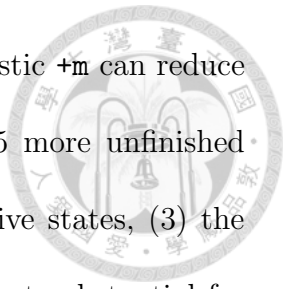
branches. Since the *merge* function does not change any deocration, all these accepting states belong to finite branches after u_i in ρ'. As ρ' and T only differ in decorations, all the accepting states of A belong to finite branches after the i-th level in T. Hence, w is rejected by A according to Lemma 5.4.1. □

5.5 Experimental Results

The heuristics proposed in Sections 5.2, 5.3, and 5.4 were implemented in the GOAL tool. For `Ramsey`, there is a naive optimization which minimizes the classic finite automata before composing the NBWs to construct the complement. This optimizaiton, refered to as `+m`, was also implemented in GOAL. We used the same 11,000 automata as in Section 5.1 as the test bench. The results showing the improvement made by the heuristics are summarized in Table 5.3 where the Ratio columns are ratios with respect to the original construction and the other columns have the same meanings as in Section 5.1.

Constructions	T	M	Effective Samples	S_R	(Ratio)	S_L	(Ratio)	S_L/S_R
`Ramsey`	4,564	36	6,388	594.68	(1.00)	22.78	(1.00)	0.04
`Ramsey+A`	4,557	33		595.59	(1.00)	22.54	(0.99)	0.04
`Ramsey+m`	3,126	2		372.08	(0.63)	11.19	(0.49)	0.03
`Ramsey+Am`	3,119	2		371.06	(0.62)	11.02	(0.48)	0.03
`SP`	5	0	10,977	256.25	(1.00)	58.72	(1.00)	0.23
`SP+A`	5	0		228.40	(0.89)	54.33	(0.93)	0.24
`SP+S`	12	9		179.82	(0.70)	47.35	(0.81)	0.26
`SP+E`	11	0		194.95	(0.76)	45.47	(0.77)	0.23
`SP+ASE`	13	7		138.97	(0.54)	37.47	(0.64)	0.27
`Rank`	5,303	0	5,697	569.51	(1.00)	33.96	(1.00)	0.06
`Rank+A`	3,927	0		181.05	(0.32)	28.41	(0.84)	0.16
`Slice`	3,131	3,213	4,514	1,088.72	(1.00)	70.67	(1.00)	0.06
`Slice+A`	2,611	2,402		684.07	(0.63)	64.94	(0.92)	0.09
`Slice+D`	1,119	0		276.11	(0.25)	117.32	(1.66)	0.42
`Slice+R`	3,081	3,250		1,028.42	(0.94)	49.58	(0.70)	0.05
`Slice+M`	2,813	3,360		978.01	(0.90)	57.85	(0.82)	0.06
`Slice+ADRM`	228	0		102.57	(0.09)	36.11	(0.51)	0.35

Table 5.3: A comparison of each complementation construction with its improved versions



The experimental results in Table 5.3 show that (1) the heuristic +m can reduce states down to around one half for Ramsey, (2) SP+ASE has 15 more unfinished tasks but creates just around one half of reachable states and live states, (3) the improvement made by +A is limited for Ramsey, SP, and Slice but substantial for Rank in helping finish 1,376 more tasks and avoid the creation of around 2/3 dead states, (4) the heuristic +D is quite useful in reducing the reachable states down to 1/4 for Slice but produces more live states, and (5) Slice+ADRM finishes 6,116 more tasks and significantly reduces the reachable states to 1/10 and live states to one half.

We also compared the four constructions with all optimization heuristics in Sections 5.2, 5.3, and 5.4 based on 4,851 effective samples and list the results on the top of Table 5.4. The table shows that SP+ASE still outperforms the other three in the average state size and in running time. Table 5.4 also shows the following changes made by our heuristics in the comparison: (1) SP+ASE outperforms Rank+A in the number of smallest complements after pruning dead states, and (2) Slice+ADRM creates fewer reachable states than Rank+A in average, and finishes more tasks than Rank+A and Ramsey+Am.

As the heuristic of preminimization applied to the input automata, denoted by +P, is considered to help the nondeterministic constructions more than the deterministic one, we also compare the four improved constructions with preminimization and summarize the results on the bottom of Table 5.4. We only applied the preminimization implemented in the GOAL tool, namely the simplification by simulation in [86]. According to our experimental results, the preminimization does improve Ramsey, Rank, and Slice more than SP in the complementation but does not close too much

Constructions	T	M	Effective Samples	S_R	(Win)	S_L	(Win)	S_L/S_R
$\mathbb{A}_{15}$								
Ramsey+Am	3,119	2	4,851	666.86	(0.0)	251.53	(895.75)	0.38
SP+ASE	13	7		23.88	(4,772.5)	5.77	(1,675.42)	0.24
Rank+A	3,927	0		384.35	(20.0)	11.38	(1,138.42)	0.03
Slice+ADRM	228	0		185.02	(58.5)	9.65	(1,141.42)	0.05
$\mathbb{A}_{15}$								
Ramsey+PAm	2,825	6	5,618	479.99	(17.50)	225.08	(1,031.25)	0.47
SP+PASE	12	7		19.47	(3,820.67)	5.89	(1,698.75)	0.30
Rank+PA	3,383	0		232.63	(875.67)	10.68	(1,476.75)	0.05
Slice+PADRM	216	0		135.74	(904.17)	9.12	(1,411.25)	0.07

Table 5.4: A comparison of the four improved complementation constructions based on $\mathbb{A}_{15}$ without and with preminimization

the gap between them in the comparison, though there are other preminimization techniques that we did not implement and apply in the experiment.

The comparisons of the four improved constructions based on the nonuniversal automata in $\mathbb{A}_{15}$ without and with preminimization are summarized in Table 5.5. These comparisons based on the nonuniversal automata are quite consistent to those based all the 11,000 automata. Additional comparisons of the four improved constructions based on $\mathbb{A}_{10}$ and $\mathbb{A}_{20}$ can be found in the appendix.

Constructions	Effective Samples	S_R	(Win)	S_L	(Win)	S_L/S_R
$\mathbb{A}_{15}$						
Ramsey+Am	1,270	1,310.06	(0.0)	957.96	(0.50)	0.731
SP+ASE		39.90	(1191.5)	19.21	(780.17)	0.482
Rank+A		314.22	(20.0)	40.64	(243.17)	0.129
Slice+ADRM		186.90	(58.5)	34.04	(246.17)	0.182
$\mathbb{A}_{15}$						
Ramsey+PAm	1,495	1,125.22	(1.0)	843.07	(0.5)	0.749
SP+PASE		38.71	(1110.5)	19.39	(668.0)	0.501
Rank+PA		260.28	(177.0)	37.39	(446.0)	0.144
Slice+PADRM		163.29	(206.5)	31.47	(380.5)	0.193

Table 5.5: A comparison of the four improved complementation constructions based on the nonuniversal automata in $\mathbb{A}_{15}$ without and with preminimization

5.6 Complete Experimental Results

This section includes the complete experimental results that we performed to compare the four representative complementation constructions and compare their improved verions. In the following, we first recall the settings of our experiments and then describe the comparisons based on the experimental results.

Let the parameter n be 10, 15, or 20, which denotes a size of states. For each n, we randomly generated 11,000 automata with an alphabet of size 2 and states of size n as a test set. Among the 11,000 automata of state size n, denoted by $\mathbb{A}_n$, 100 automata are generated from each combination of 11 transition densities (from 1.0 to 3.0) and 10 acceptance densities (from 0.1 to 1.0). For every generated automaton $A = (\Sigma, Q, q_0, \delta, \mathcal{F})$ with a given state size n, symbol $a \in \Sigma$, transition density r, and acceptance density f, we made $q \in \delta(p, a)$ for $\lceil rn \rceil$ pairs of states $(p, q) \in Q^2$ uniformly chosen at random and added $\lceil fn \rceil$ states to $\mathcal{F}$ uniformly at random. Our parameters were chosen to generate a large set of complementation problems, ranging from easy to hard.

We chose four representative complementation constructions, namely `Ramsey` [83], `SP` [74], `Rank` [79], and `Slice` [101], each of which is considered the most efficient construction in its respective approach. These constructions were implemented in the GOAL tool [94]. The experiment was performed on a cluster at Rice University (http://rcsg.rice.edu/sugar/int/) with GOAL based on the three generated test sets $\mathbb{A}_{10}$, $\mathbb{A}_{15}$, and $\mathbb{A}_{20}$. For each complementation task, we allocated one 2.83-GHz CPU and 1 GB of memory. The timeout of a complementation task was 10 minutes.

Constructions	T	M	Effective Samples	S_R	(Win)	S_L	(Win)	S_L/S_R
$\mathbb{A}_{10}$								
`Ramsey`	2,944	81	5,056	406.55	(70.5)	50.06	(1085.75)	0.123
`SP`	0	0		39.05	(3834.0)	3.50	(1312.92)	0.090
`Rank`	2,667	0		462.04	(1150.5)	7.63	(1520.92)	0.017
`Slice`	1,191	482		913.57	(1.0)	4.92	(1136.42)	0.005
$\mathbb{A}_{15}$								
`Ramsey`	4,564	36	2,259	513.85	(0)	30.82	(522.50)	0.060
`SP`	5	0		45.26	(1,843)	2.27	(556.67)	0.050
`Rank`	5,303	0		260.41	(415)	2.79	(649.17)	0.011
`Slice`	3,131	3,213		790.92	(1)	3.03	(530.67)	0.004
$\mathbb{A}_{20}$								
`Ramsey`	5,588	240	1,390	549.24	(0)	17.38	(335.5)	0.032
`SP`	53	0		57.41	(1,101)	1.88	(348.5)	0.033
`Rank`	6,784	0		290.17	(289)	2.17	(370.5)	0.007
`Slice`	3,647	4,224		736.94	(0)	2.42	(335.5)	0.003

Table 5.6: A comparison of the four representative complementation constructions based on $\mathbb{A}_{10}$, $\mathbb{A}_{15}$, and $\mathbb{A}_{20}$

5.6.1 Comparisons of Basic Constructions

The comparisons of the four representative constructions based on the three test sets are summarized in Table 5.6 where T is the total number of timed-out tasks and M is the total number of tasks that run out of memory. The column S_R is the average number of reachable states, while S_L is the average number of live states, of the complements. The column Effective Samples denotes the total number of effective samples where both S_R and S_L are calculated. There are 5056, 2259, and 1390 effective samples respectively in $\mathbb{A}_{10}$, $\mathbb{A}_{15}$, and $\mathbb{A}_{20}$. Among the effective samples of each test set, around 90% of the automata are universal. The Win column of a construction in S_R (resp., S_L) denotes the fractional share of effective samples where the construction wins w.r.t. S_R (resp., S_L). More detailed definition of effective samples and win shares can be found in Section 5.1.

The number of unfinished tasks by a construction is the sum of T and M. Compared to `SP` that has less than 0.5% of the unfinished tasks in all the test sets, each

of `Ramsey`, `Rank`, and `Slice` has more than 15% in $\mathbb{A}_{10}$, more than 40% in $\mathbb{A}_{15}$, and more than 50% in $\mathbb{A}_{20}$.

The columns S_R and S_L show that `SP` is the best in average state size. The low S_L/S_R ratio shows that `Rank` and `Slice` create more dead states that can be easily pruned off. Although `Rank` generates more dead states than `SP`, the Win column of S_L shows that `Rank` produces more complements that are the smallest after pruning dead states.

`Rank` and `Slice` become much closer to `SP` in S_L because around 90% of the effective samples are universal automata, whose complements have no live states. If we only consider nonuniversal automata, the gaps between `SP` and `Rank`, and `SP` and `Slice` in S_L become larger as shown in Table 5.7. This case also happens in the the Win column of S_L between `SP` and `Ramsey`, and `SP` and `Slice`.

Constructions	Effective Samples	S_R	(Win)	S_L	(Win)	S_L/S_R
$\mathbb{A}_{10}$						
`Ramsey`	713	1,600.64	(0)	348.88	(0)	0.644
`SP`		47.23	(493.5)	18.74	(227.17)	0.397
`Rank`		437.58	(218.5)	48.03	(435.17)	0.110
`Slice`		686.04	(1)	28.77	(50.67)	0.042
$\mathbb{A}_{15}$						
`Ramsey`	171	1,892.37	(0)	397.10	(0)	0.210
`SP`		36.38	(102.5)	17.77	(34.67)	0.488
`Rank`		156.63	(67.5)	24.61	(127.67)	0.157
`Slice`		422.88	(1)	27.75	(8.67)	0.066
$\mathbb{A}_{20}$						
`Ramsey`	48	2,052.02	(0)	475.27	(0)	0.232
`SP`		41.13	(30)	26.58	(13)	0.646
`Rank`		165.88	(18)	34.75	(35)	0.209
`Slice`		206.90	(0)	42.02	(0)	0.203

Table 5.7: A comparison of the four representative complementation constructions based on the nonuniversal automata in $\mathbb{A}_{10}$, $\mathbb{A}_{15}$, and $\mathbb{A}_{20}$

In a summary, the experimental results show that (1) `SP` is the best in average state size and in the number of finished tasks, (2) `Ramsey` is not competitive

in complementation even though it is competitive in universality and containment testing as shown in [3, 25, 26], and (3) except in $\mathbb{A}_{10}$, `Slice` has the most unfinished tasks (even more than `Ramsey`) and, compared to `SP` and `Rank`, produces many more states.

5.6.2 Comparisons of Improved Constructions

We also compared the four constructions with one optimization heuristic for `Ramsey` in Section 5.5 and all optimization heuristics in Sections 5.2, 5.3, and 5.4 based on 7963, 4851, and 2951 effective samples respectively in $\mathbb{A}_{10}$, $\mathbb{A}_{15}$, and $\mathbb{A}_{20}$. Recall that the following heuristics were proposed in this paper: simplifying DPWs by simulation (`+S`) and merging equivalent states (`+E`) for `SP`; maximizing the Büchi acceptance set (`+A`) for `Rank`; deterministic decoration (`+D`), reducing transitions (`+R`), and merging adjacent nodes (`+M`) for `Slice`. The heuristic for `Ramsey` is the simplification of intermediate classic finite automata on finite words (`+m`). The comparisons are summarized in Table 5.8, which shows that `SP+ASE` still outperforms the other three in the average state size and in running time. Table 5.8 also shows the following changes made by our heuristics in the comparison: (1) `SP+ASE` outperforms `Rank+A` in the number of smallest complements after pruning dead states, and (2) `Slice+ADRM` creates fewer reachable states than `Rank+A` in average, and finishes more tasks than `Rank+A` and `Ramsey+Am`.

As the heuristic of preminimization applied to the input automata, denoted by `+P`, is considered to help the nondeterministic constructions more than the deterministic one, we also compare the four improved constructions with preminimization and summarize the results in Table 5.9. We only applied the preminimization implemented in the GOAL tool, namely the simplification by simulation in [86]. According

Constructions	T	M	Effective Samples	S_R	(Win)	S_L	(Win)	S_L/S_R
$\mathbb{A}_{10}$								
`Ramsey+Am`	924	3	7,963	461.56	(9.50)	257.21	(1,323.75)	0.56
`SP+ASE`	0	0		26.96	(7,701.83)	7.42	(3,062.75)	0.28
`Rank+A`	2,285	0		438.66	(71.83)	23.28	(1,818.75)	0.05
`Slice+ADRM`	2	0		115.79	(179.83)	12.76	(1,757.75)	0.11
$\mathbb{A}_{15}$								
`Ramsey+Am`	3,119	2	4,851	666.86	(0.0)	251.53	(895.75)	0.38
`SP+ASE`	13	7		23.88	(4,772.5)	5.77	(1,675.42)	0.24
`Rank+A`	3,927	0		384.35	(20.0)	11.38	(1,138.42)	0.03
`Slice+ADRM`	228	0		185.02	(58.5)	9.65	(1,141.42)	0.05
$\mathbb{A}_{20}$								
`Ramsey+Am`	5,009	14	2,951	618.07	(0.00)	125.76	(618.75)	0.20
`SP+ASE`	83	133		18.81	(2,929.67)	3.81	(894.75)	0.20
`Rank+A`	4,955	0		427.75	(5.17)	8.41	(717.25)	0.02
`Slice+ADRM`	1,220	0		213.76	(16.17)	5.96	(720.25)	0.03

Table 5.8: A comparison of the four improved complementation constructions based on $\mathbb{A}_{10}$, $\mathbb{A}_{15}$, and $\mathbb{A}_{20}$

to our experimental results, the preminimization does improve `Ramsey`, `Rank`, and `Slice` more than `SP` in the complementation but does not close too much the gap between them in the comparison, though there are other preminimization techniques that we did not implement and apply in the experiment.

The comparisons of the four improved constructions based on the nonuniversal automata without and with preminimization are summarized respectively in Table 5.10 and in Table 5.11. These comparisons based on the nonuniversal automata are quite consistent to those based all the automata.

5.7 Summary of this Chapter

We examined the performance of the four complementation approaches by experiments with three test sets of 11,000 automata. The experimental results showed that the determinization-based approach performs better than the other three in most cases in terms of time and size of states. This is surprising and goes against the conventional wisdom that the nondeterministic approaches are better. The Ramsey-

Constructions	T	M	Effective Samples	S_R	(Win)	S_L	(Win)	S_L/S_R
$\mathbb{A}_{10}$								
Ramsey+PAm	813	6	8,565	355.43	(53.0)	220.28	(1,453.25)	0.62
SP+PASE	0	0		22.25	(6,032.0)	6.97	(2,794.25)	0.31
Rank+PA	1,765	0		306.92	(1,209.5)	20.54	(2,208.25)	0.07
Slice+PADRM	2	0		89.70	(1,270.5)	11.40	(2,109.25)	0.13
$\mathbb{A}_{15}$								
Ramsey+PAm	2,825	6	5,618	479.99	(17.50)	225.08	(1,031.25)	0.47
SP+PASE	12	7		19.47	(3,820.67)	5.89	(1,698.75)	0.30
Rank+PA	3,383	0		232.63	(875.67)	10.68	(1,476.75)	0.05
Slice+PADRM	216	0		135.74	(904.17)	9.12	(1,411.25)	0.07
$\mathbb{A}_{20}$								
Ramsey+PAm	4,647	15	3,741	390.30	(16.5)	159.04	(762.75)	0.41
SP+PASE	102	110		13.51	(2,335.0)	4.49	(1,059.42)	0.33
Rank+PA	4,422	0		208.18	(685.0)	7.76	(964.92)	0.04
Slice+PADRM	1,180	0		133.69	(704.5)	6.66	(953.92)	0.05

Table 5.9: A comparison of the four improved complementation constructions based on $\mathbb{A}_{10}$, $\mathbb{A}_{15}$, and $\mathbb{A}_{20}$ with preminimization

based approach is not competitive in complementation though it is competitive in universality and containment testing.

We also proposed various optimization heuristics for three of the approaches and performed an experiment with one of the test sets to show the improvement. The experimental results also showed that our heuristics substantially improve `SP` and `Slice` in creating far fewer states. `Rank` and especially `Slice` can finish more complementation tasks with our heuristics.

Constructions	Effective Samples	S_R	(Win)	S_L	(Win)	S_L/S_R
$\mathbb{A}_{10}$						
Ramsey+Am	2,672	944.62	(1)	764.56	(1)	0.809
SP+ASE		41.65	(2,428.17)	20.14	(1,740)	0.484
Rank+A		356.82	(67.67)	67.39	(496)	0.189
Slice+ADRM		116.78	(175.17)	36.05	(435)	0.309
$\mathbb{A}_{15}$						
Ramsey+Am	1,270	1,310.06	(0.0)	957.96	(0.50)	0.731
SP+ASE		39.90	(1191.5)	19.21	(780.17)	0.482
Rank+A		314.22	(20.0)	40.64	(243.17)	0.129
Slice+ADRM		186.90	(58.5)	34.04	(246.17)	0.182
$\mathbb{A}_{20}$						
Ramsey+Am	478	1,117.59	(0.00)	772.57	(0)	0.691
SP+ASE		35.62	(456.67)	18.35	(277)	0.515
Rank+A		350.93	(5.17)	46.74	(99)	0.133
Slice+ADRM		219.55	(16.17)	31.59	(102)	0.144

Table 5.10: A comparison of the four improved complementation constructions based on the nonuniversal automata in $\mathbb{A}_{10}$, $\mathbb{A}_{15}$, and $\mathbb{A}_{20}$

Constructions	Effective Samples	S_R	(Win)	S_L	(Win)	S_L/S_R
$\mathbb{A}_{10}$						
Ramsey+PAm	2,752	845.03	(1.5)	682.96	(0.5)	0.808
SP+PASE		39.54	(2,096.5)	19.57	(1,341.5)	0.495
Rank+PA		305.25	(294.5)	61.78	(755.5)	0.202
Slice+PADRM		107.76	(361.5)	33.35	(656.5)	0.309
$\mathbb{A}_{15}$						
Ramsey+PAm	1,495	1,125.22	(1.0)	843.07	(0.5)	0.749
SP+PASE		38.71	(1110.5)	19.39	(668.0)	0.501
Rank+PA		260.28	(177.0)	37.39	(446.0)	0.144
Slice+PADRM		163.29	(206.5)	31.47	(380.5)	0.193
$\mathbb{A}_{20}$						
Ramsey+PAm	692	1,084.23	(0.00)	856.38	(0.00)	0.790
SP+PASE		33.57	(478.33)	19.86	(297.67)	0.592
Rank+PA		224.91	(97.33)	37.55	(202.67)	0.167
Slice+PADRM		165.83	(116.33)	31.61	(191.67)	0.191

Table 5.11: A comparison of the four improved complementation constructions based on the nonuniversal automata in $\mathbb{A}_{10}$, $\mathbb{A}_{15}$, and $\mathbb{A}_{20}$ with preminimization

Chapter 6

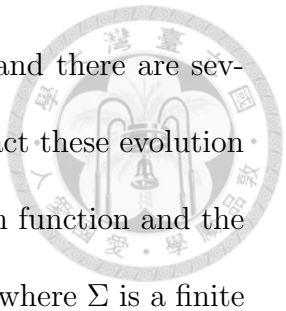
Containment Testing Algorithms

In this chapter, we propose an incremental containment testing approach based on the determinization-based complementation constructions, namely Safra's construction [77], Muller-Schupp construction [70, 4], and Safra-Piterman construction [74]. We will focus on how to apply Safra-Piterman construction in incremental containment testing. The other two approaches can be applied in the same way. In the following, we first briefly describe Safra-Piterman construction, then present how to apply the construction in incremental universality testing (a special case of containment testing), and finally describe the extension to incremental containment testing.

6.1 Safra-Piterman Construction

Safra-Piterman construction performs a subset construction with a tree structure, called *compact Safra trees*, to capture all runs of a Büchi automaton on a word. Each node in a compact Safra tree is labeled by a set of states of the automaton. A compact Safra tree satisfies the following two properties:

- If node n is a child of node m, then $n \subset m$.
- If node n is a sibling of m, then $n \cap m = \emptyset$.



Although the evolution of compact Safra trees are complicated and there are several annotations associated to compact Safra trees, we can abstract these evolution process and annotations into two functions, namely the transition function and the coloring function. Let $A = (\Sigma, Q, q_0, \delta, F)$ be a Büchi automaton where Σ is a finite alphabet, Q a finite set of states, q_0 the initial state, $\delta : Q \times \Sigma \to 2^Q$ a transition function, and $F \subset Q$ a Büchi acceptance condition. Let T^A denote the set of compact Safra trees over the states of A. The transition function is defined as $\delta_T^A : T \times \Sigma \to T^A$ while the coloring function is defined as $\rho^A : T^A \times \mathbb{N}$. We may omit the superscript A if it is clear in the context. Based on Safra-Piterman construction in [74], we have the following lemma.

Lemma 6.1.1. *Let $A = (\Sigma, Q, q_0, \delta, F)$ be a Büchi automaton and $P = (\Sigma, S, s_0, \Delta, G)$ a deterministic parity automaton defined as the following:*

- $S = T$.
- s_0 *is a tree with the only root node* $\{q_0\}$.
- $\Delta = \delta_T$.
- $G = \rho$.

Then, $L(P) = L(A)$.

Although the conventional wisdom is that the determinization-based complementation constructions have to be performed stage by stage with deterministic automata as the intermediate results, we point out that these constructions in fact can be performed incrementally. We first show how to perform the determinization constructions incrementally. Let f be a function, we define the update of f as

$f[k \Rightarrow v]$:

$$f[k \Rightarrow v](x) = \begin{cases} \text{v} & \text{if } x = k \\ \text{f(x)} & \text{if } x \neq k \end{cases}$$

The incremental Safra-Piterman determinization construction is shown in Algorithm 7.

Algorithm 7 SPDeterminization(A)

```
1:  s_0 := a tree with the only root node {q_0}
2:  S := {}
3:  U := {s_0}
4:  while U ≠ ∅ do
5:      Remove an element s from U
6:      for a ∈ Σ do
7:          t := δ_T(s, a)
8:          if t ∉ S then
9:              G := G[t ⇒ ρ(t)]
10:             S := S ∪ {t}
11:             U := U ∪ {t}
12:         Δ = Δ[(s, a) ⇒ t]
```

6.2 Incremental Universality Testing

Given a Büchi automaton $A = (\Sigma, Q, q_0, \delta, F)$, the universality testing checks if $L(A) = \Sigma^\omega$, or in other words, $L(\overline{A}) = \emptyset$. To perform the universality testing incrementally based on Safra-Piterman construction, we have to construct complement automata on the fly. The incremental Safra-Piterman determinization construction has been shown in the previous section. The rest is to show how to construct a complement deterministic parity automaton and convert it to an equivalent Büchi automaton incrementally.

To construct a complement deterministic parity automaton incrementally, we simply follow the incremental Safra-Piterman construction but increment the color of every state by 1. This is achieved by defining another function $\rho' : T \times \mathbb{N}$ such that $\rho'(t) = \rho(t) + 1$.

Lemma 6.2.1. *Let* $A = (\Sigma, Q, q_0, \delta, F)$ *and* $B = (\Sigma, Q, q_0, \delta, G)$ *be deterministic parity automata with* $G(q) = F(q) + 1$. *Then,* $L(A) = \overline{L(B)}$.

Proof. Let $r = q_0 q_1 q_2 \ldots$ be an accepting run of A on a word $w = w_0 w_1 w_2 \ldots$. Since r is an accepting run of A, $min\{F(q) \mid q \in inf(r)\}$ is even. By the definition of G, $min\{G(q) \mid q \in inf(r)\}$ is odd. Thus, r is a rejecting run and w is rejected by B.

Let $r = q_0 q_1 q_2 \ldots$ be an accepting run of B on a word $w = w_0 w_1 w_2 \ldots$. Since r is an accepting run of B, $min\{F(q) \mid q \in inf(r)\}$ is even. By the definition of G, $min\{F(q) \mid q \in inf(r)\}$ is odd. Thus, r is a rejecting run and w is rejected by A. □

To convert a deterministic parity automaton to a Büchi automaton, we apply the heuristic in Chapter 5 which start the guessing of the minimal even color when the construction sees a state with an even color. Here we restate the typical conversion with such heuristic for deterministic parity automaton as the following lemma.

Lemma 6.2.2. *Given a deterministic parity automaton* $P = (\Sigma, Q, q_0, \delta, F)$ *where* $F : Q \to \{0, 1, \ldots, 2r\}$, *the typical conversion constructs a Büchi automaton* $A = (\Sigma, S, s_0, \Delta, G)$ *where*

- $S = Q \times \{0, 2, \ldots, 2r\}$,
- $s_0 = (q_0, 0)$,
- *For all* $a \in \Sigma$,

$$\Delta((q,i),a) = \begin{cases} \{(\delta(q,a),i)\} & \text{if } i = 0 \text{ and } F(\delta(q,a)) \text{ is odd} \\ \{(\delta(q,a),i),(\delta(q,a),F(\delta(q,a)))\} & \text{if } i = 0 \text{ and } F(\delta(q,a)) \text{ is even} \\ \{(\delta(q,a),i)\} & \text{if } i > 0 \text{ and } F(\delta(q,a)) \geq i \end{cases}$$

- $G = \{(q, 2k) \in S \mid F(q) = 2k\}$.

Then, $L(A) = L(P)$.

As the deterministic parity automaton constructed by Safra-Piterman construction uses compact Safra trees as the states, to construct a complement Büchi automaton of another Büchi automaton, we can respectively replace the states, transition relation, and acceptance condition of P in the previous conversion from parity automata to Büchi automata by the compact Safra trees T, δ_T, and ρ'.

Theorem 6.2.3. *Let $A = (\Sigma, Q, q_0, \delta, F)$ be a Büchi automaton, and $B = (\Sigma, S, s_0, \Delta, G)$ another Büchi automaton constructed as the following:*

- $S = T \times \mathbb{N}$ *where T is the set of compact Safra trees over Q.*
- $s_0 = (t_0, 0)$ *where t_0 is a compact Safra tree with the only root node $\{q_0\}$.*
- *For all $a \in \Sigma$,*

$$\Delta((s,i),a) = \begin{cases} \{(\delta_T(s,a),i)\} & \text{if } i = 0 \text{ and } \rho'(\delta_T(s,a)) \text{ is odd} \\ \{(\delta_T(s,a),i),(\delta_T(s,a),\rho'(s))\} & \text{if } i = 0 \text{ and } \rho'(\delta_T(s,a)) \text{ is even} \\ \{(\delta_T(s,a),i)\} & \text{if } i > 0 \text{ and } \rho'(s) \geq i \end{cases}$$

- $G = \{(s,i) \in S \mid \rho'(s) = i \text{ and } i \text{ is even}\}$.

Then, $L(B) = \overline{L(A)}$

Proof. This theorem is proven with Lemma 6.1.1, Lemma 6.2.1, and Lemma 6.2.2. □

Finally, to test the universality of a Büchi automaton $A = (\Sigma, Q, q_0, \delta, F)$, we perform a double depth-first search in its complement $B = (\Sigma, S, s_0, \Delta, G)$, which is constructed incrementally.

Algorithm 8 SPUniversality(A)

Initialize two empty stacks $S1$ and $S2$
$U1 = U2 = \emptyset$
return SPDFS1$(A, s_0, S1, U1, S2, U2)$

Algorithm 9 SPDFS1$(A, s, S1, U1, S2, U2)$

```
Push s to S1
Add s to U1
for a ∈ Σ do
  for t ∈ Δ(s, a) do
    if t ∉ U1 then
      if SPDFS1(t) = false then
        return false
if s ∈ G then
  if SPDFS2(A, s, S1, U1, S2, U2) = false then
    return false
Pop out s from S1
```

Algorithm 10 SPDFS2$(A, s, S1, U1, S2, U2)$

```
Push s to S2
Add s to U2
for a ∈ Σ do
  for t ∈ Δ(s, a) do
    if t ∈ S1 then
      return false
    else if t ∉ U2 then
      if SPDFS2(A, t, S1, U1, S2, U2) = false then
        return false
Pop out s from S2
```

6.3 Incremental Containment Testing

Given two Büchi automata $A_1 = (\Sigma, Q_1, q_{10}, \delta_1, F_1)$ and $A_2 = (\Sigma, Q_2, q_{20}, \delta_2, F_2)$, the containment testing checks if $L(A_1) \subseteq L(A_2)$, or in other words, $L(A_1 \times \overline{A_2}) = \emptyset$ where $A_1 \times \overline{A_2}$ is the intersection of A_1 and the complement of A_2. Similar to universality testing in the previous section, to perform containment testing incrementally, we can perform a double-depth first search in the intersection, which is constructed on the fly.

Theorem 6.3.1. *Let $A_1 = (\Sigma, Q_1, q_{10}, \delta_1, F_1)$ and $A_2 = (\Sigma, Q_2, q_{20}, \delta_2, F_2)$ be two Büchi automata. Let $\overline{A_2} = (\Sigma, S, s_0, \Delta, G)$ be the complement of A_2. We can construct a Büchi automaton $B = (\Sigma, Q, q_0, \delta, F)$ as the following:*

- $Q = Q_1 \times S \times \{0, 1, 2\}$.
- $q_0 = (q_{10}, s_0, 0)$.
- *For all* $a \in \Sigma$, $\delta((q, s, i), a) = \{(q', s', i') \mid q' \in \delta_1(q, a), s' \in \Delta(s, a),$ *and* $i' = \sigma(q, s, i)\}$ *where*

$$\sigma(q, s, i) = \begin{cases} 1 & \text{if } i = 0, \text{ or } i = 1 \text{ and } q \notin F_1 \\ 2 & \text{if } i = 1 \text{ and } q \in F_1, \text{ or } i = 2 \text{ and } s \notin G \\ 0 & \text{if } i = 2 \text{ and } s \in G \end{cases}$$

- $F = Q_1 \times S \times \{0\}$

Then, $L(B) = L(A_1 \times \overline{A_2})$.

Proof. The construction of B basically follows the standard intersection of two Büchi automaton, of which the first automaton if A_1 and the second automaton is the complement of A_2 which is constructed incrementally with B. □

6.4 Experimental Results

We have implemented our determinization-based incremental containment testing algorithms in the GOAL tool. Simulation relation between two automata in a containment testing is computed same as in [1] We performed an experiment to compare our incremental containment testing algorithm based on Safra-Piterman construction with the Ramsey-based approach in [1, 2] and the naive approach that takes a full complement, then intersection, and finally an emptiness test. In the naive approach, Safra-Piterman construction with optimization heuristics proposed in Section 5.2 is used to take complements. In the experiment, we randomly generated 100 pairs of automata for each combination of state size 5, 10, or 15, transition density of 1.2 or 1.4, and acceptance density of 0.2. Each automaton has an alphabet of size 2 and does not have any unreachable or dead states. There are totally 600

State Size	Safra-Piterman			Naive			Ramsey		
	Time	Timeout	OOM	Time	Timeout	OOM	Time	Timeout	OOM
All cases									
5	28,411	0	0	673,601	1	1	19,493	0	0
10	313,405	0	0	434,165	0	0	254,946	1	0
15	1,204,464	3	0	4,118,098	0	5	1,886,536	3	0
Total	1,546,280	3	0	5,225,864	1	6	2,160,975	4	0
Cases where containment holds									
5	9,728	0	0	6,753	0	0	13,932	0	0
10	16,426	0	0	9,330	0	0	241,962	1	0
15	51,797	0	0	20,570	0	0	1,861,974	2	0
Total	77,951	0	0	36,653	0	0	2,117,868	3	0
Cases where containment does not hold									
5	18,683	0	0	666,848	1	1	5,561	0	0
10	296,979	0	0	424,835	0	0	12,984	0	0
15	1,152,667	3	0	4,097,528	0	5	24,562	1	0
Total	1,468,329	3	0	5,189,211	1	6	43,107	1	0

Table 6.1: Comparing three containment testing approaches in terms of running time based on 600 pairs of automata. The time unit is millisecond. OOM means that the approach runs out of memory.

pairs of automata generated, of which 39 pairs pass the containment testing. For each containment task, we allocated one 2.33-GHz CPU and 1 GB of memory. The timeout of a containment task was 10 minutes.

The experimental results are summarized in Table 6.1. There are total 3 unfinished tasks for the Safra-Piterman approach, 7 for the naive approach, and 4 for the Ramsey-based approach. Consider the cases where the containment holds. Both the Safra-Piterman approach and the naive approach are much better than the Ramsey-based approach. The Safra-Piterman approach performs worse than the naive approach in those cases because it has to visit all states in the product that are constructed incrementally without simplification. Consider all other cases where the containment does not hold. The Ramsey-based approach performs much better than the Safra-Piterman approach and the naive approach. A possible reason is that the Ramsey-based approach uses breath-first search while the Safra-Piterman approach uses nested depth-first search. Thus, the Ramsey-based approach could

find counterexamples faster if the counterexamples are shallow.

6.5 Summary of this Chapter

We have presented an approach for incremental containment testing based on the determinization-based constructions. Although the conventional wisdom is that the determinization-based complementation constructions have to be performed stage by stage with deterministic automata as the intermediate results, we point out that these constructions in fact can be performed incrementally. The experimental results show that our approach performs much better than the Ramsey-based approach when the containment holds but worse than the Ramsey-based approach when the containment does not hold.

Chapter 7

Tool Support

As mentioned in the introduction, there are tools for formulae translation and model checking but none of these tools provide facilities for visually manipulating automata and the temporal logics. A tool with these facilities would allow the user to learn complex automata operations, formula translations, and relationship between automata and temporal formulae. If the functions of such a tool can be accessed easily by external tools, it can also help researchers develop new algorithms and compare the new ones with the implemented algorithms in it. The GOAL tool that we built is one such tool that can help education and research on automata and temporal logics. With the help of GOAL, we already performed several comparative experiments in Chapters 4, 5, and 6. GOAL also has been used by other researchers in their work [11, 12, 56].

We also built another tool called Büchi Store, which is a Web-based repository of ω-automata and linear temporal formulae. In Büchi Store, the user can browse the repository, search for automata that are equivalent to a specified QPTL formula, translate a QPTL formula online, upload automata with equivalent formula provided and checked for correctness. Most functions of Büchi Store require operations on automata and temporal formulae provided by GOAL. Additional properties such as

classification into the Manna-Pnueli temporal hierarchy [65] are also provided.

In the following, we will first describe the GOAL tool and then Büchi Store.

7.1 GOAL

GOAL[1] is a graphical interactive tool for defining and manipulating games, ω-automata, and logic formulae. The acronym GOAL was originally derived from "**G**raphical Tool for **O**mega-**A**utomata and **L**ogics". It also stands for "**G**ames, **O**mega-**A**utomata, and **L**ogics". To the best of our knowledge, GOAL is the first graphical interactive tool designed for education and research on ω-automata and temporal logics.

The first generation of GOAL was formally introduced in [95] and later extended in [94]. It is implemented in Java and built upon the classic finite automata and graphical modules of JFLAP [76]. Later in [92], we introduced the second generation, which is a complete redesign with an extensible architecture, many enhancements to existing functions, and new features. The major algorithms implemented in GOAL are summarized in Table 7.1. In the following, we will describe the main features of GOAL, some use cases, and the implementation details.

7.1.1 Main Functions

In this section, we describe the main functions of GOAL along with some highlights of their implementation.

Graphical Manipulation of Games and Automata

The user can easily point-and-click and drag-and-drop to create a game, an ω-automaton, an alternating automaton, or a two-way alternating automaton; see

[1]The tool is available at http://goal.im.ntu.edu.tw/.

Translation of QPTL Formulae
`Tableau` *, `IncTableau` * [49], `Tester` * [50], `GPVW` * [34], `GPVW+` * [34], `LTL2AUT` * [34], `LTL2AUT+` *, `LTL2BA` * [32], `PLTL2BA` * [33], `MoDeLLa` [81], `Couvreur` [16], `LTL2BUCHI` [35], `KP02` [51], CCJ09 [13]
Complementation of Büchi Automata
Safra* [77], WAPA* [89], WAA* [57], Safra-Piterman* [74], Kurshan's [59], Ramsey-based [8, 83], Muller-Schupp [70, 4], Rank-based [57, 79], Slice-based [47]
Simplification of Automata
Direct and Reverse Simulation* [86], Pruning Fair Sets* [86], Delayed Simulation [24], Fair Simulation [38], Parity Simplification [9]
Parity Game Solving
Recursive [60], McNaughton-Zielonka [68, 104], Dominion Decomposition [46], Small Progress Measure [45], Big Steps [78], Global Optimization [28]

Table 7.1: Major algorithms in GOAL. The Modified Safra algorithm for complementation is a slight variation of Safra's construction.

Figure 7.1. After a game or an automaton is created, the user can lay it out automatically by various layout algorithms implemented in GOAL. If the automatic layout is not satisfactory, the user can manually arrange the states of the automaton in a better shape with the help of snap to grid and display of gridlines provided by GOAL.

Translation of Logic Formulae

Fourteen algorithms have been implemented for QPTL formula to Büchi automaton translation; see Table 7.1. It is also possible to translate a formula into a generalized Büchi automaton, instead of going all the way to a Büchi automaton. Five (Tableau, Incremental Tableau, Temporal Tester, KP02, and PLTL2BA) of them originally supported past operators. We have extended six more (GPVW, LTL2AUT, LTL2AUT+, MoDeLLa, Couvreur's , and LTL2BUCHI) to allow past operators. Except KP02 that originally supports full QPTL formulae, all the other thirteen algorithms are further extended to support QPTL formulae convertible to prenex normal form. It has been shown that QPTL in prenex normal form is

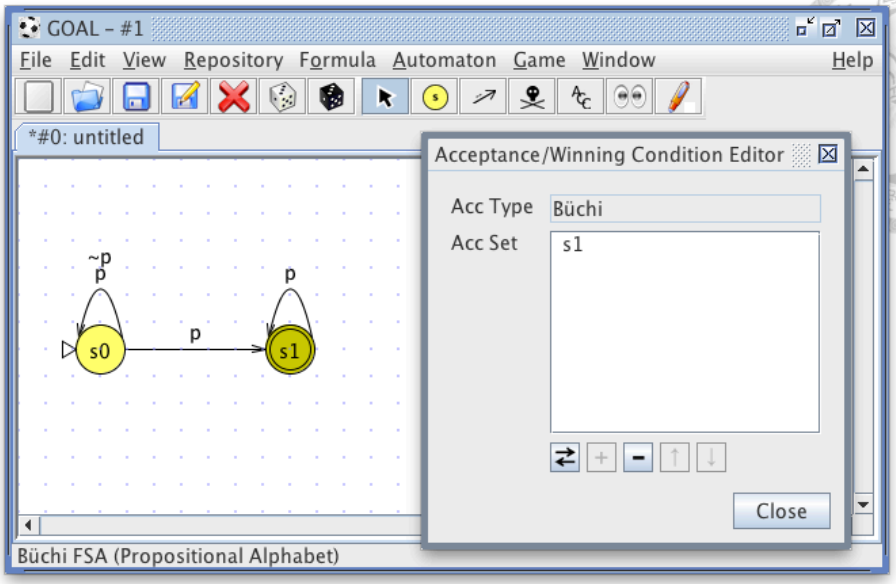


Figure 7.1: An example of drawing a Büchi automaton with GOAL. The Büchi automaton is intended for $\Diamond\Box p$. The inset window is the dialog window for updating the accepting states.

as expressive as QPTL [83]. The supported boolean and temporal operators and their input formats are shown in Table 7.2. To help learning, translations by six of the fourteen algorithms (Tableau, GPVW, GPVW+, LTL2AUT, LTL2AUT+, and MoDeLLa) can be viewed step-by-step. For instance, in the Tableau algorithm the user gets to see the closure computed for the given temporal formula, atoms created as states of the automaton, and transitions between states added one by one. Text is displayed to explain the intermediate step being carried out. The user can "play" the translation, "pause" it, and then "resume" it. A snapshot of translating $\Box\Diamond p$ with the Tableau algorithm is shown in Figure 7.3.

Besides QPTL, GOAL also supports the translation of an ACTL formula [72] to a maximal model (represented as an automaton) of the formula [37, 56]. Such model can be used in model checking or synthesis. The translation of ω-regular expressions is also available in GOAL.

Operator	$\exists$	$\forall$	$\neg$	$\vee$	$\wedge$	$\rightarrow$	$\leftrightarrow$
Format 1	E	A	~	\/	/\	-->	<-->
Format 2	E	A	!	|	&	->	<->

Operator	$\bigcirc$	$\Box$	$\Diamond$	$\mathcal{U}$	$\mathcal{W}$	$\mathcal{R}$	$\ominus$	$\circledcirc$	$\boxminus$	$\diamondminus$	$\mathcal{S}$	$\mathcal{B}$	$\mathcal{T}$
Format 1	()	[]	<>	U	W	R	(-)	(~)	[-]	<->	S	B	T
Format 2	X	G	F	U	W	R	Y	Z	H	O	S	B	T

Table 7.2: Boolean and temporal operators supported in GOAL and their input formats.

Conversion between Automata

GOAL supports many conversions between automata, especially the conversion from nondeterministic Büchi automata, if possible, to deterministic automata with Büchi [61, 6] or co-Büchi [6] acceptance conditions. Moreover, GOAL can automatically find and apply a sequence of chained conversions to convert an automaton to another type specified by the user. With the chained conversions performed by GOAL automatically, tests of Büchi automata and operations on Büchi automata can be applied to various types of automata convertible to equivalent Büchi automata.

Tests of ω-Automata

Emptiness, (language) containment, (language) equivalence, simulation relations, input, and DBW-recognizable tests are supported. In the emptiness test, if the given ω-automaton is non-empty, GOAL highlights the path that corresponds to an accepted input. The equivalence test on two ω-automata is built on top of the containment test which in turn relies on conversions to Büchi automata, the intersection and complementation operations, and the emptiness test. An equivalence test can also be performed between an ω-automaton and a temporal formula. In the input test, if a word provided by the user is accepted by an automaton, an accepting run

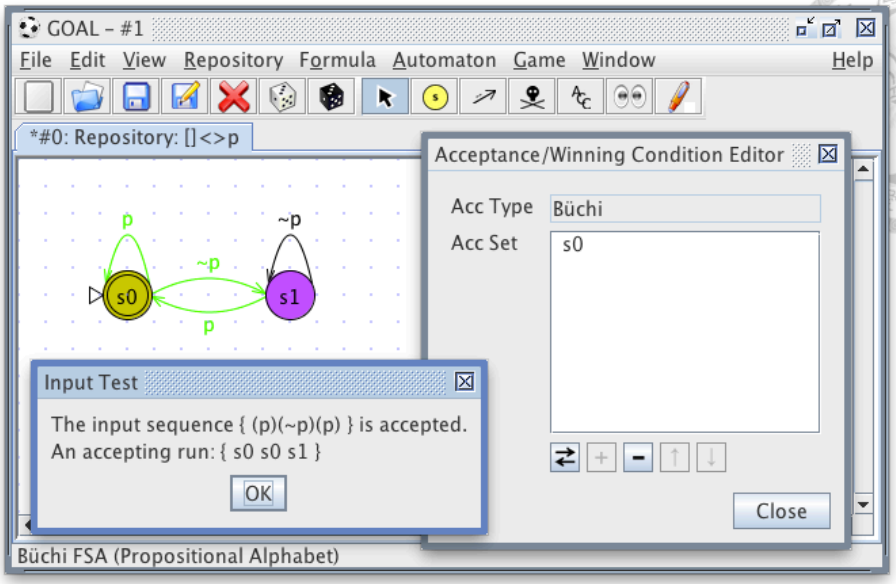


Figure 7.2: An example of testing a Büchi automaton on a word with GOAL. The Büchi automaton is intended for $\Box\Diamond p$. The word $(p\neg pp)^\omega$ is represented in ASCII as `{(p)(~p)(p)}` where `(.)` is used to separate symbols, `!a!a` denotes the negation, and `{.}` denotes the infinite repetition of the enclosed word. The accepting run is highlighted in green.

of the automaton on the word will be highlighted; see Figure 7.2. The user may also run an input test interactively, or construct the run tree or run dag of an automaton on a word step-by-step.

Boolean Operations on ω-Automata

The three standard boolean operations—union, intersection, and complementation are supported. Büchi complementation is crucial in the implementation of language containment and equivalence tests, which are perhaps the most distinct functions of GOAL. Algorithms for Büchi complementation, because of their technical difficulty, are themselves a separate topic of learning (and also of research). Ten algorithms have been implemented in GOAL for Büchi complementation; see Table 7.1 for a listing. Nine of the ten algorithms can be performed step-by-step or stage-by-stage such that the results of intermediate steps or stages can be shown, which will be convenient for learning. Cross-checking greatly increases our confidence in the

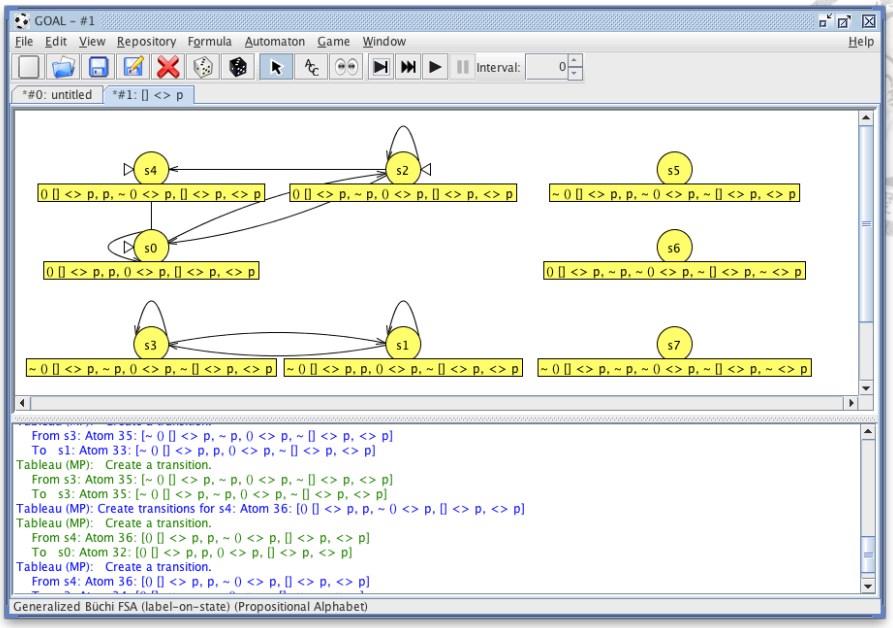


Figure 7.3: A screen shot of the step-by-step translation of a temporal formula into an equivalent generalized Büchi automaton using the Tableau algorithm. The given PTL formula is $\Box\Diamond p$. The lower window displays explanatory descriptions, while the steps are played out in the upper window.

correctness of the different complementation algorithms and hence the correctness of the language containment and equivalence tests.

Classification of ω-Regular Languages

We have implemented an algorithm for testing whether an ω-regular language or Büchi automaton is star-free [19]. If an ω-regular language is star-free, it can be specified by a formula in PTL, which is less expressive than QPTL. We have also implemented the classification of ω-regular languages into the temporal hierarchy of Manna and Pnueli [65]. Such classification can be used not only for educational purposes but also for helping model checking [10].

Tests on QPTL Formulae

Satisfiability and validity tests are supported. The equivalence test between two formulae is not supported directly, but can be easily realized by connecting the two

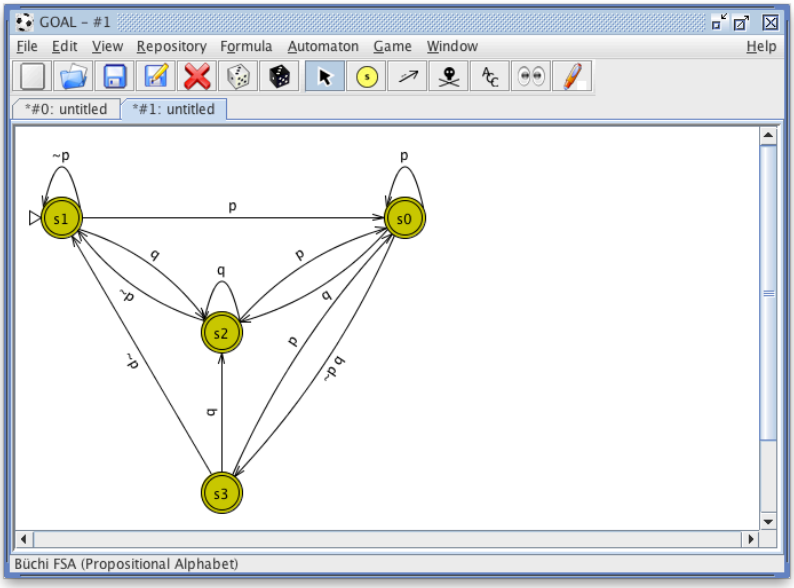
formulae with the mutual implication operator ($\leftrightarrow$) and testing the resulting formula for validity.

Simplifying Büchi Automata

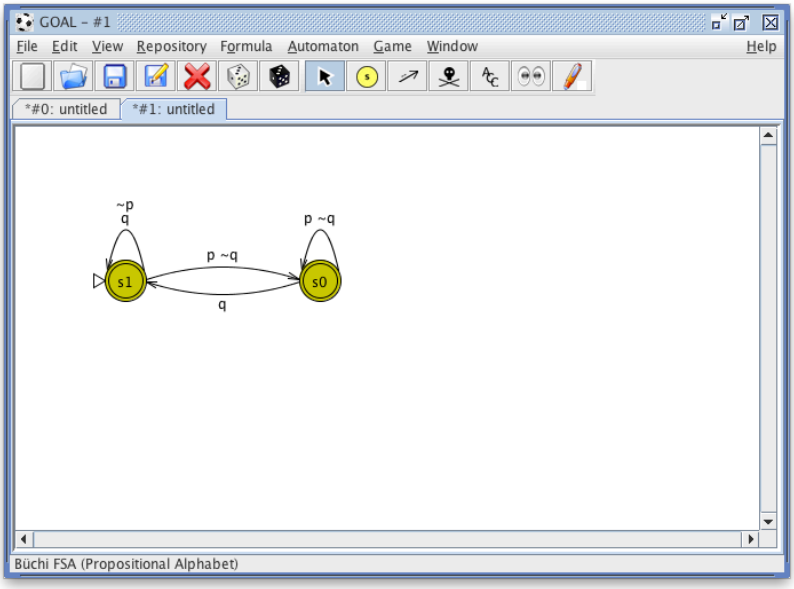
The user can use the simplification (by direct simulation, delayed simulation, or fair simulation) operation to find states of a Büchi automaton that simulate each other and merge those states (if possible); there is also an operation for simplifying generalized Büchi Automata by pruning fair sets (acceptance sets). Four algorithms for finding direct simulation relations are implemented. One algorithm is an adaption of that proposed by Somenzi and Bloem [86] and the other three algorithms are adaptions of those proposed by Henzinger *et al.* [41]. Figure 7.4 shows an example of running the simplification algorithm on an automaton translated from the formula $\Box(p \rightarrow p \,\mathcal{W}\, q)$ (once p becomes true, it will remain true continuously until q becomes true, which may never occur). To understand the original machine-translated automaton is somewhat difficult. After the simplification, one gets a smaller automaton, as shown in Figure 7.4(b), which is easier to understand.

Game Solving and Conversion

We have implemented eight game solving algorithms of which one is for reachability games, one for Büchi games, and six for parity games. Winning regions and strategies in a solved game are highlighted and can be exported with the game to a file. Four of the parity game solving algorithms as well as the Büchi game solving algorithm can be performed step-by-step. Several conversions between games, including the conversion from a Muller game to a parity game [67], are implemented. To help experiments with games, the generation of random games is provided as well. GOAL can also take the product of a game arena (that describes the allowed interactions



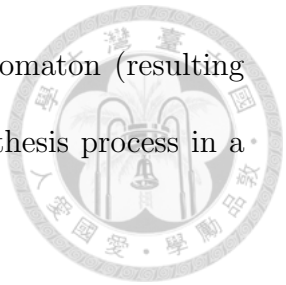
(a) The automaton before simplification



(b) The automaton after simplification

Figure 7.4: A demonstration of the automaton simplification.

between a module and its environment) and a specification automaton (resulting in a game), and hence may be used to experiment with the synthesis process in a game-based approach to the synthesis of reactive modules [84].



Exporting Games and Automata

Once a Büchi automaton has been defined and tested, the user can export it in the Promela (the system modeling language of SPIN) syntax on the screen or as a file. This makes it possible to use GOAL as a graphical specification definition frontend to an automata-theoretic model checker like SPIN. The user may also export a game or an automaton in various image formats such as JPEG, PNG, and SVG, or to external tools such as JGraph[2], LBTT [88], and LaTeX.

The Automata Repository

GOAL comes with a local repository which contains a collection of frequently used temporal formulae and their corresponding equivalent Büchi automata. These automata have been optimized by hand and checked by the GOAL tool itself. The user can also browse automata and temporal formulae in Büchi Store with GOAL. For beginners, this should be very convenient for learning the relation between Büchi automata and linear temporal logic.

The Command-Line Mode

This mode makes most of the GOAL functions accessible by programs or shell scripts and therefore provides an interface between GOAL and external tools. GOAL also provides an interpreter that can execute scripts in a customized language. Thus a batch of GOAL commands can be written as a script and executed by a single GOAL

[2]JGraph is a tool for the visualization of layout of graphs. It is available in http://www.jgraph.com.

process to achieve better performance. Sample scripts that compare translation algorithms and output the results as text files are provided. They can be easily adapted to handle other different tasks.

Utility Functions

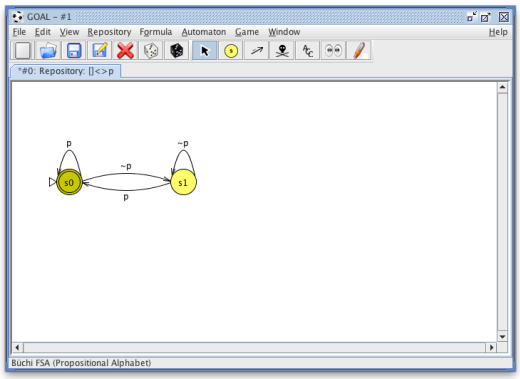
Utility functions are available for collecting statistic data (numbers of states, transitions, and acceptance sets) and for generating random automata, random games, and random temporal formulae. Outputs from external automata tools MoDeLLa [81], LTL2BUCHI [35], and the translation tool in `Spin` may also be converted to the GOAL File Format (GFF, which is an XML file format designed to cover all ω-automata) for further processing by GOAL.

7.1.2 Use Cases

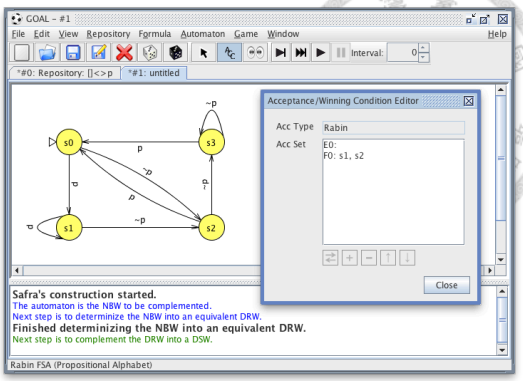
We describe a number of use cases that illustrate how the GOAL functions may be combined and used in particular as a tool for learning/teaching Büchi automata and linear temporal logics or for specification development.

Translating a Temporal Formula into an Equivalent Büchi Automaton

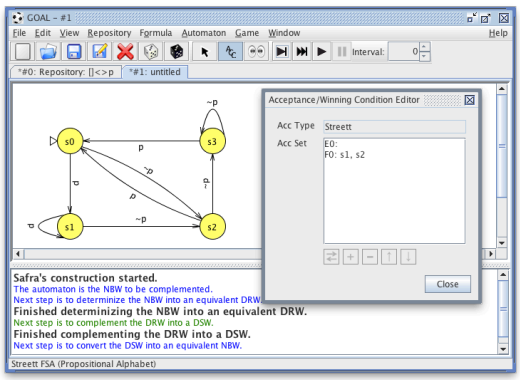
Understanding how a temporal formula can be translated into a Büchi automaton is an essential step in learning automata-theoretic model checking. As we have explained in the introduction, temporal formulae and Büchi automata are very different artifacts and it can be difficult for the student to grasp their correspondence. In the translation function provided by GOAL, the user has an option of viewing the intermediate steps that a translation goes through. The visual aide can be very useful. For example, after studying a translation algorithm, the user can test his understanding of the algorithm by running the algorithm with paper and pencil and



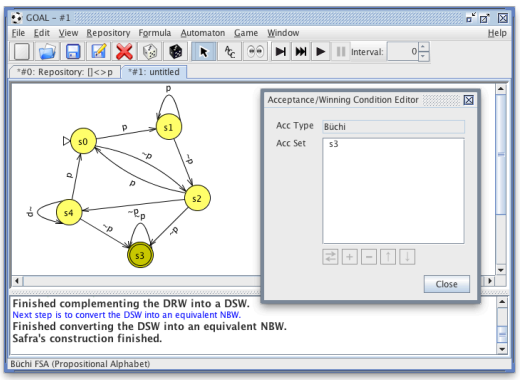
(a) The given automaton (for $\Box\Diamond p$)



(b) Determinization into a Rabin automaton



(c) Conversion into a Streett automaton



(d) Translation back into a Büchi automaton

Figure 7.5: The stages in complementing a Büchi automaton by Safra's construction.

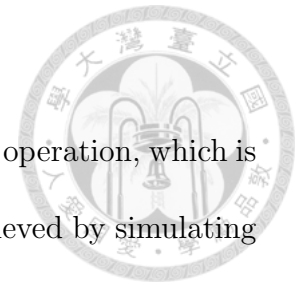
comparing each step with that generated by GOAL.

We suggest that beginners start with the tableau construction of Manna and Pnueli [66]. Though it generates more states than most others do, this algorithm is relatively simple and easy to understand. The steps can be easily divided and their intentions clearly described.

Performing Boolean Operations on Büchi Automata

Büchi automata are closed under boolean operations and these operations can be done algorithmically. To learn any of the boolean operations, the user can perform the operation by hand and then verify correctness by checking the equivalence between the resulting automaton (hand-drawn using the automaton editing function

of GOAL) and the machine-computed one (also by GOAL).



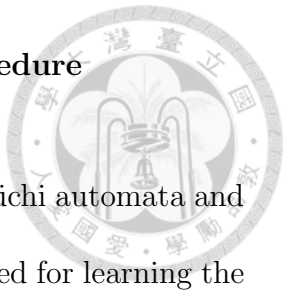
GOAL is particularly useful for learning the complementation operation, which is very complex and difficult to understand. This again can be achieved by simulating an algorithm by hand and checking its correctness by machine. A stage-by-stage complementation with Safra's construction is shown in Figure 7.5.

Checking correctness of a hand-drawn Büchi automaton

Understanding a Büchi automaton is in general harder than understanding an equivalent temporal formula. Consequently, defining or drawing a Büchi automaton that conveys one's intention is also harder than writing a temporal formula for the same purpose. Whether a hand-drawn automaton is correct, i.e., if it conveys the specifier's intention, can be verified using GOAL by following these steps: (1) Write a QPTL (or PTL if it suffices) formula that specifies the same thing. (2) Translate the formula into an equivalent Büchi automaton. (3) Test the equivalence between the machine-translated and the hand-drawn automata. If the equivalence test is positive, then one can be assured that the hand-drawn Büchi automaton is indeed what is intended.

Manual optimization of a specification Büchi automaton

In principle, a smaller specification automaton makes a model checker run faster. GOAL may be used to manually optimize a Büchi automaton by repeatedly merging or removing its states or transitions and checking if the resulting automaton is equivalent to a previous correct automaton. Though this is essentially a trial-and-error process, the equivalence test provided by GOAL will greatly ease the pain.

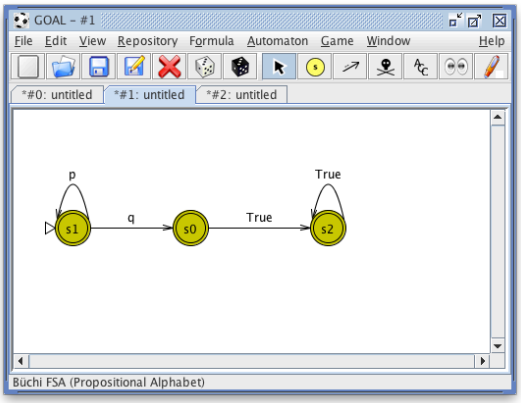


Learning the Automata-Theoretic Model Checking Procedure

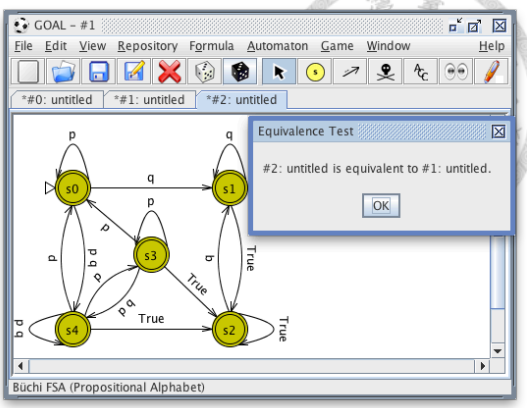
With the ability to translate temporal formulae into equivalent Büchi automata and perform boolean operations on Büchi automata, GOAL can be used for learning the basics of automata-theoretic model checking. It should be a helpful and interesting exercise for the student to go through the typical verification steps: (1) prepare a system Büchi automaton for some small verification problem, e.g., the two-process mutual exclusion problem, (2) write a temporal formula describing the system's safety property (e.g., mutual exclusion) or liveness property (e.g., starvation freedom), (3) negate the formula and translate it into a Büchi automaton, representing all "bad" behaviors, (4) compute the intersection of the given system automaton and the translated negative specification automaton, and (5) check the emptiness of the intersection.

Developing Specification Automata for a Model Checker

In SPIN, the specification can either be given as a PTL formula (without past operators) or directly as a Büchi automaton in Promela code. For a property that is not expressible in PTL, defining a suitable Büchi automaton becomes necessary. In this case, GOAL supplements SPIN by providing a convenient graphical interface for drawing and manipulating Büchi automata. Once the specification automaton has been successfully constructed and checked, it can be exported as Promela code. One can then copy-and-paste the Promela code to SPIN's model file as the "never claim" (a Büchi automaton specifying all behaviors disallowed by the model) and continue the model checking procedure as usual.



(a) The automaton generated from $p \mathcal{W} q$



(b) The automaton generated from $\Box(\diamondsuit\neg p \rightarrow \diamondsuit q)$

Figure 7.6: A safety formula and its equivalent canonical formula. The inset window in Part (b) displays a message from the equivalence test, stating that the two automata are "Equivalent!".

Learning Safety Properties and Safety Formulae

Safety properties are requirements that should be met continuously by the system. A temporal formula is called a *safety* formula (specifying a safety property) if it is equivalent to some formula in the *canonical* form $\Box p$, where p is a past formula (which contains no future operators) [65, 66]. The correspondence between a formula and its equivalent canonical safety formula can be hard to recognize. For example, the formula $p \mathcal{W} q$ (read "p wait-for q", which means p holds until an occurrence of q or p holds forever) is a safety formula, because it is equivalent to the canonical safety formula $\Box(\diamondsuit\neg p \rightarrow \diamondsuit q)$. The equivalence is not intuitive, but it can be easily verified with GOAL by either checking the validity of $p \mathcal{W} q \leftrightarrow \Box(\diamondsuit\neg p \rightarrow \diamondsuit q)$ or translating both formulae into Büchi automata and checking their equivalence, as shown in Figure 7.6. Further examples include $\Box p \vee \Box q \leftrightarrow \Box(\boxminus p \vee \boxminus q)$, $\neg(p \mathcal{U} \neg q) \leftrightarrow \Box(\circledcirc \boxminus p \rightarrow q)$, etc.

Understanding Why "Even p" Is QPTL-Expressible but Not PTL-Expressible

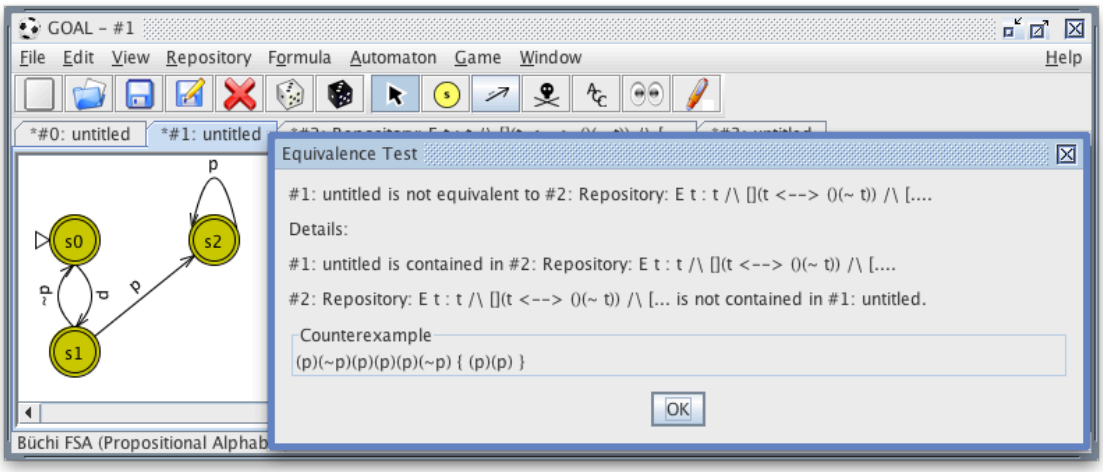
"Even p" is a typical case for showing PTL is strictly less expressive than Büchi automata. A plausible PTL formula for the property would be "$p \wedge \Box(p \rightarrow \bigcirc\bigcirc p)$". We translate the formula into a Büchi automaton, as shown in Figure 7.7(a), and open the "Even p" case in the repository, as shown in Figure 7.7(b). An equivalence test shows that the two automata are not equivalent and displays a counter-example, as shown in Figure 7.7(a) (again, an infinite word or sequence starts with position 0). The formula $p \wedge \Box(p \rightarrow \bigcirc\bigcirc p)$ is overly restrictive. Once p holds at some odd position, this formula forces p to hold at all following odd positions, which is not required by "Even p".

A correct QPTL formula is $\exists t : t \wedge \Box(t \leftrightarrow \neg\bigcirc t) \wedge \Box(t \rightarrow p)$. In this formula, t is an auxiliary variable that is true at all even positions and false at all odd positions along a computation. From the subformula $\Box(t \rightarrow p)$, we know that p must be true when t is true (at even positions), but p can be any value when t is false (at odd positions). The formula is translated into a Büchi automaton, as shown in Figure 7.7(c). One can perform an equivalence test on the translated automaton and the one in the repository to be assured that the formula indeed expresses "Even p".

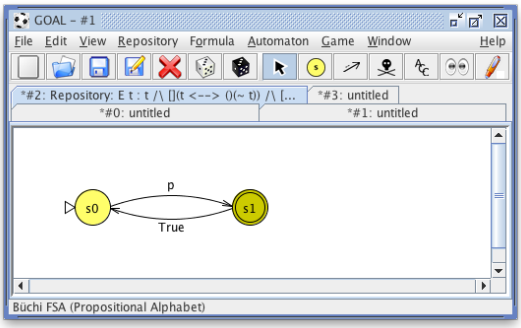
Understanding Temporal Assume-Guarantee Formulae

Informally, an assume-guarantee specification asserts that "some property is guaranteed while the assumption holds". In the literature [14, 93, 71], we can find at least three temporal logic formulations:

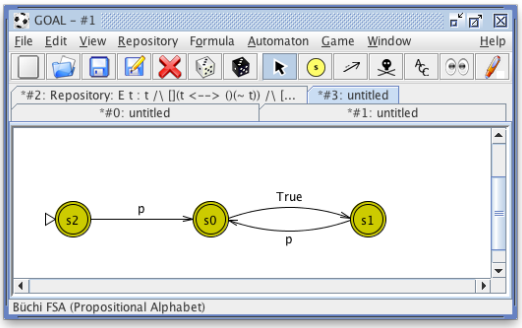
1. $\neg(p \,\mathcal{U}\, \neg q)$



(a) An automaton machine-translated from $p \wedge \Box(p \rightarrow \bigcirc\bigcirc p)$



(b) The "Even p" case from the repository



(c) An automaton machine-translated from $\exists t : t \wedge \Box(t \leftrightarrow \neg\bigcirc t) \wedge \Box(t \rightarrow p)$

Figure 7.7: Automata intended for "Even p"; the one in (a) is incorrect.

2. $\square(\circleddash \boxminus p \rightarrow q)$

3. $q \mathcal{W} (\neg p \wedge q)$

Though quite different in appearance, all these three formulae are in fact equivalent, which can be easily confirmed with GOAL. There is another similar but weaker formula $\square(\boxminus p \rightarrow \boxminus q)$ [93]. The formula can be translated into an equivalent Büchi automaton and checked to be *inequivalent* to any of the previous three formulae. Counterexamples from the tests should be helpful in understanding the difference.

7.1.3 Implementation Details

GOAL is implemented in Java and built upon a third-party library called Java Plugin Framework [43]. The structure of GOAL is shown in Figure 7.8. With the help of Java Plugin Framework, GOAL can be easily extended by third-party plugins. The user may add a new menu item, command, or algorithm with a plugin simply by extending the classes or interfaces of GOAL and then writing a configuration file for the plugin. During runtime, GOAL will read the configuration file and enable the third-party plugin. In fact, GOAL itself is composed of three plugins, namely CORE, UI, and CMD, where CORE provides basic data structures and implementations of algorithms, UI provides a graphical interface, and CMD provides a command-line interface.

7.2 Büchi Store

Büchi Store [96, 97] is a Web-based application containing an extensible repository of ω-automata and temporal formulae. In the following, we will describe the features of Büchi Store, some use cases, and its implementation details.

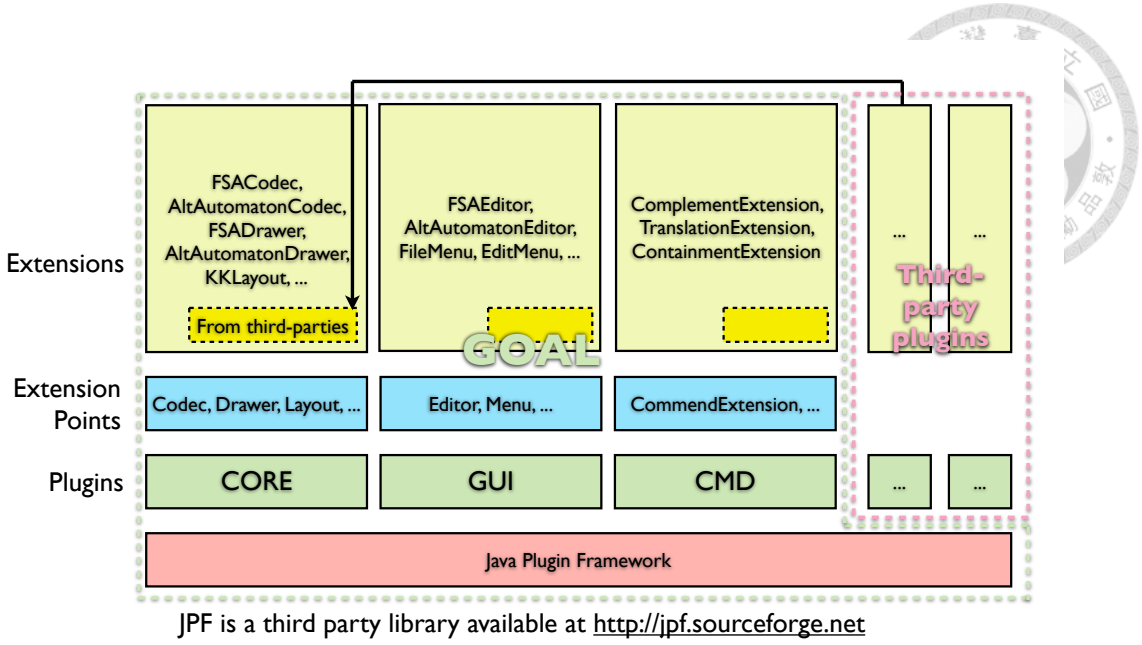


Figure 7.8: The structure of GOAL

7.2.1 Features

Automata in Büchi Store are grouped into equivalence classes induced by the languages that they recognize. For each equivalence class and each type of automaton, only the three smallest automata known to the Store are kept. Every automaton can be identified by a temporal formula in QPTL [82, 83], of which the well-known LTL is a subset.

The main features of the current Store include:

- **Search**: The user may find the automata that recognize a particular language by posing a query with an equivalent temporal formula. Formula rewriting based on congruence relations [66, 23] is performed and propositions are automatically renamed through unification, to increase matches (semantic matching is not attempted online due to its high cost). For instance, a search of $q\,\mathcal{U}\,(q\,\mathcal{U}\,(q\,\mathcal{U}\,p))$ will get the same results as that of $p\,\mathcal{U}\,q$, since $q\,\mathcal{U}\,(q\,\mathcal{U}\,(q\,\mathcal{U}\,p))$ is equivalent (via the congruence $q\,\mathcal{U}\,(q\,\mathcal{U}\,p) \equiv q\,\mathcal{U}\,p$) to $q\,\mathcal{U}\,p$, which does not

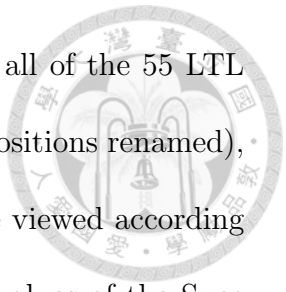
exit in the Store but matches with $p \,\mathcal{U}\, q$ via unification. An "autocomplete" feature can suggest to the user formulae that match an incomplete input formula up to propositions renaming, as illustrated in Figure 1.1. In the case of Büchi automata, the search is like asking for a translation from the temporal formula into an equivalent Büchi automaton. A big difference is that the answer automata, if present in the Store, are the best among the results obtained from a large number of translation algorithms, enhanced with various optimization techniques such as simplification by simulation [86] or even manually optimized (and machine-checked for correctness). If no temporal formula in the Store matches the input, a Büchi automaton is generated on demand by taking the best result from several translation and simplification algorithms (without manual optimization). A background process of the Store will later check whether the generated automaton indeed represents a new language and take appropriate actions.

No.	Pattern	LTL Formula
	Absence (p is false)	
1	Globally	$\Box\neg p$
2	Before q	$\Diamond q \rightarrow (\neg p \,\mathcal{U}\, q)$
3	After q	$\Box(q \rightarrow \Box(\neg p))$
4	Between q and r	$\Box((q \wedge \neg r \wedge \Diamond r) \rightarrow (\neg p \,\mathcal{U}\, r))$
5	After q until r	$\Box(q \wedge \neg r \rightarrow (\neg p \,\mathcal{W}\, r))$
	Uiversality (p is true)	
6	Globally	$\Box p$
7	Before q	$\Diamond q \rightarrow (p \,\mathcal{U}\, q)$
8	After q	$\Box(q \rightarrow \Box p)$
9	Between q and r	$\Box((q \wedge \neg r \wedge \Diamond r) \rightarrow (p \,\mathcal{U}\, r))$
10	After q until r	$\Box(q \wedge \neg r \rightarrow (p \,\mathcal{W}\, r))$
	Existence (p becomes true)	
11	Globally	$\Diamond p$
12	Before q	$\neg q \,\mathcal{W}\, (p \wedge \neg q)$
13	After q	$\Box(\neg q) \vee \Diamond(q \wedge \Diamond p))$
14	Between q and r	$\Box(q \wedge \neg r \rightarrow (\neg r \,\mathcal{W}\, (p \wedge \neg r)))$
15	After q until r	$\Box(q \wedge \neg r \rightarrow (\neg r \,\mathcal{U}\, (p \wedge \neg r)))$
	Bounded Existence (p goes from false to true at most twice)	
16	Globally	$(\neg p \,\mathcal{W}\, (p \,\mathcal{W}\, (\neg p \,\mathcal{W}\, (p \,\mathcal{W}\, \Box\neg p))))$
17	Before q	$\Diamond q \rightarrow ((\neg p \wedge \neg q) \,\mathcal{U}\, (q \vee ((p \wedge \neg q) \,\mathcal{U}\, (q \vee ((\neg p \wedge \neg q) \,\mathcal{U}\, (q \vee ((p \wedge \neg q) \,\mathcal{U}\, (q \vee (\neg p \,\mathcal{U}\, q)))))))))$
18	After q	$\Diamond q \rightarrow (\neg q \,\mathcal{U}\, (q \wedge (\neg p \,\mathcal{W}\, (p \,\mathcal{W}\, (\neg p \,\mathcal{W}\, (p \,\mathcal{W}\, \Box\neg p))))))$
19	Between q and r	$\Box((q \wedge \Diamond r) \rightarrow ((\neg p \wedge \neg r) \,\mathcal{U}\, (r \vee ((p \wedge \neg r) \,\mathcal{U}\, (r \vee ((\neg p \wedge \neg r) \,\mathcal{U}\, (r \vee ((p \wedge \neg r) \,\mathcal{U}\, (r \vee (\neg p \,\mathcal{U}\, r)))))))))$
20	After q until r	$\Box(q \rightarrow ((\neg p \wedge \neg r) \,\mathcal{U}\, (r \vee ((p \wedge \neg r) \,\mathcal{U}\, (r \vee ((\neg p \wedge \neg r) \,\mathcal{U}\, (r \vee ((p \wedge \neg r) \,\mathcal{U}\, (r \vee (\neg p \,\mathcal{W}\, r) \vee \Box p)))))))))$
	Precedence (q precedes p)	
21	Globally	$\neg p \,\mathcal{W}\, q$
22	Before r	$\Diamond r \rightarrow (\neg p \,\mathcal{U}\, (q \vee r))$

23	After r	$\Box\neg r \vee \Diamond(r \wedge (\neg p \,\mathcal{W}\, q))$
24	Between s and r	$\Box((s \wedge \neg r \wedge \Diamond r) \rightarrow (\neg p \,\mathcal{U}\, (q \vee r)))$
25	After s until r	$\Box(s \wedge \neg r \rightarrow (\neg p \,\mathcal{W}\, (q \vee r)))$
	Response (q responds to p)	
26	Globally	$\Box(p \rightarrow \Diamond q)$
27	Before r	$\Diamond r \rightarrow (p \rightarrow (\neg r \,\mathcal{U}\, (q \wedge \neg r))) \,\mathcal{U}\, r$
28	After r	$\Box(r \rightarrow \Box(p \rightarrow \Diamond q))$
29	Between s and r	$\Box((s \wedge \neg r \wedge \Diamond r) \rightarrow (p \rightarrow (\neg r \,\mathcal{U}\, (q \wedge \neg r))) \,\mathcal{U}\, r)$
30	After s until r	$\Box(s \wedge \neg r \rightarrow ((p \rightarrow (\neg r \,\mathcal{U}\, (q \wedge \neg r))) \,\mathcal{W}\, r))$
	Precedence Chain ((q, r) precedes p)	
31	Globally	$\Diamond p \rightarrow (\neg p \,\mathcal{U}\, (q \wedge \neg p \wedge \bigcirc(\neg p \,\mathcal{U}\, r)))$
32	Before s	$\Diamond s \rightarrow (\neg p \,\mathcal{U}\, (s \vee (q \wedge \neg p \wedge \bigcirc(\neg p \,\mathcal{U}\, r))))$
33	After s	$(\Box\neg s) \vee (\neg s \,\mathcal{U}\, (s \wedge \Diamond p \rightarrow (\neg p \,\mathcal{U}\, (q \wedge \neg p \wedge \bigcirc(\neg p \,\mathcal{U}\, r)))))$
34	Between s and t	$\Box((s \wedge \Diamond t) \rightarrow (\neg p \,\mathcal{U}\, (t \vee (q \wedge \neg p \wedge \bigcirc(\neg p \,\mathcal{U}\, r)))))$
35	After s until t	$\Box(s \rightarrow (\Diamond p \rightarrow (\neg p \,\mathcal{U}\, (t \vee (q \wedge \neg p \wedge \bigcirc(\neg p \,\mathcal{U}\, r))))))$
	Precedence Chain (p precedes (q, r))	
36	Globally	$(\Diamond(q \wedge \bigcirc\Diamond r)) \rightarrow ((\neg q) \,\mathcal{U}\, p))$
37	Before s	$\Diamond s \rightarrow ((\neg(q \wedge (\neg s) \wedge \bigcirc(\neg s \,\mathcal{U}\, (r \wedge \neg s)))) \,\mathcal{U}\, (s \vee p))$
38	After s	$(\Box\neg s) \vee ((\neg s) \,\mathcal{U}\, (s \wedge ((\Diamond(q \wedge \bigcirc\Diamond r)) \rightarrow ((\neg q) \,\mathcal{U}\, p))))$
39	Between s and t	$\Box((s \wedge \Diamond t) \rightarrow ((\neg(q \wedge (\neg t) \wedge \bigcirc(\neg t \,\mathcal{U}\, (r \wedge \neg t)))) \,\mathcal{U}\, (t \vee p)))$
40	After s until t	$\Box(s \rightarrow (\neg(q \wedge (\neg t) \wedge \bigcirc(\neg t \,\mathcal{U}\, (r \wedge \neg t))) \,\mathcal{U}\, (t \vee p) \vee \Box(\neg(q \wedge \bigcirc\Diamond r))))$
	Response Chain (p responds to (q, r))	
41	Globally	$\Box(q \wedge \bigcirc\Diamond r \rightarrow \bigcirc(\Diamond(r \wedge \Diamond p)))$
42	Before s	$\Diamond s \rightarrow (q \wedge \bigcirc(\neg s \,\mathcal{U}\, r) \rightarrow \bigcirc(\neg s \,\mathcal{U}\, (r \wedge \Diamond p))) \,\mathcal{U}\, s$
43	After s	$\Box(s \rightarrow \Box(q \wedge \bigcirc\Diamond r \rightarrow \bigcirc(\neg r \,\mathcal{U}\, (r \wedge \Diamond p))))$
44	Between s and t	$\Box((s \wedge \Diamond t) \rightarrow (q \wedge \bigcirc(\neg t \,\mathcal{U}\, r) \rightarrow \bigcirc(\neg t \,\mathcal{U}\, (r \wedge \Diamond p))) \,\mathcal{U}\, t)$
45	After s until t	$\Box(s \rightarrow (q \wedge \bigcirc(\neg t \,\mathcal{U}\, r) \rightarrow \bigcirc(\neg t \,\mathcal{U}\, (r \wedge \Diamond p))) \,\mathcal{U}\, (t \vee \Box(q \wedge \bigcirc(\neg t \,\mathcal{U}\, r) \rightarrow \bigcirc(\neg t \,\mathcal{U}\, (r \wedge \Diamond p)))))$
	Response Chain ((q, r) responds to p)	
46	Globally	$\Box(p \rightarrow \Diamond(q \wedge \bigcirc\Diamond r))$
47	Before s	$\Diamond s \rightarrow (p \rightarrow (\neg s \,\mathcal{U}\, (q \wedge \neg s \wedge \bigcirc(\neg s \,\mathcal{U}\, r)))) \,\mathcal{U}\, s$
48	After s	$\Box(s \rightarrow \Box(p \rightarrow (q \wedge \bigcirc\Diamond r)))$
49	Between s and t	$\Box((s \wedge \Diamond t) \rightarrow (p \rightarrow (\neg t \,\mathcal{U}\, (q \wedge \neg t \wedge \bigcirc(\neg t \,\mathcal{U}\, r)))) \,\mathcal{U}\, t)$
50	After s until t	$\Box(s \rightarrow (p \rightarrow (\neg t \,\mathcal{U}\, (q \wedge \neg t \wedge \bigcirc(\neg t \,\mathcal{U}\, r)))) \,\mathcal{U}\, (t \vee \Box(p \rightarrow (q \wedge \bigcirc\Diamond r))))$
	Constrained Chain Patterns ((q, r) without s responds to p)	
51	Globally	$\Box(p \rightarrow \Diamond(q \wedge \neg s \wedge \bigcirc(\neg s \,\mathcal{U}\, r)))$
52	Before t	$\Diamond t \rightarrow (p \rightarrow (\neg t \,\mathcal{U}\, (q \wedge \neg t \wedge \neg s \wedge \bigcirc((\neg t \wedge \neg s) \,\mathcal{U}\, r)))) \,\mathcal{U}\, t$
53	After t	$\Box(t \rightarrow \Box(p \rightarrow (q \wedge \neg s \wedge \bigcirc(\neg s \,\mathcal{U}\, r))))$
54	Between t and u	$\Box((t \wedge \Diamond u) \rightarrow (p \rightarrow (\neg u \,\mathcal{U}\, (q \wedge \neg u \wedge \neg s \wedge \bigcirc((\neg u \wedge \neg s) \,\mathcal{U}\, r)))) \,\mathcal{U}\, u)$
55	After t until u	$\Box(t \rightarrow (p \rightarrow (\neg u \,\mathcal{U}\, (q \wedge \neg u \wedge \neg s \wedge \bigcirc((\neg u \wedge \neg s) \,\mathcal{U}\, r)))) \,\mathcal{U}\, (u \vee \Box(p \rightarrow (q \wedge \neg s \wedge \bigcirc(\neg s \,\mathcal{U}\, r)))))$

Table 7.3: The LTL formulae in Büchi Store that correspond to those on the Spec Patterns site. We have used lower case for the atomic propositions and in many places changed the proposition names, to make the presentation consistent with that of other temporal formulae in the Store. We have also moved Universality forwards to follow Absence, as they are essentially isomorphic.

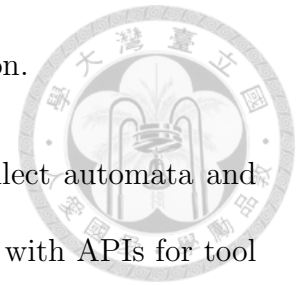
- **Browse**: The user may browse the entire collection of automata by having the collection sorted according to number of states, temporal formula length, class in the Temporal Hierarchy [65] illustrated in Figure 3.1, or class on the Spec Patterns site [87] (see Table 7.3). The several sorting keys may be applied in any order as defined by the user using a drag-and-drop interface. Types of automata may be selected via a filter. Possible values of the sorting keys also function as filters for the user to select automata of particular sizes, classes, etc.



Büchi Store contains automata and their complements for all of the 55 LTL formulae listed on the Spec Patterns site (with several propositions renamed), which are reproduced in Table 7.3. These automata can be viewed according to their classes in Spec Patterns, by choosing appropriate values of the Spec Patterns filter.

- **Upload**: The user may upload an automaton for a particular temporal formula. The uploaded automaton is checked for correctness, i.e., if it is indeed equivalent to the accompanying temporal formula. If it is correct and smaller than the third smallest automaton of the same type in the Store for the language specified by the temporal formula, the repository is updated accordingly, keeping only the three smallest automata. Unification is performed to increase the possibility of a match between the uploaded formula and an equivalent but syntactically different formula in the Store, which helps to reduce the number of syntactic variants of automata of essentially the same structure. While classification into the Temporal Hierarchy has been automated, the classification for Spec Patterns (whose pattern classes lack a semantically precise characterization) has not and, for the moment, simply follows the syntactic form of the formulae listed on the Spec Patterns site. The user may choose to upload an automaton without giving a temporal formula. In this case, a background process of the Store will check whether the automaton belongs to an existing language class in the Store. If the check is positive, the Store will send the user an email (if an email address has been provided) containing a link to the language class where the equivalent temporal formulae, if any, may be found; otherwise, a new language class as defined by the uploaded automaton

is added to the Store and the user is informed of this action.



- **Miscellany**: There is a shopping cart for the user to collect automata and download them in one single batch. The Store also comes with APIs for tool integration. Through the APIs, a tool can search in the Store for automata that are equivalent to a temporal formula. Propositions renaming of the returned automata can be performed by the Store automatically. Finally, a help page explains all of the features.

7.2.2 Use Cases

We describe a number of cases that we expect to represent typical usages of Büchi Store.

- **Linear-time model checking**: The user may shop in the Store for Büchi automata that are equivalent (with probable propositions renaming) to the negations of the temporal formulae he wants to verify. The automata may be downloaded in the PROMELA format for model checking using SPIN.

- **Benchmark cases for evaluating translation and complementation algorithms**: Büchi Store has a substantial collection of automata with their equivalent temporal formulae. A subset of the collection may serve as a set of benchmark cases for evaluating translation algorithms. Table 7.5 suggests how such an evaluation may be carried out using the negations of the LTL formulae in Spec Patterns. This use case can certainly be adapted for evaluating Büchi complementation algorithms. All Büchi automata in the current collection, including those for the LTL formulae in Spec Patterns, have their complements in the Store which are reasonably well optimized. Table 7.6 suggests how such

n	Büchi Store		LTL2AUT [17]		Couvreur [16]		LTL2BA [32]		LTL2Buchi [35]		Spin [42]	
	state	tran.	state	tran.	state	tran.	state	tran.	state	tran.	state	tran.
1	4	16	10	53	4	25	4	16	5	34	4	25
2	9	144	50	1278	13	440	11	201	11	362	9	289
3	16	1024	179	20153	18	2681	32	2860	19	2906	16	2401
4	25	6400	485	242069	29	18993	50	20686	30	19954	25	16641
5	36	36864	-	-	69	203296	-	-	59	167545	36	103041
6	49	200704	-	-	-	-	-	-	-	-	49	591361

Table 7.4: A comparison of the results from translating formulae of the form $\Diamond(p_1 \wedge \Diamond(p_2 \wedge \Diamond(\cdots \wedge \Diamond(p_{n-1} \wedge \Diamond p_n)\cdots))) \wedge \Diamond(q_1 \wedge \Diamond(q_2 \wedge \Diamond(\cdots \wedge \Diamond(q_{n-1} \wedge \Diamond q_n)\cdots)))$. Under the title of every column except the first, "state" and "tran." stand for "number of states" and "number of transitions" respectively; "-" indicates out of memory (1 GB) or timeout (1 hour). Implementations of the algorithms here were not optimized for performance, but were made as faithful as possible to the respective original publications in terms of the structure of the generated automaton. The implementation of Couvreur's algorithm here uses an explicit automata representation and also the conversion algorithm applied in LTL2BA for converting transition-based generalized Büchi automata to Büchi automata. The implementation of LTL2BA is different from [32] in the definitions of transition relations; however, the results are very close to those in [13] for the same formulae. Except for `Spin` (over which we do not have control), no optimization was attempted on the input formulae or the result automata.

an evaluation of complementation algorithms may be carried out.

No.	Büchi Store		LTL2AUT [17]		Couvreur [16]		LTL2BA [32]		LTL2Buchi [35]		Spin [42]	
	state	tran.	state	tran.	state	tran.	state	tran.	state	tran.	state	tran.
1	2	4	3	7	2	5	2	4	2	5	2	5
2	3	10	5	23	3	13	3	10	4	16	3	13
3	3	12	7	35	3	17	3	13	3	17	3	17
4	4	27	10	94	7	66	4	30	6	55	4	37
5	3	19	7	55	3	25	3	21	3	25	3	25
6	2	4	3	7	2	5	2	4	2	5	2	5
7	3	10	5	23	3	13	3	10	4	16	3	13
8	3	12	8	42	5	30	3	13	4	23	3	17
9	4	27	10	94	7	66	4	30	6	55	4	37
10	3	19	7	55	3	25	3	21	3	25	3	25
11	1	1	1	1	1	1	1	1	1	1	1	1
12	2	7	4	20	2	8	2	7	2	9	2	9
13	2	5	4	12	2	5	2	5	4	15	3	10
14	3	22	10	101	3	25	3	23	3	27	3	27
15	3	23	6	61	3	25	3	23	3	27	3	27
16	6	12	136	985	9	19	7	14	24	149	80	298
17	7	18	24	140	7	21	7	18	21	165	7	25
18	7	28	582	16298	13	54	13	52	144	3243	-	-
19	8	43	32	401	8	52	8	46	19	249	10	88
20	9	45	117	1788	9	48	9	46	49	896	-	-
21	2	6	3	11	2	7	2	6	2	7	2	7
22	3	18	5	43	3	23	3	18	4	26	4	26
23	3	16	12	116	5	35	5	26	6	54	6	54
24	4	51	9	151	4	65	4	54	4	65	5	71
25	3	35	8	114	5	66	3	37	4	47	3	41
26	2	7	3	9	2	7	2	7	2	7	2	7
27	3	18	8	93	3	21	3	19	7	70	6	65
28	3	23	7	49	3	27	3	27	3	27	3	27
29	4	52	18	414	7	110	4	59	8	161	7	151
30	4	54	13	249	4	63	4	59	4	67	4	67
31	3	20	26	428	3	30	3	20	11	152	14	206
32	4	44	20	360	4	62	4	44	7	116	6	98
33	4	48	134	4848	4	76	4	48	23	724	40	1616
34	5	113	28	1018	5	172	5	128	7	246	7	246
35	4	81	54	1972	7	216	4	92	9	316	15	588

36	3	30	18	166	5	52	5	42	9	88	8	78
37	4	44	9	132	4	58	6	60	5	64	8	94
38	5	68	399	15074	9	176	9	148	53	1983	78	3412
39	5	119	22	658	8	228	5	130	7	190	9	226
40	5	105	88	2802	19	576	16	370	18	582	16	502
41	3	20	7	50	4	28	4	28	5	42	5	42
42	4	34	16	188	4	42	6	56	7	88	9	120
43	5	76	25	388	5	84	5	84	9	168	10	184
44	5	100	37	980	9	236	6	132	13	384	10	284
45	5	106	102	2904	8	216	6	132	31	924	20	584
46	3	24	6	50	3	30	3	24	3	30	4	42
47	4	47	16	370	4	74	4	52	11	244	15	426
48	4	62	11	182	4	82	4	68	4	82	4	82
49	5	126	33	1646	5	194	5	148	9	450	16	898
50	9	248	124	5568	13	460	13	324	21	924	33	1624
51	3	48	9	228	3	70	3	48	3	70	5	110
52	4	93	33	2744	4	166	4	104	9	484	15	908
53	4	118	16	608	6	300	4	128	5	238	4	178
54	5	250	75	13056	5	426	5	296	10	1162	16	1910
55	13	594	278	47500	16	1532	14	688	26	3226	34	4366

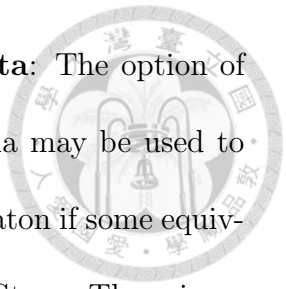
Table 7.5: A comparison of the results from translating the negations of the LTL formulae in Spec Patterns (see Table 7.3). "-" indicates timeout (10 minutes). The notes about the translation algorithms in Table 7.4 also apply here.

No.	Büchi Store		Safra [77]		Piterman [74]		Rank-Based [57, 79]		Slice-Based [47]	
	state	tran.	state	tran.	state	tran.	state	tran.	state	tran.
1	2	4	6	18	3	9	2	4	3	6
2	3	10	16	70	6	24	4	12	8	25
3	3	12	8	48	4	23	3	12	4	16
4	4	27	25	256	6	60	4	32	21	161
5	3	19	8	96	4	43	3	24	4	32
6	2	4	6	18	3	9	2	4	3	6
7	3	10	16	70	6	24	4	12	8	25
8	3	12	8	48	4	23	3	12	4	16
9	4	27	25	256	6	60	4	32	21	161
10	3	19	8	96	4	43	3	24	4	32
11	1	1	2	3	3	5	2	3	2	2
12	2	7	6	31	3	17	2	7	3	10
13	2	5	10	41	6	20	10	40	4	10
14	3	22	8	96	4	44	3	24	4	32
15	3	23	13	144	5	53	4	29	5	36
16	6	12	14	42	7	17	6	12	7	14
17	7	18	28	100	11	34	8	20	16	40
18	7	28	16	96	8	38	7	28	8	32
19	8	43	79	816	16	140	10	80	59	476
20	9	45	174	1632	24	202	15	120	220	2049
21	2	6	6	26	3	15	2	6	3	8
22	3	18	16	118	6	42	4	20	6	25
23	3	16	18	134	14	92	9	70	12	68
24	4	51	35	720	7	136	5	80	27	410
25	3	35	8	192	4	83	3	48	4	64
26	2	7	5	22	3	13	3	13	4	16
27	3	18	16	148	8	65	4	24	8	52
28	3	23	7	60	4	35	4	35	5	40
29	4	52	33	672	8	153	4	64	27	420
30	4	54	16	368	6	123	5	75	6	88
31	3	20	18	140	7	60	3	20	14	83

32	4	44	36	396	13	124	7	60	33	226
33	4	48	16	296	9	148	4	48	8	100
34	5	113	1028	35936	95	3088	16	512	2338	83936
35	4	81	1086	37920	67	2326	13	416	932	31274
36	3	30	8	76	4	40	3	20	4	24
37	4	44	27	340	9	108	6	56	12	108
38	5	68	33	516	9	132	6	72	21	250
39	5	119	75	3104	13	464	9	288	73	2244
40	5	105	103	4256	18	620	11	352	174	5552
41	3	20	10	88	6	50	6	50	7	50
42	4	34	24	354	10	108	28	346	8	76
43	5	76	19	332	11	196	10	185	28	504
44	5	100	250	8296	232	7276	1558	19355	267	10056
45	5	106	-	-	453	14946	10986	117610	-	-
46	3	24	13	125	8	77	8	77	9	72
47	4	47	22	404	11	168	5	60	11	140
48	4	62	21	480	7	148	6	98	7	108
49	5	126	43	1760	11	396	5	160	38	1160
50	9	248	868	31328	171	5082	1098	15292	840	30607
51	3	48	13	222	14	226	7	126	13	192
52	4	93	30	1000	13	368	7	152	23	520
53	4	118	21	960	7	311	6	181	7	208
54	5	250	57	4672	13	920	7	448	60	3664
55	13	594	634	45888	73	4904	219	8286	616	38171

Table 7.6: A comparison of the results from complementing the automata for the LTL formulae in Spec Patterns (see Table 7.3). "-" indicates timeout (10 minutes). The implementations of complementation algorithms here follow the respective original publications as faithfully as possible without applying any other additional optimization to the input automata or to the resulting complements. Naive conversions from Streett automata and parity automata to Büchi automata are applied in Safra and Piterman. For Slice-Based, the transition relation of an input automaton is made complete, as required by the original algorithm, before complementation.

- **Classification of temporal formulae**: The look of a temporal formula may not tell immediately to which class it belongs in the Temporal Hierarchy. It should be educational to practice on the cases that do not involve going through complicated operations on automata. For example, $\Box p \vee \Box q$ is a safety formula because it is equivalent to $\Box(\boxminus p \vee \boxminus q)$ in the canonical safety form (where $\boxminus$ means "so-far" or "always in the past"). Another formula $\Box(p \rightarrow \Diamond q)$ is a recurrence formula because it is equivalent to $\Box\Diamond(\neg p \mathcal{B} q)$ (where $\mathcal{B}$ means "back-to", the past version of "wait-for" or "weak until").



- **Reverse lookup of temporal formulae from automata**: The option of uploading an automaton without giving a temporal formula may be used to find temporal formulae that are equivalent to a given automaton if some equivalent automaton with temporal formulae is present in the Store. There is no guarantee of success. However, as the collection of automata and temporal formulae in the Store continues to grow, the success rate of such reverse lookups should increase accordingly.

- **Synthesis**: Deterministic parity automata are widely used in the synthesis of a reactive system that satisfies desired specifications expressed in linear temporal logic. For example, the quantitative synthesis tool, QUASY [12], takes as qualitative specifications deterministic parity automata in the GOAL file format (GFF). Thus, the user may shop in the Store for the deterministic parity automata that are equivalent (with probable propositions renaming) to the temporal formulae to be satisfied by the system, rename propositions properly perhaps with the help of the GOAL tool, and then perform the synthesis.

- **Tool Integration** Here is a simple use case of the APIs for tool integration. Suppose the user of a model checker wants to verify if a system satisfies the temporal logic formula $\Box(\mathit{request} \rightarrow \Diamond \mathit{response})$. Instead of translating the formula into an equivalent automaton by a translation algorithm, the model checker can request equivalent automata from the Store. Since the Store contains automata equivalent to $\Box(p \rightarrow \Diamond q)$, propositions of these automata will be renamed by the Store such that the returned automata are exactly equivalent to the required formula.

7.2.3 Implementation Details

We describe below in more details the basic infrastructure of Büchi Store, the search features of propositions renaming and autocomplete, a procedure for classification into the Temporal Hierarchy based on deterministic Büchi automata, and a "containment partial order" that is helpful in calculating equivalence classes incrementally.

- **Basic Infrastructure**: The basic client-server interactions in accessing the Store are realized by customizing the CodeIgniter [15], which is an open-source Web application framework written in PHP. To perform automata and temporal formulae-related operations, such as equivalence checking and formula to automaton translation, the Store relies on the GOAL tool [94] and its recent extensions. To achieve better performance, the Store does not invoke the GOAL tool every time an automata or temporal formulae-related operation is needed. Instead, the Store interacts with a single GOAL process hosted on a Tomcat application server [90] via PHP/Java Bridge [73].

- **Propositions Renaming**: The formulae search with propositions renaming increases matches such that for example $\Box(p \rightarrow \Diamond q)$ in the Store can be found even if the queried formula is $\Box(\mathit{request} \rightarrow \Diamond \mathit{response})$. This feature is achieved by formulae normalization which renames the propositions of a formula uniformly based on the order of propositions in the parse tree of the formula such that the i-th new proposition is renamed to $\mathit{aux}i$. When searching for a formula, both the original formula entered by the user and its normalization will be used to find matches. For example, $\Box(p \rightarrow \Diamond q)$ has the normalized formula $\Box(\mathit{aux1} \rightarrow \Diamond \mathit{aux2})$ stored in the internal database. Since $\Box(\mathit{request} \rightarrow \Diamond \mathit{response})$ has the same normalized formula, $\Box(p \rightarrow \Diamond q)$ can

$$
\begin{array}{lcl}
I & ::= & C \mid \circ_u \mid \circ_u\ I \mid C\ \circ_b \mid C\ \circ_b\ I \mid \bullet \mid \bullet X \mid \bullet X\ I \mid (\mid (\ I \\
C & ::= & X \mid \circ_u\ C \mid C\ \circ_b\ C \mid \bullet X\ C \mid (\ C\ ) \\
\circ_u & ::= & \neg \mid \bigcirc \mid \Diamond \mid \Box \mid \ominus \mid \circledcirc \mid \diamondminus \mid \boxminus \\
\circ_b & ::= & \wedge \mid \vee \mid \rightarrow \mid \leftrightarrow \mid \mathcal{U} \mid \mathcal{W} \mid \mathcal{R} \mid \mathcal{S} \mid \mathcal{B} \\
\bullet & ::= & \exists \mid \forall
\end{array}
$$

Figure 7.9: The syntax of incomplete formulae I and complete formulae C where X is the set of atomic propositions.

be easily found.

- **Autocomplete**: The formulae search with autocomplete makes suggestions about formulae that may match an incomplete formula queried by the user. For example, $\Box(p \rightarrow \bigcirc p)$, $\Box(p \rightarrow \diamondminus q)$, $\Box(p \rightarrow \Diamond q)$, etc. will be suggested to the user when the incomplete formula "$\Box($*request* $\rightarrow$" is entered in the search field. This feature is achieved by parsing incomplete formulae and formulae normalization. Due to the proposition order of the formulae normalization, the syntax of incomplete formulae is restricted to the form shown in Figure 7.9 such that only the rightmost leaves of the parse tree may be missing.

- **Classification**: The classification of automata and temporal formulae into the Temporal Hierarchy may be performed by the approach in [65] through deterministic Streett automata, which was adopted by an earlier version of Büchi Store. The Store now performs the classification in a different way through deterministic Büchi automata. Consider a Büchi automaton A and its complement $\overline{A}$. A belongs to the Recurrence (Response) class if the language of A can be recognized by a deterministic Büchi automaton B [61]. The deterministic Büchi automaton B can be constructed by the approach in [61] through deterministic Muller automata or by the approach in [6] through de-

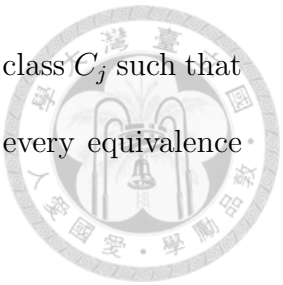
terministic co-Büchi automata[3]. A belongs to the Persistence class if $\overline{A}$ is in the Recurrence class. A is in the Reactivity class if it belongs to neither the Recurrence class nor the Persistence class, and A is in the Obligation class if it belongs to both the Recurrence class and the Persistence class. A belongs to the Safety class if the language of B is prefix-closed, which can be verified by checking that all accepting runs stay only in accepting states after maximizing the Büchi acceptance set of B without changing the recognized language [91]. In the maximization of a Büchi acceptance set, a state s becomes accepting if every cycle containing s also contains some accepting state. Finally, A belongs to the Guarantee class if $\overline{A}$ belongs to the Safety class.

- **Containment partial order**: Automata in Büchi Store are stored per language-equivalence class for consistency and for saving disk space. The grouping of automata and temporal formulae into equivalence classes is performed offline in batches because equivalence checking is time-consuming. Suppose there are n equivalence classes $C_1, C_2, \ldots, C_n$ in the Store when a new automaton A is considered. Let $B_i \in C_i$, for $1 \leq i \leq n$, be the representative automata of the equivalence classes. A naive grouping procedure may check for every equivalence class C_i if A is equivalent to B_i and, if it is the case, add A to C_i. To accelerate the grouping, Büchi Store keeps a "containment partial order"[4]

[3]The main construction in [6] may be used to obtain a DCW from $\overline{A}$ that is either exactly equivalent to $\overline{A}$ if $\overline{A}$ is DCW-recognizable or an over-approximation of $\overline{A}$. We check the equivalence between the constructed DCW and $\overline{A}$ to see which case it is. If it is the former case, then the DCW equivalent to $\overline{A}$ can be easily complemented to get a DBW that is equivalent to A, affirming that A belongs to the Recurrence class.

[4]The containment partial order maintained in the Store is a (very small) portion of the lattice of ω-regular languages over a finite alphabet (of boolean combinations of atomic propositions, determined by the automaton with the most number of atomic propositions) where the partial order is language containment and the meet and join are respectively the intersection and union of sets. The maintained portion itself may not be a lattice and rather is just a succinct representation of the containment relation between the languages/automata in the Store.

of the equivalence classes. If $A \not\subseteq B_i$, then every equivalence class C_j such that $B_j \subseteq B_i$ need not be checked. Similarly, if $B_i \not\subseteq A$, then every equivalence class C_j such that $B_i \subseteq B_j$ need not be checked.



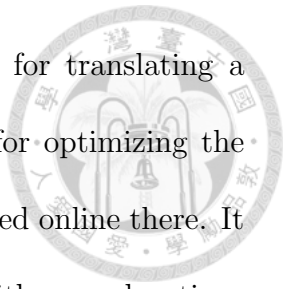
7.3 Summary of this Chapter

We have described GOAL, a graphical interactive tool designed for ω-automata and temporal logics. As the source "**G**raphical Tool for **O**mega-**A**utomata and **L**ogics" of the acronym GOAL suggests, our long-term goal is for the tool to handle the common variants of ω-automata and the logics that are expressively equivalent to these automata. For example, besides Büchi and generalized Büchi automata, we have extended GOAL to support the editing of and a limited set of direct operations on Muller, Rabin, Streett, and parity automata [36], as well as alternating automata and two-way alternating automata. Although these variants of ω-automata do not necessarily have a direct impact on the model-checking process, they are powerful intermediaries for the development of automata-based algorithms and will make GOAL complete as a learning and teaching tool.

We have also described Büchi Store, a Web-based repository that contains a large and expanding collection of Büchi automata and other types of ω-automata. Every automaton in the current collection is described by at least one temporal formula and can be easily searched.[5] With various options for sorting and filtering, browsing the collection is also fairly convenient to do.

Büchi Store appears to be unique in the features that it provides. When only Büchi automata are of concern, the most closely-related tool perhaps is the Web

[5]This may no longer be true when a user uploads an automaton without a temporal formula such that the automaton defines a new language.



version of LTL2BA [31]. LTL2BA provides several algorithms for translating a temporal formula into a Büchi automaton and several options for optimizing the automaton during or after the translation. Translation is performed online there. It is left for the user to try out different combinations of the algorithms and options to generate the best result. In contrast, Büchi Store provides a single access point for the user to get the best known Büchi automata for a temporal formula. No translation is performed online here, except in the case that no syntactic match for the temporal formula is found. A manually-optimized automaton may be uploaded and checked for correctness (with respect to the given temporal formula). It can be smaller than any machine-translated automaton. Another advantage of Büchi Store is that past temporal operators are supported.

Chapter 8

Conclusion

The automata-theoretic approach to model checking has been developed for near three decades. Because of its proven effectiveness and ease of use, the approach has been a viable alternative to testing and simulation in industry. In this approach, translation of temporal formulae, complementation of Büchi automata, and containment testing of Büchi automata play important roles. A more efficient translation, complementation, or containment testing can make the model checking process faster in general.

8.1 Contributions

In this dissertation, we make the following contributions.

- Improved translation algorithms for PTL formulae: A backtrace procedure is proposed to extend both state-based and transition-based incremental algorithms to support past operators. Two optimization heuristics are also proposed to produce smaller automata.

- Improved complementation constructions for Büchi automata: Two optimization heuristics are proposed for the Safra-Piterman construction. One optimization heuristic is proposed for the rank-based construction and is also

applicable to other constructions. Three optimization heuristics are proposed for the slice-based construction.

- An incremental containment testing algorithm based on the determinization-based approach.

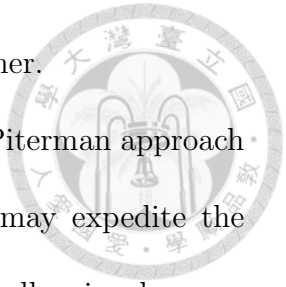
- Comparative experimentation on complementation constructions, on translation algorithms for PTL formulae, and on containment testing algorithms.

- Two tools, namely GOAL and Büchi Store, which can provide an environment for education and research on ω-automata and temporal logics.

8.2 Future Work

One possible direction for future work is to apply the extended translation algorithms to truly on-the-fly model checking, even if past operators are involved. The more recent work [40] that interweaves simplification operations with the on-the-fly states construction seems promising. It will be interesting to see if our techniques can be applied to extend this type of model checking algorithms to handle past operators.

As the experimental results in Chapter 5 showed, the nondeterministic constructions `Rank` and `Slice` produced many more dead states of complements. One reason is that there are many nondeterministic choices (rank functions in the rank-based approach and decorations in the slice-based approach) but only few of them are correct. While Friedgut *et al.* proposed tight ranking in [27] to reduce the number of ranking functions for the rank-based approach, we proposed the heuristic of deterministic decoration to alleviate this problem for the slice-based approach. However, the improved constructions `Rank+A` and `Slice+ADRM` still produced many more dead states compared to `SP+ASE` in our experiments. There may be other opportunities



to improve the rank-based and the slice-based constructions further.

Consider incremental containment testing, althoug the Safra-Piterman approach already performs better than the Ramsey-based approach, we may expedite the testing procedure with a subsumption relation. This will be challenging because the structure of compact Safra trees is complicated and checking the subsumption relation between two compact Safra trees may consume more time than that it saves.

To further improve Büchi Store, we can extend the repository by featured collections such as a collection of both all the two-state Büchi automata with two propositions and their shortest equivalent formulae. The challenge of this improvement is that finding a shortest formula equivalent to an automaton is hard. The work in [51] can construct a QPTL formula equivalent to a specified Büchi automaton but quantifications are not required for counter-free automata that are actually as expressive as PTL. We can also provide explanatory descriptions other than temporal formulae. Such descriptions would be helpful additions for searching and understanding. A more systematic classification of temporal formulae into the various specification patterns, which cannot be fully automated though, should also be useful.

To improve GOAL, we can implement translation algorithms for IEEE Property Specification Language (PSL) such that GOAL can be more practical for the industry. The visualization and manipulation of tree automata would also be interesting. Besides, there are still many algorithms for automata and temporal logics not available in GOAL. During the implementation of these algorithms, we would get better understanding of the algorithm and may find improvements, whose performance can be verified easily by experiments with the help of GOAL and Büchi Store.

Bibliography

[1] P. Abdulla, Y.-F. Chen, L. Clemente, L. Holík, C.-D. Hong, R. Mayr, and T. Vojnar. Simulation subsumption in Ramsey-based Büchi automata universality and inclusion testing. In *CAV 2010*, LNCS 6174, pages 132–147. Springer, 2010.

[2] P. Abdulla, Y.-F. Chen, L. Clemente, L. Holík, C.-D. Hong, R. Mayr, and T. Vojnar. Advanced Ramsey-based Büchi automata inclusion testing. In *CONCUR 2011*, LNCS 6901, pages 187–202. Springer, 2011.

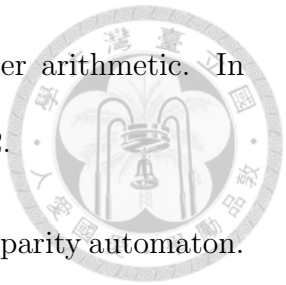
[3] P. Abdulla, Y.-F. Chen, L. Holík, R. Mayr, and T. Vojnar. When simulation meets antichains. In *TACAS 2010*, LNCS 6015, pages 158–174. Springer, 2010.

[4] C. Althoff, W. Thomas, and N. Wallmeier. Observations on determinization of Büchi automata. *TCS*, 363(2):224–233, 2006.

[5] A. Ben-Amram and C. Lee. Program termination analysis in polynomial time. *TOPLAS*, 29(1), 2007.

[6] U. Boker and O. Kupferman. Co-ing Büchi made tight and useful. In *LICS 2009*, pages 245–254. IEEE Computer Society, 2009.

[7] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE TC*, 35(8):677–691, 1986.



[8] J. Büchi. On a decision method in restricted second order arithmetic. In *ICLMPS 1960*, pages 1–12. Stanford University Press, 1962.

[9] O. Carton and R. Maceiras. Computing the Rabin index of a parity automaton. *ITA*, 33(6):495–506, 1999.

[10] I. Cerná and R. Pelánek. Relating hierarchy of temporal properties to model checking. In *MFCS*, LNCS 2747, pages 318–327. Springer, 2003.

[11] K. Chatterjee, T. Henzinger, B. Jobstmann, and A. Radhakrishna. GIST: A solver for probabilistic games. In *CAV 2010*, LNCS 6174, pages 665–669. Springer, 2010.

[12] K. Chatterjee, T. Henzinger, B. Jobstmann, and R. Singh. QUASY: Quantitative synthesis tool. In *TACAS 2011*, LNCS 6605, pages 267–271. Springer, 2011.

[13] J. Cichoń, A. Czubak, and A. Jasiński. Minimal Büchi automata for certain classes of LTL formulas. In *DepCoS-RELCOMEX*, pages 17–24. IEEE, 2009.

[14] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 2000.

[15] CodeIgniter. `http://codeigniter.com/`.

[16] J.-M. Couvreur. On-the-fly verification of linear temporal logic. In *FM*, LNCS 1708, pages 253–271. Springer, 1999.

[17] M. Daniele, F. Giunchiglia, and M. Vardi. Improved automata generation for linear temporal logic. In *CAV 1999*, LNCS 1633, pages 249–260. Springer, 1999.

[18] M. De Wulf, L. Doyen, T. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *CAV*, LNCS 4144, pages 17–30. Springer, 2006.

[19] V. Diekert and P. Gastin. First-order definable languages. In *Logic and Automata: History and Perspective*, TLG 2, pages 261–306. Amsterdam University Press, 2008.

[20] L. Doyen and J.-F. Raskin. Improved algorithms for the automata-based approach to model-checking. In *TACAS 2007*, LNCS 4424, pages 451–465. Springer, 2007.

[21] L. Doyen and J.-F. Raskin. Antichains for the automata-based approach to model-checking. *LMCS*, 5(1:5):1–20, 2009.

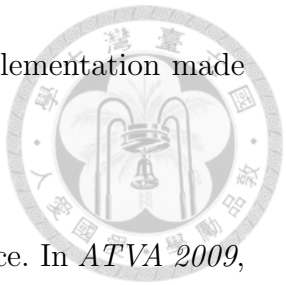
[22] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *ICSE 1999*, pages 411–420. ACM, 1999.

[23] K. Etessami and G. Holzmann. Optimizing Büchi automata. In *CONCUR 2000*, LNCS 1877, pages 153–167. Springer, 2000.

[24] K. Etessami, T. Wilke, and R. A. Schuller. Fair simulation relations, parity games, and state space reduction for Büchi automata. In *ICALP 2011*, LNCS 2076, pages 694–707. Springer, 2001.

[25] S. Fogarty and M. Vardi. Büchi complementation and size-change termination. In *TACAS 2009*, LNCS 5505, pages 16–30. Springer, 2009.

[26] S. Fogarty and M. Vardi. Efficient Büchi universality checking. In *TACAS 2010*, LNCS 6015, pages 205–220. Springer, 2010.



[27] E. Friedgut, O. Kupferman, and M. Y. Vardi. Büchi complementation made tighter. *IJFCS*, 17(4):851–868, 2006.

[28] O. Friedmann and M. Lange. Solving parity games in practice. In *ATVA 2009*, LNCS 5799, pages 182–196. Springer, 2009.

[29] D. Gabbay. The declarative past and imperative future: Executable temporal logic for interactive systems. In *TLS 1987*, LNCS 398, pages 409–448. Springer, 1987.

[30] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *POPL 1980*, pages 163–173. ACM Press, 1980.

[31] P. Gastin and D. Oddoux. LTL2BA: fast translation from LTL formulae to Büchi automata. `http:// www.lsv.ens-cachan.fr/~gastin/`

[32] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *CAV 2001*, LNCS 2102, pages 53–65. Springer, 2001.

[33] P. Gastin and D. Oddoux. LTL with past and two-way very-weak alternating automata. In *MFCS 2003*, LNCS 2747, pages 439–448. Springer, 2003.

[34] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *PSTV*, pages 3–18. Chapman & Hall, 1995.

[35] D. Giannakopoulou and F. Lerda. From states to transitions: Improving translation of LTL formulae to Büchi automata. In *FORTE 2002*, LNCS 2529, pages 308–326. Springer, 2002.

[36] E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, LNCS 2500. Springer, 2002.

[37] O. Grumberg and D. Long. Model checking and modular verification. *TOPLAS*, 16(3):843–871, 1994.

[38] S. Gurumurthy, R. Bloem, and F. Somenzi. Fair simulation minimization. In *CAV 2002*, LNCS 2404, pages 610–624. Springer, 2002.

[39] S. Gurumurthy, O. Kupferman, F. Somenzi, and M. Vardi. On complementing nondeterministic Büchi automata. In *CHARME 2003*, LNCS 2860, pages 96–110. Springer, 2003.

[40] M. Hammer, A. Knapp, and S. Merz. Truly on-the-fly LTL model checking. In *TACAS 2005*, LNCS 3440, pages 191–205, 2005.

[41] M. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In *FOCS 1995*, pages 453–462. IEEE Computer Society, 1995.

[42] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[43] Java Plugin Framework. `http://jpf.sourceforge.net`.

[44] D. Johnson. Finding all the elementary circuits of a directed graph. *SICOMP*, 4(1):77–84, 1975.

[45] M. Jurdziński. Small progress measures for solving parity games. In *STACS 2000*, LNCS 1770, pages 290–301. Springer, 2000.

[46] M. Jurdziński, M. Paterson, and U. Zwick. A deterministic subexponential algorithm for solving parity games. *SICOMP*, 38(4):1519–1532, 2008.

[47] D. Kähler and T. Wilke. Complementation, disambiguation, and determinization of Büchi automata unified. In *ICALP 2008*, LNCS 5125, pages 724–735. Springer, 2008.

[48] H. Karmarkar and S. Chakraborty. On minimal odd rankings for Büchi complementation. In *ATVA 2009*, LNCS 5799, pages 228–243. Springer, 2009.

[49] Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A decision algorithm for full propositional temporal logic. In *CAV 1993*, LNCS 697, pages 97–109. Springer, 1993.

[50] Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, 163(1):203–243, 2000.

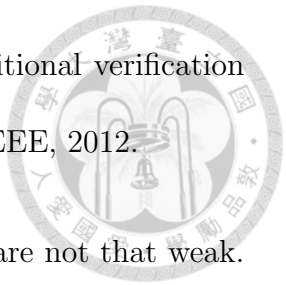
[51] Y. Kesten and A. Pnueli. Complete proof system for QPTL. *IJLP*, 12(5):701–745, 2002.

[52] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In *ICALP 1998*, LNCS 1443, pages 1–16. Springer, 1998.

[53] V. King, O. Kupferman, and M. Vardi. On the complexity of parity word automata. In *FoSSaCS 2001*, LNCS 2030, pages 276–286. Springer, 2001.

[54] N. Klarlund. Progress measures for complementation of ω-automata with applications to temporal logic. In *FOCS 1991*, pages 358–367. IEEE, 1991.

[55] J. Klein and C. Baier. Experiments with deterministic ω-automata for formulas of linear temporal logic. *TCS*, 363(2):182–195, 2006.



[56] T. Klotz, N. Seßler, B. Straube, and E. Fordran. Compositional verification of material handling systems. In *ETFA 2012*, pages 1–8. IEEE, 2012.

[57] O. Kupferman and M. Vardi. Weak alternating automata are not that weak. *TOCL*, 2(3):408–429, 2001.

[58] O. Kupferman and M. Vardi. Safraless decision procedures. In *FOCS 2005*, pages 531–540. IEEE Computer Society, 2005.

[59] R. Kurshan. Complementing deterministic Büchi automata in polynomial time. *JCSS*, 35(1):59–71, 1987.

[60] R. Küsters. Memoryless determinacy of parity games. In *Automata, Logics, and Infinite Games*, LNCS 2500, pages 95–106. Springer, 2001.

[61] L. H. Landweber. Decision problems for ω-automata. *MST*, 3(4):376–384, 1969.

[62] F. Laroussinie, N. Markey, and P. Schnoebelen. Temporal logic with forgettable past. In *LICS 2002*, pages 383–392. IEEE, 2002.

[63] T. Latvala, A. Biere, K. Heljanko, and T. A. Junttila. Simple is better: Efficient bounded model checking for past LTL. In *VMCAI 2005*, LNCS 3385, pages 380–395. Springer, 2005.

[64] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Logic of Programs*, LNCS 193, pages 196–218. Springer, 1985.

[65] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *PODC 1990*, pages 377–410. ACM, 1990.

[66] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer, 1995.

[67] R. Mazala. Infinite games. In *Automata, Logics, and Infinite Games*, LNCS 2500, pages 23–42. Springer, 2001.

[68] R. McNaughton. Infinite games played on finite graphs. *APAL*, 65(2):149–184, 1993.

[69] M. Michel. Complementation is more difficult with automata on infinite words. In *CNET*, 1988.

[70] D. Muller and P. Schupp. Simulating alternating tree automata by nondeterministic automata: New results and new proofs of the theorems of Rabin, McNaughton and Safra. *TCS*, 141(1&2):69–107, 1995.

[71] K. Namjoshi and R. Trefler. On the completeness of compositional reasoning. In *CAV 2000*, LNCS 1855, pages 139–153. Springer, 2000.

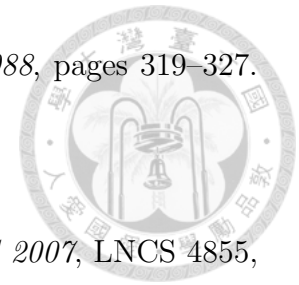
[72] H. Peng, Y. Mokhtari, and S. Tahar. Environment synthesis for compositional model checking. In *ICCD 2002*, pages 70–75. IEEE, 2002.

[73] PHP/Java Bridge. `http://php-java-bridge. sourceforge.net/`.

[74] N. Piterman. From nondeterministic Büchi and Streett automata to deterministic parity automata. *LMCS*, 3(3), 2007.

[75] M. Rabin and D. Scott. Finite automata and their decision problems. *IBM JRD*, 3:115–125, 1959.

[76] S. Rodger and T. Finley. JFLAP. `http://www.jflap.org/`.



[77] S. Safra. On the complexity of ω-automata. In *FOCS 1988*, pages 319–327. IEEE, 1988.

[78] S. Schewe. Solving parity games in big steps. In *FSTTCS 2007*, LNCS 4855, pages 449–460. Springer, 2007.

[79] S. Schewe. Büchi complementation made tight. In *STACS 2009*, LIPIcs 3, pages 661–672. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2009.

[80] S. Schewe. Tighter bounds for the determinisation of Büchi automata. In *FOSSACS 2009*, LNCS 5504, pages 167–181. Springer, 2009.

[81] R. Sebastiani and S. Tonetta. "More deterministic" vs. "smaller" Büchi automata for efficient LTL model checking. In *CHARME 2003*, LNCS 2860, pages 126–140. Springer, 2003.

[82] A. Sistla. *Theoretical Issues in the Design and Verification of Distributed Systems*. PhD thesis, Harvard, 1983.

[83] A. Sistla, M. Vardi, and P. Wolper. The complementation problem for büchi automata with appplications to temporal logic. *TCS*, 49:217–237, 1987.

[84] S. Sohail and F. Somenzi. Safety first: A two-stage algorithm for LTL games. In *FMCAD 2009*, pages 77–84. IEEE, 2009.

[85] S. Sohail, F. Somenzi, and K. Ravi. A hybrid algorithm for LTL games. In *VMCAI 2008*, LNCS 4905, pages 309–323. Springer, 2008.

[86] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *CAV 2000*, LNCS 1855, pages 248–263. Springer, 2000.

[87] The Spec Patterns repository. http://patterns.projects.cis.ksu.edu/.

[88] H. Tauriainen and K. Heljanko. Testing LTL formula translation into Büchi automata. *STTT*, 4(1):57–70, 2002.

[89] W. Thomas. Complementation of Büchi automata revisited. In *Jewels are Forever*, pages 109–120. Springer, 1999.

[90] Tomcat. `http://tomcat.apache.org/`.

[91] M.-H. Tsai, S. Fogarty, M. Vardi, and Y.-K. Tsay. State of Büchi complementation. In *CIAA 2010*, LNCS 6482, pages 261–271. Springer, 2010.

[92] M.-H. Tsai, Y.-K. Tsay, and Y.-S. Hwang. GOAL for games, omega-automata, and logics. In *CAV 2013*, LNCS 8044. Springer, 2013.

[93] Y.-K. Tsay. Compositional verification in linear-time temporal logic. In *FoSSaCS 2000*, LNCS 1784, pages 344–358. Springer, 2000.

[94] Y.-K. Tsay, Y.-F. Chen, M.-H. Tsai, W.-C. Chan, and C.-J. Luo. GOAL extended: Towards a research tool for omega automata and temporal logic. In *TACAS 2008*, LNCS 4963, pages 346–350, 2008.

[95] Y.-K. Tsay, Y.-F. Chen, M.-H. Tsai, K.-N. Wu, and W.-C. Chan. GOAL: A graphical tool for manipulating Büchi automata and temporal formulae. In *TACAS 2007*, LNCS 4424, pages 466–471, 2007.

[96] Y.-K. Tsay, M.-H. Tsai, J.-S. Chang, and Y.-W. Chang. Büchi Store: An open repository of Büchi automata. In *TACAS 2011*, LNCS 6605, pages 262–266. Springer, 2011.

[97] Y.-K. Tsay, M.-H. Tsai, J.-S. Chang, Y.-W. Chang, and C.-S. Liu. Büchi Store: An open repository of ω-automata. *STTT*, 15(2):109–123, 2013.

[98] M. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, LNCS 1043, pages 238–266. Springer, 1996.

[99] M. Vardi. Automata-theoretic model checking revisited. In *VMCAI 2007*, LNCS 4349, pages 137–150. Springer, 2007.

[100] M. Vardi. The Büchi complementation saga. In *STACS 2007*, LNCS 4393, pages 12–22. Springer, 2007.

[101] M. Vardi and T. Wilke. Automata: from logics to algorithms. In *Logic and Automata: History and Perspective*, TLG 2, pages 629–736. Amsterdam University Press, 2007.

[102] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS 1986*, pages 322–331, Cambridge, 1986.

[103] Q. Yan. Lower bounds for complementation of omega-automata via the full automata technique. *LMCS*, 4(1), 2008.

[104] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *TCS*, 200(1-2):135–183, 1998.