DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Making Trap Abstraction Practical

Maximilian Kamps

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Making Trap Abstraction Practical

Fallen Abstraktionen praktisch machen

Author:	Maximilian Kamps
Supervisor:	Prof. Francisco Javier Esparza Estaun
Advisor:	M.Sc. Christoph Welzel-Mohr
Submission Date:	15.2.2024

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.2.2024 Maximilian Kamps

Abstract

Regular model checking is a technique for automatically verifying infinite-state systems using regular transition systems. Put simply, regular model checking tests for a system if an undesired state is reachable from a set of initial states through system transitions. Many different approaches have been developed to solve the problem; the survey [1] gives an overview of some of them. This thesis aims to explore the practical possibilities of one such approach. The method in question was first introduced in [2] and reasons over regular transitions systems using *inductive statements*. Inductive statements are statements that, if satisfied by a configuration c (a state of the system), are also satisfied by all reachable configuration c' from c via the regular transition system. These statements allow for an approximation of reachability between configurations. We focus on the contribution of [3], which provides a *PSPACE-complete* verification algorithm based on inductive statements. Additionally, [3] has implemented the algorithm under the name *oneshot* in the tool [4, dodo].

First, we present how [4, dodo] has adapted the PSPACE-complete algorithm in practice. Moreover, we investigate different experimental algorithmic techniques to improve the run time of *oneshot*. Additionally, we provide an enhanced re-implementation in [5, pytraps], including implementations of the experimental techniques. Lastly, we compare the two tools to evaluate the effectiveness of the optimizations presented in this work.

Contents

1 Introduction

Regular model checking can be applied to different classes of infinite-state systems. In this thesis, we consider only one class of infinite-state systems, namely *Parameterized systems*. Parameterized systems consist of an arbitrary amount of agents that communicate with each other or compete for a shared resource. A prime example of such a system is a cloud-based application, where many servers share a resource that can only be accessed by one participant at a time. Ensuring mutual exclusion for the resource between servers requires strict descriptions of how agents can transition into requesting, accessing, and freeing the resource. A predefined protocol captures this rule set. The question in protocols of this kind is if they consistently satisfy the property, which they should ensure. Established methods like unit testing and fuzzing can never achieve absolute certainty; instead, they can only give a negative counterexample. In systems where a hundred percent confidence (or formal verification) is required, regular model checking (if applicable) is a suitable powerful technique.

The technique relies on concepts of automata theory by encoding parameterized systems in regular languages and finite automata. Using these encodings, properties can be verified algorithmically. The approach discussed in this work captures all possible transitions (changes in the state of the participating agents) in the system. With the transition behavior, it can formally be proven that an undesired configuration (agents are in forbidden states) can never be reached *or* that there is a path of consecutive transitions of the systems that lead to an undesired configuration. Consequently, the core problem lies in efficiently (both time and memory-wise) capturing possible transitions of the system.

Research in the field has allowed for several verification methods for different class problems. [1] mentions the examples of *"systems with unbounded FIFO channel, systems with stacks, and systems with counters"*. Ergo, the promising results of regular model checking motivate continued research to find more efficient and widely applicable methods for verifying parameterized systems. In this work, we try to contribute to the research domain by searching for refinements of the method presented in [3].

2 Preliminaries

2.1 Regular languages and automata

This section's purpose is to summarize the theoretical background already introduced in [3]. This section includes standard notation (from the book [6]) of regular languages, finite automata, and transducers.

Regular Languages

A regular expression is an abstract grammar that recognizes words over a finite alphabet. The set of all (potentially infinite) words is called a regular language. An explanation of the notation of regular languages can be found in chapters 1.1 and 1.2 in [6].

Finite automata

A finite automaton recognizes a regular language. Finite automatons can be either deterministic or non-deterministic. Chapter 1.4 of [6] provides algorithms to translate regular expressions to finite automatons and vice versa.

Definition 2.1.1| Deterministic Finite Automata (DFA).

A DFA is a quintuple $\mathcal{A} = \langle \mathcal{Q}, q_0, \Sigma, \delta, \mathcal{F} \rangle$. The elements of $\mathcal{A}$ correspond to the finite state space $\mathcal{Q}$, the single initial state $q_0 \in \mathcal{Q}$, the alphabet Σ (a set of letters), the transition function $\delta : \mathcal{Q} \times \Sigma \to \mathcal{Q}$ and the set of accepting states $\mathcal{F} \subseteq \mathcal{Q}$. The DFA recognizes a word $w \in \Sigma^*$ iff for w exists a path of transitions from δ between states in $\mathcal{Q}$, starting in the initial state q_0 and ending in a final state $f \in \mathcal{F}$. More formally the DFA $\mathcal{A}$ accepts a word $w = a_1 a_2 ... a_n \in \Sigma^*$ iff:

- $p_0 = q_0$ and
- $p_{i+1} \in \delta(p_i, a_{i+1})$, for $i = 0, ..., n-1$ and
- $p_n \in \mathcal{F}$

Definition 2.1.2| Non-Deterministic Finite Automata (NFA).

A NFA is a quintuple $\mathcal{A} = \langle \mathcal{Q}, \mathcal{Q}_0, \Sigma, \delta, \mathcal{F} \rangle$. The elements of $\mathcal{A}$ correspond to the finite state space $\mathcal{Q}$, the set of initial states $\mathcal{Q}_0 \subseteq \mathcal{Q}$, the alphabet Σ (a set of letters), the transition function $\delta : \mathcal{Q} \times \Sigma \to 2^{\mathcal{Q}}$ and the set of accepting states $\mathcal{F} \subseteq \mathcal{Q}$. A NFA recognizes a word $w \in \Sigma^*$ iff for w exists a path of transitions from Δ between states in $\mathcal{Q}$, starting in an initial state $q_0 \in \mathcal{Q}_0$ and ending in a final state $f \in \mathcal{F}$. More formally the NFA $\mathcal{A}$ accepts a word $w = a_1 a_2 ... a_n \in \Sigma^*$ iff:

- $p_0 \in Q_0$ and
- $p_{i+1} \in \Delta(p_i, a_{i+1})$, for $i = 0, ..., n-1$ and
- $p_n \in \mathcal{F}$

Additionally, we denote the language accepted by DFAs $\mathcal{A}$ and NFAs $\mathcal{A}$ by $\mathcal{L}(\mathcal{A})$.

2.2 Regular transition systems

This work uses the same standard notation for *regular transition systems (RTS)* used in [3] that credits the surveys [1] and [7]. Moreover, new notation (in the field of RTS) introduced in this chapter mainly stems from [3]. *Regular model checking (RMC)* deals with the verification of systems $\mathcal{S}$ that are parameterized by some value n. The parameter n represents the number of agents in the system. RMC requires that agents only be in a finite number of states Σ. *Configurations* of the system $\mathcal{S}$ can be encoded as words over Σ^n. These words correspond to the current state of every agent; e.g., the word's first letter represents the state of the first agent, the second letter represents the state of the second agent, and so on.
RMC tries to reason over said systems to deduce if a word from a set of *undesirable* configurations can be reached. For example, consider the token passing protocol, where n agents pass around precisely one token. An agent either has the token denoted by state t or does not have the token represented by state n. The system allows all configurations of the form $n^*\ t\ n^*$ (with length n). For instance, RMC now wants to verify that *undesirable* configurations n^* (where there is no token) cannot be reached from configurations $n^*\ t\ n^*$ via the token-passing protocol.

The changing of states by agents in the systems $\mathcal{S}$ can be captured by a regular language $N \subseteq \Sigma^n \times \Sigma^n$. Words from this language correspond to a configuration $v_1 ... v_n$ transitioning in to the configuration $u_1 ... u_n$ (with $\langle v_1, u_1 \rangle ... \langle v_n, u_n \rangle \in N$). Transition languages N are recognized by *transducers*.

Definition 2.2.1| Transducer.

A Σ-Γ-transducer $\mathcal{T}$ is an NFA $\langle \mathcal{Q}, \mathcal{Q}_0, \Sigma \times \Gamma, \Delta, \mathcal{F} \rangle$. We write:

$$[\![\mathcal{T}]\!] = \left\{ \langle v_1 \dots v_n, u_1 \dots u_n \rangle \in \bigcup_{n \geq o} \Sigma^n \times \Gamma^n \mid \langle v_1, u_1 \rangle \dots \langle v_n, u_n \rangle \in \mathcal{L}(\mathcal{T}) \right\}$$

The relation $[\![\mathcal{T}]\!]$ captures which words are accepted by $\mathcal{T}$. Note, that $[\![\mathcal{T}]\!]$ only contains words $\langle u, v \rangle \in \Sigma$-$\Gamma$ where u and v have the same length.
We also introduce the useful notations:

$$\text{For } v \in \Sigma^* : \text{target}_{\mathcal{T}}(v) = \{u \in \Gamma^* | \langle v, u \rangle \in [\![\mathcal{T}]\!]\}$$
$$\text{target}_{\mathcal{T}}(\mathcal{V}) = \bigcup_{v \in \mathcal{V}} \text{target}_{\mathcal{T}}(v)$$

Moreover, pairs $\langle \sigma_1, \sigma_2 \rangle \in \Sigma \times \Gamma$ in transducers are represented as $\begin{bmatrix} \sigma_1 \\ \sigma_2 \end{bmatrix}$ for readability. Using the concept of transducers we can give a formal definition of *regular transition systems*.

Definition 2.2.2| Regular transition system (RTS).

A *regular transition system* (RTS) is a triple $\mathcal{R} = \langle \Sigma, \mathcal{I}, \mathcal{T} \rangle$ where Σ is a finite alphabet while $\mathcal{I}$ is an NFA with alphabet Σ and $\mathcal{T}$ is Σ-Σ-transducer.
With $\rightsquigarrow_{\mathcal{T}}$ we denote the relation $[\![\mathcal{T}]\!]$ and call a pair $\langle v, u \rangle \in \rightsquigarrow_{\mathcal{T}}$ (which we also write $v \rightsquigarrow_{\mathcal{T}} u$) a transition of $\mathcal{R}$. Moreover, let $\rightsquigarrow_{\mathcal{T}}^*$ denote the reflexive transitive closure of $\rightsquigarrow_{\mathcal{T}}$.
We consider $w \in \Sigma$ *reachable* in $\mathcal{R}$ if there exists $u \in \mathcal{L}(\mathcal{I})$ with $u \rightsquigarrow_{\mathcal{T}}^* w$. Let $\mathit{reach}(\mathcal{R}) \subseteq \Sigma^*$ denote all reachable configurations.

In the following, we give an example of a *regular transition system*. We will use the exemplified RTS as a running example throughout this work.

Example 2.2.1| Voting token passing as RTS.

In this parameterized system, we consider a modified version of the token-passing protocol, where the agents are ordered in a linear array.
The agents of this system represent vote-counting machines for different voting regions after an election. These machines want to count the votes for their region. After completing their task, they want to enter their count into a shared system, representing the critical section of this protocol. The system can be accessed by only one machine at a time.
At first, all but one agent are in the **idle state (i)** beckoning that they are still counting votes. The one agent not in the idle state starts in the **state token (t)**. Formally, having the token means that the district has counted all its votes and has access to the critical

section. After counting his votes, an agent transitions from the idle state to the **ready state (r)**, signaling that it wants to access the token to enter its votes. The agent (in possession of the token) can pass on the token to any agent in state r. After passing on the token, the agents transition to the state **marked (m)**. After all regions are in the state marked, the RTS cannot execute any more transitions. In other words, all agents have successfully entered their votes.

Encoding the system in regular languages:

The initial configurations are defined by the language: i^*ti^*.

The transitions can be captured by the union of two languages. To simplify the explanation of the system, we introduce the placeholders Z, representing that an agent stays in the idle, ready, or marked state after one step or transitions from idle to ready.

Formally, we set:

$$Z = \left(\begin{bmatrix} i \\ i \end{bmatrix} | \begin{bmatrix} i \\ r \end{bmatrix} | \begin{bmatrix} r \\ r \end{bmatrix} | \begin{bmatrix} m \\ m \end{bmatrix} \right)$$

The first language encodes that at least one agent is either in the state r or i. Intuitively, this means that the counting of votes is still ongoing.

$$Z^* \left(\begin{bmatrix} t \\ t \end{bmatrix} | \left(\begin{bmatrix} r \\ t \end{bmatrix} Z^* \begin{bmatrix} t \\ m \end{bmatrix} \right) | \left(\begin{bmatrix} t \\ m \end{bmatrix} Z^* \begin{bmatrix} r \\ t \end{bmatrix} \right) \right) Z^*$$

The second language encodes that all but one agent is in the state m. The one agent is in the state t. In other words, the system is one step away from "completion".

$$\begin{bmatrix} m \\ m \end{bmatrix}^* \begin{bmatrix} t \\ m \end{bmatrix} \begin{bmatrix} m \\ m \end{bmatrix}^*$$

Figure 2.2.1| $\mathcal{I}$ of the voting token passing protocol.

Illustration of the NFA recognizing the initial states of the voting token passing protocol of Example 2.2.1.

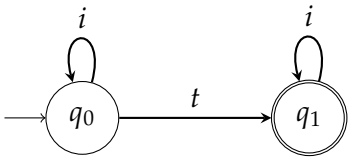
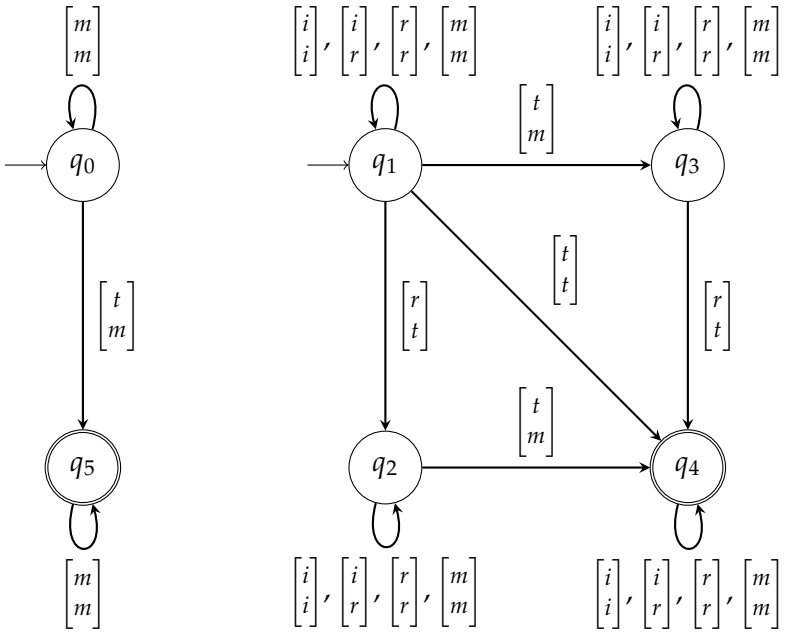


Figure 2.2.2| $\mathcal{T}$ of the voting token passing protocol.

Illustration of the transducer recognizing the transitions of the voting token passing protocol of Example 2.2.1.



3 Approximating reachability

In this chapter, we will introduce how reachability between configurations in a regular transition system can be approximated. This requires the notion of *inductive statements* presented in [3].

3.1 Statements

Inductive statements are a framework to reason over the reachable set of any RTS. To arrive at the concept of inductive statements, we must first introduce what a statement is.

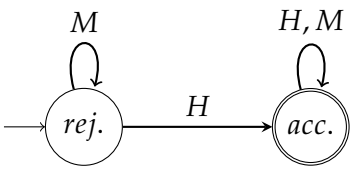
A statement consists of an *encoding* and an *interpretation*. The *encoding* represents a property for some configuration as a word over the same alphabet Γ. For instance, consider the alphabet $\Gamma = \{a, b, c\}$, the configuration $a\ c\ b\ a \in \Gamma^*$ and the encoding $\{b\}\ \{a, b\}\ \{a, b, c\}\ \{a, c\} \in (2^\Gamma)^*$. The encoding assigns a set of states from Γ to every position of a configuration. Subsequently, for the first position, the *encoding* assigns $\{b\}$; for the second position, the *encoding* assigns $\{a, b\}$ and so on. The encoding is evaluated for a configuration by using an *interpretation*. For instance, observe the interpretation:

"There is at least one position where the agent is in the state assigned by the encoding."

Applying this interpretation to the configuration $a\ c\ b\ a$ and the encoding $\{b\}\ \{a, b\}\ \{a, b, c\}\ \{a, c\}$ yields true, because the agent at position 3 is the in the state b, which it is in the set $\{a, b, c\}$. Note that agents 1 and 2 are not in the state assigned to them by the encoding; compare $a \notin \{b\}$ and $c \notin \{a, b\}$. We call the above-introduced interpretation the *trap-interpretation*. We can evaluate a configuration v and an encoding $\mathcal{I}$ for the *trap-interpretation* with the transducers $\mathcal{V}_{trap}$. More formally:

Definition 3.1.1| Acceptance of the trap-interpretation.

For a pair $\langle v, I \rangle$ holds that $\langle v, I \rangle \in [\![\mathcal{V}_{trap}]\!]$ iff the pair is accepted by the Σ-2^Σ-transducer $\mathcal{V}_{trap}$:



H denotes all pairs $\langle v, I \rangle \in \Sigma \times 2^\Sigma$, such that $v \in I$ and M denotes all pairs $\langle v, I \rangle \in$

$\Sigma \times 2^{\Sigma}$ such that $v \notin I$.

We introduce the notation $u \models_{\mathcal{V}_{trap}} I$ which is equivalent to $\langle v, I \rangle \in [\![\mathcal{V}_{trap}]\!]$.

3.2 Making statements inductive

First, we give a formal definition of inductive statements. Afterward, we explain the notion of a statement being inductive. The following introduces the set of inductive statements for a RTS $\mathcal{R}$. Please note that we restrict the interpretations of statements to $\mathcal{V}_{Trap}$. Consequently, the subsequent set captures all *encodings*, evaluated with the interpretation $\mathcal{V}_{Trap}$, which are inductive for $\mathcal{R}$.

Definition 3.2.1| Inductive statements.

For the interpretation $\mathcal{V}_{trap}$ and the RTS $\mathcal{R} = \langle \Sigma, \mathcal{I}, \mathcal{T} \rangle$ let

$$\begin{aligned} Inductive_{\mathcal{V}_{trap}}(\mathcal{R}) &= \{I \in 2^{\Sigma^*} \mid \forall u \rightsquigarrow_{\mathcal{T}} v \,.\, \text{if } \langle u, I \rangle \in [\![\mathcal{V}_{trap}]\!] \text{ then } \langle v, I \rangle \in [\![\mathcal{V}_{trap}]\!]\} \\ &= \{I \in 2^{\Sigma^*} \mid \forall u \rightsquigarrow_{\mathcal{T}} v \,.\, \text{if } u \models_{\mathcal{V}_{trap}} I \text{ then } v \models_{\mathcal{V}_{trap}} I\} \end{aligned}$$

The paper [3] proves that the set of all inductive statements is a regular language and, as such, algorithmically accessible. An inductive statement $I \in Inductive_{\mathcal{V}_{trap}}(\mathcal{R})$ is a statement that if satisfied by a configuration u ($u \models_{\mathcal{V}_{trap}} I$), is also satisfied by all configurations v that is potentially reachable from u ($v \models_{\mathcal{V}_{trap}} I$ for all $u \rightsquigarrow_{\mathcal{T}} v$). Corollary I holds for every configuration between u and v. We call this relationship for u and v *potential reachability* or simply that v is *potentially reachable* from u. The trap interpretation derives its name in the context of inductive statements. Remember, a configuration is satisfied by $\mathcal{V}_{trap}$ if it matches the encoding of the statement in at least one position. Consequently, *once a configuration has some value in the inductive statement, it cannot remove all its values again from it – it gets "trapped"* for all subsequent potentially reachable configurations.
With the use of inductive statements, we can try to approximate the relation $\rightsquigarrow^*_{\mathcal{T}}$ by considering some configuration v reachable from configuration u if v satisfies the same inductive statements as u. Note that the set of configurations that satisfy the same inductive statements as the configuration for which we try to express potential reachability is an over-approximation of the actually reachable set of configurations in the RTS. In other words, if $u \models_{\mathcal{V}_{trap}} I$ then $v \models_{\mathcal{V}_{trap}} I$ for all $u \rightsquigarrow^*_{\mathcal{T}} v$, **but not necessarily the contrary** if $u \models_{\mathcal{V}_{trap}} I$ then $u \rightsquigarrow^*_{\mathcal{T}} v$ for all $v \models_{\mathcal{V}_{trap}} I$. In the following, we give an example of how reachability can be approximated for the voting token passing protocol from 2.2.1.

Example 3.2.1| Approximating reachability via inductive statements.

Consider the following inductive statements for the RTS $\mathcal{R}$ of the voting token passing protocol; additionally, recall the transition transducer $\mathcal{T}$ (depicted in 2.2.2) for the

protocol. The inductive statements approximate the transitions of $\mathcal{R}$. We say that a configuration is *potentially reachable* from another configuration if the latter satisfies all the inductive statements satisfied by the former.

1. **Statement** $\varnothing^* \{r, t, m\} \varnothing^* \subseteq Inductive_{\mathcal{V}_{trap}}(\mathcal{R})$**.** This statement corresponds to: "The agent cannot return to state i from state r. For example, consider the inductive statements for state $r\ r\ i\ i$:

$$\{t, m\, r\} \varnothing \varnothing \varnothing$$
$$\varnothing \{t, m, r\} \varnothing \varnothing$$

   $t\ r\ i\ i$ is potentially reachable from $r\ r\ i\ i$, while $i\ r\ i\ i$ is not.
   The statement is inductive, because there are no transitions $\begin{bmatrix} r \\ i \end{bmatrix}, \begin{bmatrix} t \\ i \end{bmatrix}$ or $\begin{bmatrix} m \\ i \end{bmatrix}$ in $\mathcal{T}$. In other words, once an agent is in state r, t, or m, the RTS contains no transition with which it could return to state i. It is evident, that if u satisfies some statements from $\varnothing^* \{r, t, m\} \varnothing^*$ and $u \rightsquigarrow_{\mathcal{T}}^* v$ holds, then v also satisfies the same statements as u from $\varnothing^* \{r, t, m\} \varnothing^*$. Therefore, we can conclude that $\varnothing^* \{r, t, m\} \varnothing^* \subseteq Inductive_{\mathcal{V}_{trap}}(\mathcal{R})$.

2. **Statement** $\varnothing^* \{t, m\} \varnothing^* \subseteq Inductive_{\mathcal{V}_{trap}}(\mathcal{R})$**.** This statement corresponds to: "The agent cannot return to state r or i from state t. For example, consider the inductive statements for state $t\ t\ r\ i$:

$$\{t, m\} \varnothing \varnothing \varnothing$$
$$\varnothing \{t, m\} \varnothing \varnothing$$

   $m\ t\ r\ i$ is potentially reachable from $t\ t\ r\ i$, while $r\ t\ r\ i$ is not.
   The statement is inductive due to a similar chain of reasoning as applied in 1. (There are no transitions $\begin{bmatrix} t \\ i \end{bmatrix}, \begin{bmatrix} m \\ i \end{bmatrix}, \begin{bmatrix} t \\ r \end{bmatrix}$ or $\begin{bmatrix} m \\ r \end{bmatrix}$ in $\mathcal{T}$).

3. **Statement** $\varnothing^* \{m\} \varnothing^* \subseteq Inductive_{\mathcal{V}_{trap}}(\mathcal{R})$**.** This statement corresponds to: "The agent stays in state m." For example, consider the inductive statements for state $m\ m\ t\ i$:

$$\{m\} \varnothing \varnothing \varnothing$$
$$\varnothing \{m\} \varnothing \varnothing$$

   $m\ m\ t\ r$ is potentially reachable from $m\ m\ t\ i$, while $m\ t\ t\ i$ is not.
   The statement is inductive due to a similar chain of reasoning as applied in 1. (There are no transitions $\begin{bmatrix} m \\ i \end{bmatrix}, \begin{bmatrix} m \\ r \end{bmatrix}$ or $\begin{bmatrix} m \\ t \end{bmatrix}$ in $\mathcal{T}$).

4. **Statement** $\varnothing^* \{m, i, r\} \varnothing^* \{m, i, r\} \varnothing^* \subseteq Inductive_{\mathcal{V}_{trap}}(\mathcal{R})$. This statement corresponds to: "The amount of tokens does not increase." For example, consider the inductive statements for state $i\ t\ i\ i$:

$$\{m,i,r\}\ \{m,i,r\}\ \varnothing\ \varnothing$$
$$\{m,i,r\}\ \varnothing\ \{m,i,r\}\ \varnothing$$
$$\{m,i,r\}\ \varnothing\ \varnothing\ \{m,i,r\}$$
$$\varnothing\ \{m,i,r\}\ \{m,i,r\}\ \varnothing$$
$$\varnothing\ \{m,i,r\}\ \varnothing\ \{m,i,r\}$$
$$\varnothing\ \varnothing\ \{m,i,r\}\ \{m,i,r\}$$

$i\ i\ i\ t$ is potentially reachable from $i\ t\ i\ i$, while $i\ t\ t\ i$ is not.
[3] has already proven that this statement is inductive for token-passing protocols: "To this end, observe that each word of this language corresponds to the statement "for two given indices i,j either the i-th letter is in $\{i,r,m\}$ or the j-th letter is in $\{i,r,m\}$" where the indices correspond to the two positions where the letters are in $\{i,r,m\}$; e.g., $\{i,r,m\}\ \varnothing\ \varnothing\ \{i,r,m\}\ \varnothing$ applies to words of the length five and ensures that either the first or fourth letter is $\{i,r,m\}$. Note that all transitions of this example originate in a configuration where exactly one letter is t and end up in a configuration where exactly one letter is t. Consequently, the origin and the target of every transition satisfy all of these statements. Therefore, we can conclude that $\varnothing^*\ \{m,i,r\}\ \varnothing^*\ \{m,i,r\}\ \varnothing^* \subseteq Inductive_{\mathcal{V}_{trap}}(\mathcal{R})$."

5. **Statement** $\{t\}^*\ \{t,m\}\ \{t\}^* \subseteq Inductive_{\mathcal{V}_{trap}}(\mathcal{R})$. This statement corresponds to: "Either all agents are marked, or there is at least one agent with the token." For example, consider the inductive statements from $\{t\}^*\ \{t,m\}\ \{t\}^*$ for state $t\ r\ r\ r$:

$$\{t,m\}\ \{t\}\ \{t\}\ \{t\}$$
$$\{t\}\ \{t,m\}\ \{t\}\ \{t\}$$
$$\{t\}\ \{t\}\ \{t,m\}\ \{t\}$$
$$\{t\}\ \{t\}\ \{t\}\ \{t,m\}$$

$m\ m\ m\ m$ is potentially reachable from $t\ r\ r\ r$, while $r\ r\ r\ r$ is not.
We must analyze when the token can disappear in the systems to prove that the statement is inductive. Take note of the states q_0 and q_5 in $\mathcal{T}$. The token disappears (through transitions from $\mathcal{T}$) from the configuration only if all previous and subsequent agents are in state m. The set of statements from $\{t\}^*\ \{t,m\}\ \{t\}^*$ for a configuration enforce exactly this restriction. The statements are either satisfied if an agent is in state t (at any position) or all agents are in state m. It is evident, that if u satisfies some statements from $\{t\}^*\ \{t,m\}\ \{t\}^*$ and $u \rightsquigarrow_T^* v$ holds, then v also satisfies the same statements as u from $\{t\}^*\ \{t,m\}\ \{t\}^*$. Therefore, we can conclude that $\{t\}^*\ \{t,m\}\ \{t\}^* \subseteq Inductive_{\mathcal{V}_{trap}}(\mathcal{R})$.

3.3 From reachability to approximating reachability

The motivation behind inductive statements is to solve the *reachability problem*. The problem entails, can for a given RTS $\mathcal{R}$ a set of undesirable configurations (words) $\mathcal{B}$ be reached? More formally:

Problem 1. The reachability problem.

Given: *RTS* $\mathcal{R}$ *and NFA* $\mathcal{B}$
Compute: $reach(\mathcal{R}) \cap \mathcal{L}(\mathcal{B}) = \varnothing$?

[8] establishes that this problem is generally undecidable.

Nevertheless, inductive statements allow for a semi-procedure. [3] rephrases the problem as follows:

"Do an initial configuration v and an undesired configuration u exist,
such that u satisfies all inductive statements that v satisfies?"

The resulting new problem is called "the approximated reachability problem." To formalize it, we first need to introduce the following notations:

- We introduce the *chained transducer* $\mathcal{C}$ of two transducer $\mathcal{F}$ and $\mathcal{S}$ which accepts the language:
$$[\![\mathcal{C}]\!] = [\![\mathcal{F}]\!] \circ [\![\mathcal{S}]\!] = \left\{ \langle u, w\rangle \in \bigcup_{n\geq o} \Sigma^n \times \Upsilon^n \mid \exists \langle u, v\rangle \in [\![\mathcal{F}]\!] \text{ and } \exists \langle v, w\rangle \in [\![\mathcal{S}]\!] \right\}.$$
Intuitively, $[\![\mathcal{C}]\!]$ represents the language of words $\langle u, w\rangle$, where the first component of $\langle u, v\rangle \in [\![\mathcal{F}]\!]$ is "combined" with the second component of $\langle v, w\rangle \in [\![\mathcal{S}]\!]$ by the shared component $v \in \Sigma$.
- We introduce for a DFA $\mathcal{A}$ the transducer $Id(\mathcal{A})$ which accepts the language:
$$[\![Id(\mathcal{A})]\!] = \{\langle u, u\rangle : u \in \mathcal{L}(\mathcal{A})\}.$$
- Recall the concept of potential reachability via inductive statements presented in 3.2. We introduce the relation $\Rightarrow_{\mathcal{V}_{trap}}$, which captures potential reachability for a *RTS* $\mathcal{R}$ for all configuration pairs u and v.

Problem 2. The approximated reachability problem.

Given: *RTS* $\mathcal{R} = \langle \Sigma, \mathcal{I}, \mathcal{T}\rangle$ *and NFA*
Compute: $[\![Id(\mathcal{I})]\!] \circ \Rightarrow_{\mathcal{V}_{trap}} \circ [\![Id(\mathcal{B})]\!] = \varnothing$?

Section 2.5 of [3] proves that the problem can be solved in *PSPACE*.

3.3.1 Inductive statements for the approximation reachability problem

[3, Theorem 2.1] proofs, that for the interpretation $\mathcal{V}_{trap}$ and RTS $\mathcal{R} = \langle \Sigma, \mathcal{I}, \mathcal{T} \rangle$ a regular over-approximation of $reach(\mathcal{R})$ can be computed. The following example illustrates how $reach(\mathcal{R})$ can be approximated for the voting token passing protocol.

Example 3.3.1| Computing an over-approximation

Recall the formalization of the token voting algorithm in 2.2.1 and its corresponding inductive statements in 3.2.1. We argue now, that the over-approximation that arises from $\text{target}_{\mathcal{P}}(\mathcal{R})$ (where $\mathcal{P}$ is the transducer that accepts $[\![Id(\mathcal{I})]\!] \circ \Rightarrow_{\mathcal{V}\,trap}$) coincides with the set of actually reachable configurations $((i|r|m)^*\ t\ (i|r|m)^*) \mid m^*$ (one t and arbitrary distribution of i, r, m **or** just m). This becomes evident when performing a case distinction over the inductive statements from 3.2.1:

- **Statements 1** and **2** enforce that agents can not change their state back from r to i and from t to r. Additionally, **statement 3** ensures that an agent is trapped in state m after assuming it.
- **Statement 4** ensures that the number of tokens does not increase. Recall the initial configuration of the voting passing protocol. The system is initialized with exactly one token.
- **Statement 5** requires configurations to have at least one token while not all agents are marked. One can deduce that together, statement **4** and **5** enforce that there is exactly one token until all the agents are marked.

Using the over-approximation, one can infer that a configuration u is only reachable from v via the aforementioned inductive statements if it adheres to the transitions defined by the RTS $\mathcal{R}$. Consequently, the over-approximation equates (for this example) to *reach($\mathcal{R}$)*.

4 Oneshot

The purpose of this section is to give an intuition of how the relation $\Rightarrow_{trap}$ from **Problem** 2 can be captured by a transducer which we call $\mathcal{G}_{Trap}$. In [3, **Traps in PSpace**], the notion of a *separator* $S \in Inductive_{\mathcal{V}_{trap}}(\mathcal{R})$ is introduced. S is an inductive statement (from 3.2.1) that expresses potential reachability between two configurations v and u. It is important to note that S is uniquely defined by the configuration u. We say that u is potentially reachable from v, iff $u \models_{\mathcal{V}_{trap}} S$ and $v \not\models_{\mathcal{V}_{trap}} S$.
Moreover, the separator S is algorithmically accessible, allowing the formulation of an automatic procedure to compute potential reachability between configuration pairs v and u. [3] names the procedure the *step game*. Motivated by these observations, we apply the step game to all pairs v and u to encapsulate potential reachability for the entire RTS. We call the transducer that recognizes this language $\mathcal{G}_{trap}$. The step game is crucial for constructing $\mathcal{G}_{trap}$; thus, we will discuss it in detail. Furthermore, we will explain the algorithm *oneshot* (first implemented in [4, dodo]), which automatically solves **Problem 2** by constructing $\mathcal{G}_{trap}$ for the relation $\Rightarrow_{trap}$. For the *step game* and *oneshot*, we present optimizations that can likely reduce the complexity of the problem in practice.

4.1 The transducer $\mathcal{G}_{trap}$

We give a formal definition of the *inductive transducer* $\mathcal{G}_{trap}$:

Definition 4.1.1| Inductive transducer.

Let $\mathcal{Q}_{\mathcal{T}}$ denote the states of $\mathcal{T}$ (the transition transducer of a RTS $\mathcal{R} = \langle \Sigma, \mathcal{I}, \mathcal{T} \rangle$), and let $\mathcal{Q}_0 \subseteq \mathcal{Q}_{\mathcal{T}}$ and $\mathcal{F} \subseteq \mathcal{Q}_{\mathcal{T}}$ the initial and accepting states respectively. Let $bases(\mathcal{T}) = \{c \in \mathcal{Q}_{\mathcal{T}}^* \mid c \text{ contains a state } q \in \mathcal{Q}_{\mathcal{T}} \text{ at most once}\}$. Intuitively, the set $bases(\mathcal{T})$ contains all ordered subsets of $\mathcal{T}$. We call the Σ-Σ transducer $\mathcal{G}_{trap} = \langle bases(\mathcal{T}), bases(\mathcal{T}) \cap \mathcal{Q}_0^*, \Sigma \times 2^{\Sigma}, \Delta, bases(\mathcal{T}) \cap \mathcal{F}^* \rangle$ with

$$\left\langle c, \begin{bmatrix} v \\ u \end{bmatrix}, d \right\rangle \in \Delta \text{ if and only if}$$

there are $c \in bases(\mathcal{T})$ and $d \in bases(\mathcal{T})$ which compute S for u and $v \notin S$

the *inductive transducer* for $\mathcal{T}$.

4.2 Step game

The transition relation Δ for the transducer $\mathcal{G}_{trap}$ can be computed by a single-player game (named the *step game*) introduced in [3]. The procedure allows us to decide if a set of transitions $\left\{ \left\langle c, \begin{bmatrix} v \\ u \end{bmatrix}, d \right\rangle \middle| v \notin S \right\}$ is in Δ by computing $\left\langle c, \begin{bmatrix} u \\ S \end{bmatrix}, d \right\rangle$ (where S is the unique separator of u). In practice, both the tools [5, pytraps] and [4, dodo] directly compute transitions $\left\langle c, \begin{bmatrix} v \\ u \end{bmatrix}, d \right\rangle$ of the Σ-Σ transducer $\mathcal{G}_{trap}$ instead of first computing $\left\langle c, \begin{bmatrix} u \\ S \end{bmatrix}, d \right\rangle$. Moreover, instead of "guessing" a successor state $d \in bases(\mathcal{T})$ and deciding if it is part of $\mathcal{G}_{trap}$, both implementations construct successors iteratively.

The new procedure fixes an origin state $c \in bases(\mathcal{T})$ and a transition $\begin{bmatrix} v \\ u \end{bmatrix}$ in the pair $\left\langle c, \begin{bmatrix} v \\ u \end{bmatrix} \right\rangle$ and lazily constructs all successor states $d \in bases(\mathcal{T})$. This way of computing transitions allows for an optimization we showcase in 4.3.1. We call the new procedure that captures how successors are constructed in practice the *lazy step game*. The procedure is lazy in the sense that the implementations (in [5, pytraps] and [4, dodo]) return successors immediately upon winning the game and only produce more successors when demanded.

Definition 4.2.1| Lazy step game.

For $\left\langle c, \begin{bmatrix} u \\ v \end{bmatrix} \right\rangle$ where $c = b_1^1 ... b_n^1 \in bases(\mathcal{T})$ and $u, v \in \Sigma$ we consider the following game:
The current state of the game is represented by a triple $\langle c_p, S, d \rangle$ where

- $c_p = b_1^1 ... b_l^1$ is a prefix of c (with $l \leq n$)
- $S \subseteq 2^\Sigma$
- $d = b_1^2 ... b_m^2 \in bases(\mathcal{T})$ with $m \geq 1$

In every turn, the player can play any step $\left\langle q, \begin{bmatrix} x \\ y \end{bmatrix}, p \right\rangle$ from $\mathcal{T}$. The player **loses immediately** if

- $y \in S$
- q does not occur in c_p or
- c_p is a proper prefix of c and $q \neq b^1_{|c_p|+1}$ with $b^1_{|c_p|+1} \in c$

Otherwise $\left\langle q, \begin{bmatrix} x \\ y \end{bmatrix}, p \right\rangle$ **progresses** the game state to $\langle c'_p, S', d' \rangle$ such that:

- $c'_p = c_p$ if $q \in c_p$ and otherwise, $c_p = c_p \cup p^1_{|c_p|+1}$
- $d' = d$ if $p \in d$ and otherwise, $d' = d \cup p^2_{|d|+1}$
- $S' = S \setminus \{x\}$

Initially, the state is $\langle \varnothing, \Sigma \setminus \{u\}, \varnothing \rangle$. The player wins the game if the final state becomes $\langle c_p, S \setminus \{v\}, d \rangle$ with $c_p = c$.

For a tuple $b = \left\langle c, \begin{bmatrix} u \\ v \end{bmatrix} \right\rangle$ we capture the set of winning successors by:

$$\Omega(b) = \left\{ \left\langle c, \begin{bmatrix} u \\ v \end{bmatrix}, d \right\rangle | \text{d} \in \mathit{bases}(\mathcal{T}) \text{ and } d \text{ wins the step game} \right\}$$

Intuitively, one can think of the lazy step game "reconstructing" the origin state $c \in \mathit{bases}(\mathcal{T})$ with the origin states q of transitions $t = \left\langle q, \begin{bmatrix} x \\ y \end{bmatrix}, p \right\rangle$ from $\mathcal{T}$ while trying to remove the letter v from the separator S. As a side effect, the successor $d \in \mathit{bases}(\mathcal{T})$ is constructed with the target states p of the same transitions t. We illustrate the construction of states via the lazy step game in an example:

Example 4.2.1| Computing transitions with the lazy step game.

In this example, we partially compute $\Omega(b)$ with $b = \left\langle q_1, \begin{bmatrix} m \\ t \end{bmatrix} \right\rangle$ and the initial game state $\langle \varnothing, \Sigma \setminus \{m\}, \varnothing \rangle$ for the voting token passing RTS 2.2.1. In other words, we derive some of the successors of the state q_1 in the transducer $\mathcal{G}_{trap}$ for the transition $\begin{bmatrix} m \\ t \end{bmatrix}$. Note that q_1 is a base with a single element ($q_1 \in \mathit{bases}(\mathcal{T})$). In the table below, we track the *winning transition* (transitions to successors of q_1), the *steps from* $\mathcal{T}$ (to win the step game), and the Separator S of the game state gs. For a transition to be winning, $\mathcal{T}$ has to be removed from S such that $t \notin S$.

winning transition	steps from $\mathcal{T}$	S
$\left\langle q_1, \begin{bmatrix} m \\ t \end{bmatrix}, q_3 \right\rangle$	$\left\langle q_1, \begin{bmatrix} t \\ m \end{bmatrix}, q_3 \right\rangle$	$\{i, r\}$
$\left\langle q_1, \begin{bmatrix} m \\ t \end{bmatrix}, q_3 q_4 \right\rangle$	$\left\langle q_1, \begin{bmatrix} t \\ m \end{bmatrix}, q_3 \right\rangle \left\langle q_1, \begin{bmatrix} t \\ t \end{bmatrix}, q_4 \right\rangle$	$\{i, r\}, \{i, r\}$
$\left\langle q_1, \begin{bmatrix} m \\ t \end{bmatrix}, q_3 q_1 \right\rangle$	$\left\langle q_1, \begin{bmatrix} t \\ m \end{bmatrix}, q_3 \right\rangle, \left\langle q_1, \begin{bmatrix} m \\ m \end{bmatrix}, q_1 \right\rangle$	$\{i, r\}, \{i, r\}$
$\left\langle q_1, \begin{bmatrix} m \\ t \end{bmatrix}, q_3 q_1 q_2 \right\rangle$	$\left\langle q_1, \begin{bmatrix} t \\ m \end{bmatrix}, q_3 \right\rangle, \left\langle q_1, \begin{bmatrix} m \\ m \end{bmatrix}, q_1 \right\rangle, \left\langle q_1, \begin{bmatrix} r \\ t \end{bmatrix}, q_2 \right\rangle$	$\{i, r\}, \{i, r\}, \{i\}$
$\left\langle q_1, \begin{bmatrix} m \\ t \end{bmatrix}, q_3 q_1 q_4 \right\rangle$	$\left\langle q_1, \begin{bmatrix} t \\ m \end{bmatrix}, q_3 \right\rangle, \left\langle q_1, \begin{bmatrix} m \\ m \end{bmatrix}, q_1 \right\rangle, \left\langle q_1, \begin{bmatrix} t \\ t \end{bmatrix}, q_4 \right\rangle$	$\{i, r\}, \{i, r\}, \{i, r\}$
$\left\langle q_1, \begin{bmatrix} m \\ t \end{bmatrix}, q_3 q_2 \right\rangle$	$\left\langle q_1, \begin{bmatrix} t \\ m \end{bmatrix}, q_3 \right\rangle, \left\langle q_1, \begin{bmatrix} r \\ t \end{bmatrix}, q_2 \right\rangle$	$\{i, r\}, \{i\}$
$\left\langle q_1, \begin{bmatrix} m \\ t \end{bmatrix}, q_3 q_2 q_1 \right\rangle$	$\left\langle q_1, \begin{bmatrix} t \\ m \end{bmatrix}, q_3 \right\rangle, \left\langle q_1, \begin{bmatrix} r \\ t \end{bmatrix}, q_2 \right\rangle, \left\langle q_1, \begin{bmatrix} r \\ r \end{bmatrix}, q_1 \right\rangle$	$\{i, r\}, \{i\}, \{i\}$
$\left\langle q_1, \begin{bmatrix} m \\ t \end{bmatrix}, q_3 q_2 q_1 \right\rangle$	$\left\langle q_1, \begin{bmatrix} t \\ m \end{bmatrix}, q_3 \right\rangle, \left\langle q_1, \begin{bmatrix} r \\ t \end{bmatrix}, q_2 \right\rangle, \left\langle q_1, \begin{bmatrix} i \\ r \end{bmatrix}, q_1 \right\rangle$	$\{i, r\}, \{i\}, \varnothing$
$\left\langle q_1, \begin{bmatrix} m \\ t \end{bmatrix}, q_1 q_3 \right\rangle$	$\left\langle q_1, \begin{bmatrix} m \\ m \end{bmatrix}, q_1 \right\rangle, \left\langle q_1, \begin{bmatrix} t \\ m \end{bmatrix}, q_3 \right\rangle$	$\{i, r, t\}, \{i, r\}$

4.3 Optimizing the step game

In this section, we present a mechanism to optimize the lazy step game in practice, which we have implemented in [5, pytraps].

4.3.1 Ambiguous construction of states

The construction of states by the lazy step game is ambiguous because different sequences of transducer steps can lead to the same game state. The following example illustrates this:

Example 4.3.1| Construction of ambiguous states.

Recall the voting passing RTS from 2.2.1. We will now show that the computation of the transition $\left\langle q_1q_2q_4, \begin{bmatrix} t \\ t \end{bmatrix}, q_2q_1q_4 \right\rangle$ of $\mathcal{G}_{trap}$ is ambiguous.

Let $b = \left\langle q_1q_2q_4, \begin{bmatrix} t \\ t \end{bmatrix} \right\rangle$ and the initial game state $gs = \left\langle 0, \{i, r, m\}, \varepsilon \right\rangle$. Then the lazy step game can take steps in the following order to arrive at the successor $q_2q_1q_4$:

1. $\left\langle q_1, \begin{bmatrix} r \\ t \end{bmatrix}, q_2 \right\rangle$ with $gs_1 = \left\langle q_1, \{i, m\}, q_2 \right\rangle$
2. $\left\langle q_1, \begin{bmatrix} i \\ r \end{bmatrix}, q_1 \right\rangle$ with $gs_2 = \left\langle q_1, \{m\}, q_2q_1 \right\rangle$
3. $\left\langle q_2, \begin{bmatrix} i \\ i \end{bmatrix}, q_2 \right\rangle$ or $\left\langle q_2, \begin{bmatrix} i \\ r \end{bmatrix}, q_2 \right\rangle$ or $\left\langle q_2, \begin{bmatrix} r \\ r \end{bmatrix}, q_2 \right\rangle$ with $gs_3 = \left\langle q_1q_2, \{m\}, q_2q_1 \right\rangle$
4. $\left\langle q_4, \begin{bmatrix} i \\ i \end{bmatrix}, q_4 \right\rangle$ with $gs_4 = \left\langle q_1q_2q_4, \{m\}, q_2q_1q_4 \right\rangle$

Three transitions from $\mathcal{T}$ result in the same game state in the third step. Recall the lazy step game rules introduced in 4.2.1. The three steps satisfy the conditions to not lose immediately and add q_2 to c_p, leaving the rest of the game state unchanged.

These findings allow us to exclude some step sequences that arise from gs_3.
In step three of example 4.3.1, the program [5, pytraps] construct all reachable successors for the step sequence beginning in

$$\left\langle q_1, \begin{bmatrix} r \\ t \end{bmatrix}, q_2 \right\rangle, \left\langle q_1, \begin{bmatrix} i \\ r \end{bmatrix}, q_1 \right\rangle, \left\langle q_2, \begin{bmatrix} i \\ i \end{bmatrix}, q_2 \right\rangle.$$

It follows that the sequences beginning in

$$\left\langle q_1, \begin{bmatrix} r \\ t \end{bmatrix}, q_2 \right\rangle, \left\langle q_1, \begin{bmatrix} i \\ r \end{bmatrix}, q_1 \right\rangle, \left(\left\langle q_2, \begin{bmatrix} i \\ r \end{bmatrix}, q_2 \right\rangle \text{ or } \left\langle q_2, \begin{bmatrix} r \\ r \end{bmatrix}, q_2 \right\rangle \right)$$

can be excluded from further computations because they all lead to the same game state gs_3 (and subsequently lead to the same Ω) as shown in 4.3.1. We have implemented this optimization in the tool [5, pytraps], and in practice, it yields a significant reduction of the time needed to construct $\mathcal{G}_{trap}$.

4.4 Revisiting the approximated reachability problem

In this section, we introduce the procedure to automatically solve the *approximated reachability problem* in 2. We follow the naming convention of the tool [4, dodo] and call the procedure *oneshot*. Before formalizing the algorithm, we first need to introduce some notation. *The pairing transducer* $\mathcal{X} = \mathcal{I} \times \mathcal{B}$ accepts the cross product of the languages accepted by the NFA $\mathcal{A}$ and NFA $\mathcal{B}$:

$$[\![\mathcal{X}]\!] = \{\langle u, v\rangle \mid u \in [\![A]\!], v \in [\![B]\!]\}$$

Solving the abstract reachability problem with oneshot

Oneshot computes $[\![\mathcal{I} \times \mathcal{B}]\!] \cap [\![\mathcal{G}_{trap}]\!] \stackrel{!}{=} \varnothing$, which is equivalent to the problem definition $[\![Id(\mathcal{I})]\!] \circ [\![\mathcal{G}_{trap}]\!] \circ [\![Id(\mathcal{B})]\!] \stackrel{!}{=} \varnothing$ in **Problem 2**. In other words, instead of chaining the transducers together to probe if some configuration of $\mathcal{B}$ is potentially reachable via $\mathcal{G}_{trap}$ from $\mathcal{I}$, we first construct all pairs of configurations of $\mathcal{I}$ and $\mathcal{B}$ and check if $\mathcal{G}_{trap}$ recognizes such a pair. We formalize the intersection of the three automata in the following transducer:

Definition 4.4.1| Intersection transducer.

Let $\mathcal{X} = \langle Q_{\mathcal{X}}, Q_0^{\mathcal{X}}, \Sigma \times \Sigma, \Delta_{\mathcal{X}}, \mathcal{F}_{\mathcal{X}}\rangle$ denote the resulting $\Sigma \times \Sigma$-transducer from the product construction of the initial word NFA $\mathcal{I}$ and the bad word NFA $\mathcal{B}$.
Let $\mathcal{G} = \langle bases(Q_{\mathcal{T}}^*), bases(\mathcal{T}) \cap Q_0^*, \Sigma \times 2^{\Sigma}, \Delta, bases(\mathcal{T}) \cap \mathcal{F}^*\rangle$ denote the reduced inductive transducer $\mathcal{G}_{trap}$ for $\mathcal{T}$. We call the Σ-Σ-transducer $\mathcal{Z} = \langle (Q_{\mathcal{X}} \times Q_{\mathcal{G}}), (Q_0^{\mathcal{X}} \times Q_0^{\mathcal{G}}), \Sigma \times \Sigma, \Delta_{\mathcal{Z}}, (\mathcal{F}_{\mathcal{X}} \times \mathcal{F}_{\mathcal{G}})\rangle$ with

$$\left\langle (a_{\mathcal{X}}, c_{\mathcal{G}}), \begin{bmatrix} x \\ y \end{bmatrix}, (b_{\mathcal{X}}, d_{\mathcal{G}}) \right\rangle \in \Delta_{\mathcal{Z}} \text{ if and only if there are:}$$

- $a_{\mathcal{X}}, b_{\mathcal{X}} \in \Delta_{\mathcal{X}}$ and $(a_{\mathcal{X}}, \begin{bmatrix} x \\ y \end{bmatrix}, b_{\mathcal{X}}) \in \Delta_{\mathcal{X}}$
- $c_{\mathcal{G}}, d_{\mathcal{G}} \in \Delta_{\mathcal{G}}$ and $(c_{\mathcal{G}}, \begin{bmatrix} x \\ y \end{bmatrix}, d_{\mathcal{G}}) \in \Delta_{\mathcal{G}}$

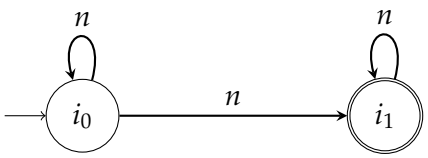
the *intersection transducer* for $\mathcal{I}$, $\mathcal{B}$ and $\mathcal{G}$.

This way of solving the problem allows for an **on the fly** algorithm. We compute successor states $d_{\mathcal{G}}$ necessary for the construction of states in $\mathcal{Z}$ (as described in definition 4.4.1) lazily

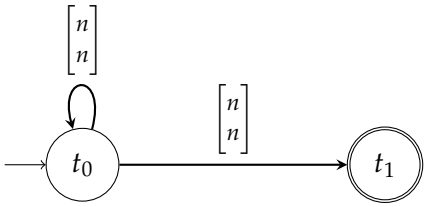
(via the *lazy step game* 4.2.1) and immediately terminate if we find a final state in $\mathcal{Z}$ (which correlates to disproving the property). Most importantly, we only construct states in $\mathcal{G}_{trap}$ that are reachable in the intersection (and, as such, contribute to proving the property). In other words, we do not need to construct the entire transducer $\mathcal{G}_{trap}$ to prove the property. Note that in practice, both tools [5, pytraps] and [4, dodo] only store states of $\mathcal{Z}$ and not transitions. This is because they do not contribute to proving or disproving the property captured by $\mathcal{B}$. Consequently, we are only interested in finding a final state in the intersection $[\![\mathcal{I} \times \mathcal{B}]\!] \cap [\![\mathcal{G}_{trap}]\!] \stackrel{!}{=} \varnothing$. We illustrate the construction of $\mathcal{Z}$ in the following example:

Example 4.4.1| Constructing the intersection transducer $\mathcal{Z}$.

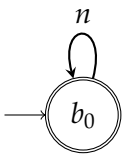
Observe the following RTS $\mathcal{R} = \langle \Sigma, \mathcal{I}, \mathcal{T} \rangle$ and property $\mathcal{B}$, with $\Sigma = \{n\}$.
The transducer below accepts the language of initial configuration $[\![\mathcal{I}]\!]$:



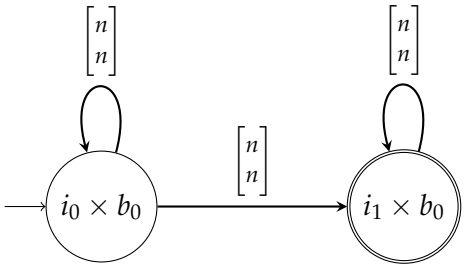
The transducer below accepts the transition language $[\![\mathcal{T}]\!]$:



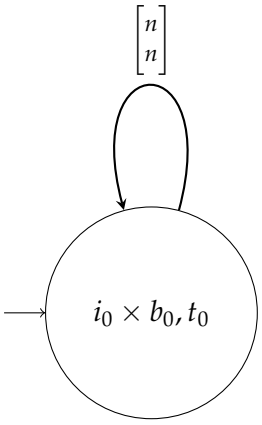
The language of "bad" words $[\![\mathcal{B}]\!]$ is accepted by the transducer:



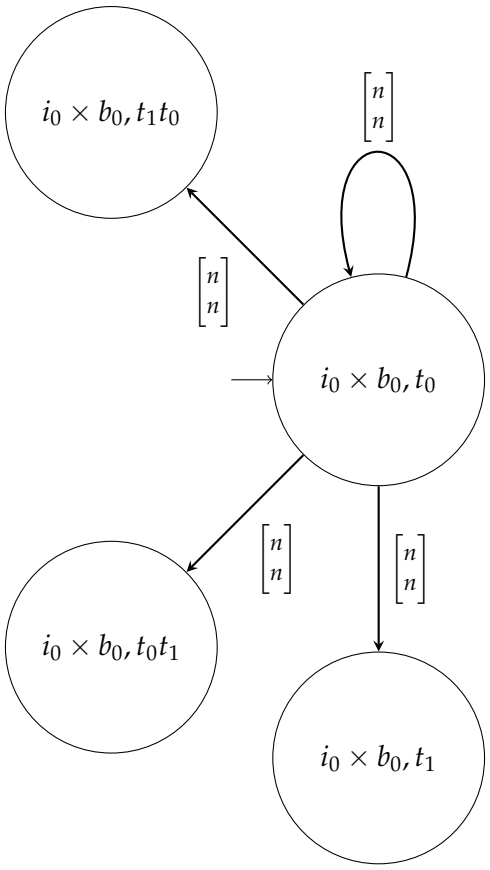
In the following, we will construct the transducer $\mathcal{Z}$ for $\mathcal{I}$, $\mathcal{T}$, and $\mathcal{B}$. Additionally, we will illustrate how results of the *lazy step game* can be cached and reused for the construction of $\mathcal{Z}$.
We construct the transducer $\mathcal{X} = \mathcal{I} \times \mathcal{B}$. A word $(u, v) \in \Sigma^* \times \Sigma^*$ is accepted by $\mathcal{X}$, if and only if u is accepted by the transducer $\mathcal{I}$ and v is accepted by the transducer $\mathcal{B}$ (remember that we fix the length of configurations and as such u and v must have the same length). The following transducer captures $[\![\mathcal{X}]\!]$:



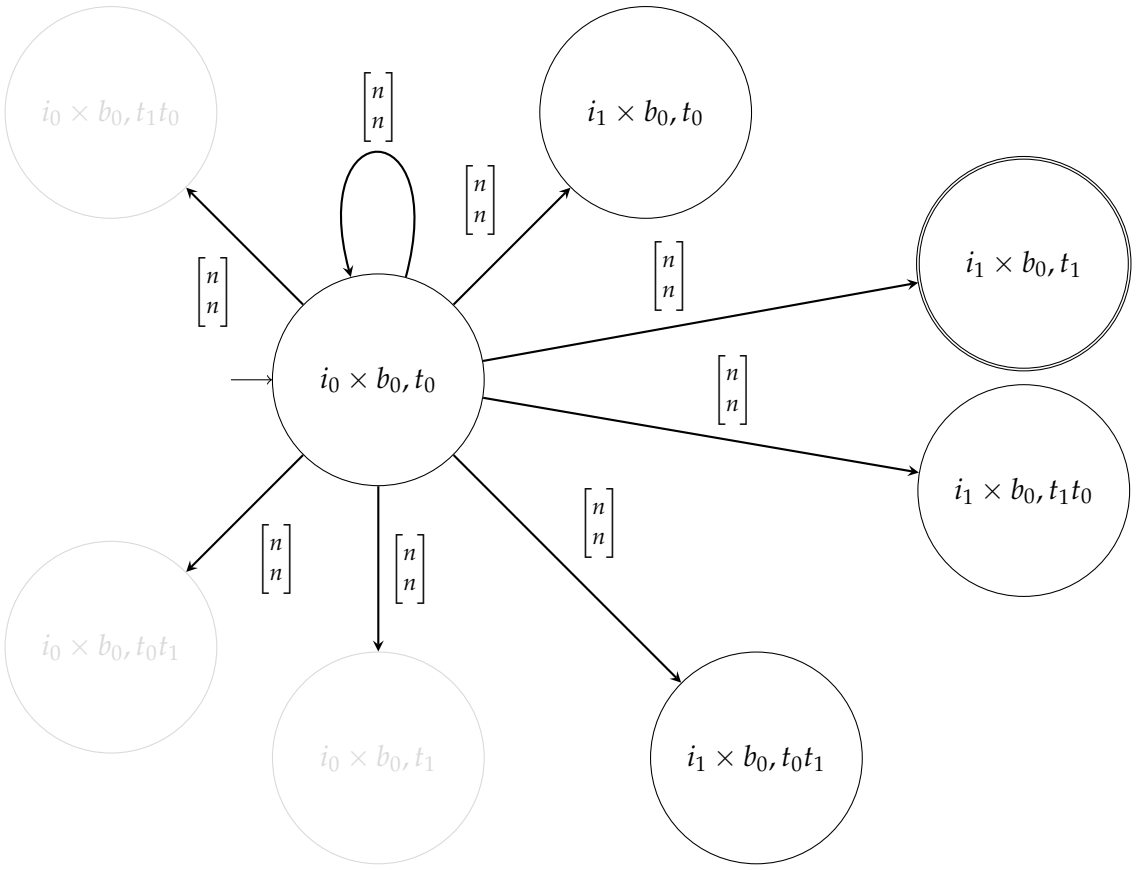
In the next step, we begin constructing $\mathcal{Z}$. Consider the pairing $\langle i_0 \times b_0, t_0 \rangle$ of the initial states of $\mathcal{X}$ and $\mathcal{G}_{trap}$ (recall that the set of initial states of $\mathcal{G}_{trap}$ coincides with the set $bases(\mathcal{Q}_0) = \{t_0\}$, where $\mathcal{Q}_0$ are the initial states of $\mathcal{T}$). This pairing of states represents the initial state of $\mathcal{Z}$ and is depicted by the following automaton:



Next, we will compute the successors of $\langle i_0 \times b_0, t_0 \rangle$ in $\mathcal{Z}$. We consider the first component of the state $\langle i_0 \times b_0, t_0 \rangle$, namely the initial state $i_0 \times b_0$ of $\mathcal{X}$. From $i_0 \times b_0$, the states $i_0 \times b_0$ and $i_0 \times b_1$ can be reached in $\mathcal{X}$ via the transition $\begin{bmatrix} n \\ n \end{bmatrix}$. Let us first fix the successor $i_0 \times b_0$. Using the *lazy step game*, we now compute the set of successors Ω of the second component t_0 for the transition $\begin{bmatrix} n \\ n \end{bmatrix}$. We derive that $\Omega(\langle t_0, \begin{bmatrix} n \\ n \end{bmatrix} \rangle)$ coincides with set of states $\{t_0, t_1, t_0t_1, t_1t_0\}$. By pairing the component wise successors of $\langle i_0 \times b_0, t_0 \rangle$ in $\mathcal{X}$ and $\mathcal{G}_{trap}$, we obtain the successors $\langle i_0 \times b_0, t_0 \rangle$, $\langle i_0 \times b_0, t_1 \rangle$, $\langle i_0 \times b_0, t_0t_1 \rangle$ and $\langle i_0 \times b_0, t_1t_0 \rangle$ in $\mathcal{Z}$. We depict the newly explored states in the following transducer:

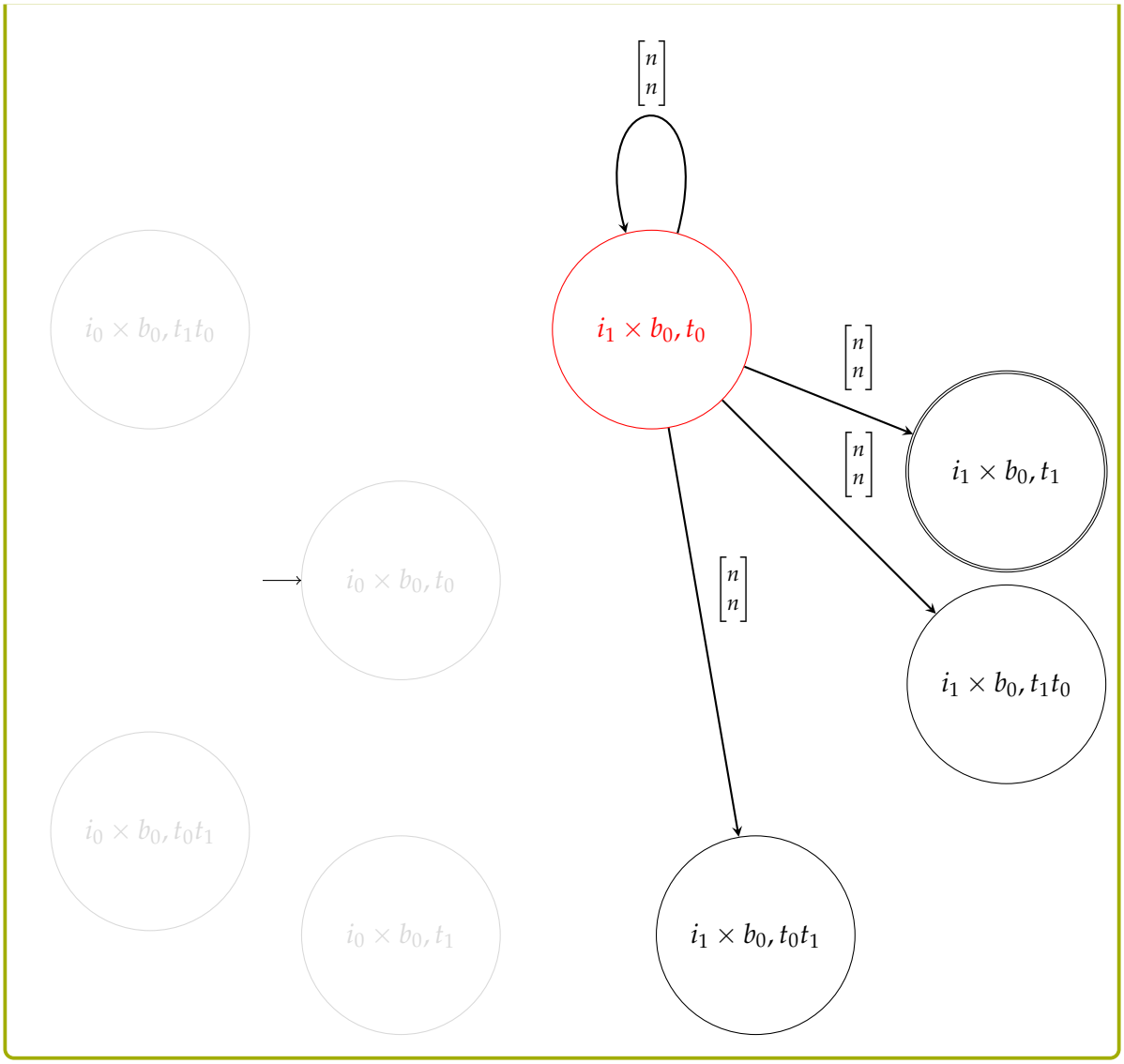


Let us now fix the **other** successor in $\mathcal{X}$ of the first component of $\langle i_0 \times b_0, t_0 \rangle$, that is $i_1 \times b_0$. For the second component t_0, we observe that we have already computed its successors in the previous step. This motivates caching the result of every played lazy step game for a state c in $\mathcal{G}_{trap}$ and a transition $a \in \Sigma \times \Sigma$. Using the result of the prior execution of the *lazy step game*, we acquire the successors $\langle i_1 \times b_0, t_0 \rangle$, $\langle i_1 \times b_0, t_1 \rangle$, $\langle i_1 \times b_0, t_0 t_1 \rangle$ and $\langle i_1 \times b_0, t_1 t_0 \rangle$ in $\mathcal{Z}$. In the following, we depict the **newly** explored successor states in black and color the previously explored successor states opaque:



We draw attention to the final state $\langle i_1 \times b_0, t_1 \rangle$. The state is final because the first component $i_1 \times b_0$ is a final state in $\mathcal{X}$, and the second component t_1 is a final state in $\mathcal{G}_{trap}$. Note that the implementations [4, dodo] and [5, pytraps] would terminate in this step because a final state in $\mathcal{Z}$ denotes a counterexample for the property $\mathcal{B}$. For the sake of this example, we disregard this fact and continue exploring $\mathcal{Z}$.

In the previous two steps, we have constructed all successors of $\langle i_0 \times b_0, t_0 \rangle$. Next, we will continue the exploration of $\mathcal{Z}$ by applying the same procedure to the successor $\langle i_1 \times b_0, t_0 \rangle$. We limit our exploration to the state above because it is the only one we can make progress in. This is because the step game can (in our example) only be won for the $t_0 \in \mathit{bases}(\mathcal{T})$ of $\mathcal{G}_{trap}$. As such, necessary successors (for constructing successors in $\mathcal{Z}$) of the second component can only be found if the component is t_0. This is the case only for the states $\langle i_0 \times b_0, t_0 \rangle$ and $\langle i_1 \times b_0, t_0 \rangle$. Again, we notice that we have played this particular step game before and pair the cached results with the first component $i_1 \times b_0$. Adding the new states to our illustration, we completed our exploration of $\mathcal{Z}$. For the sake of readability, previously discovered transitions have been omitted. The state $\langle i_1 \times b_0, t_0 \rangle$ for which we computed successors is marked in red.


$\begin{bmatrix} n \\ n \end{bmatrix}$
$i_0 \times b_0, t_1 t_0$
$i_1 \times b_0, t_0$
$\begin{bmatrix} n \\ n \end{bmatrix}$
$\begin{bmatrix} n \\ n \end{bmatrix}$
$i_1 \times b_0, t_1$
$i_0 \times b_0, t_0$
$\begin{bmatrix} n \\ n \end{bmatrix}$
$i_1 \times b_0, t_1 t_0$
$i_0 \times b_0, t_0 t_1$
$i_0 \times b_0, t_1$
$i_1 \times b_0, t_0 t_1$

4.5 Optimizing oneshot

In this section, we will present an optimization for *oneshot*. We will focus on *disproving* properties. This is because when trying to prove the contrary, we need to construct every state of $\mathcal{G}_{trap}$ that contributes to establishing the property. Consequently, we can only improve the run time of the program by optimizing the procedure to construct $\mathcal{G}_{trap}$, namely the *lazy step game* 4.2.1. We have already provided an optimization for the lazy step game in 4.3.1. When optimizing to disprove a property, we want to minimize the number of states we need to construct in $\mathcal{G}_{trap}$ to find a counterexample (a final state in $\mathcal{Z}$).

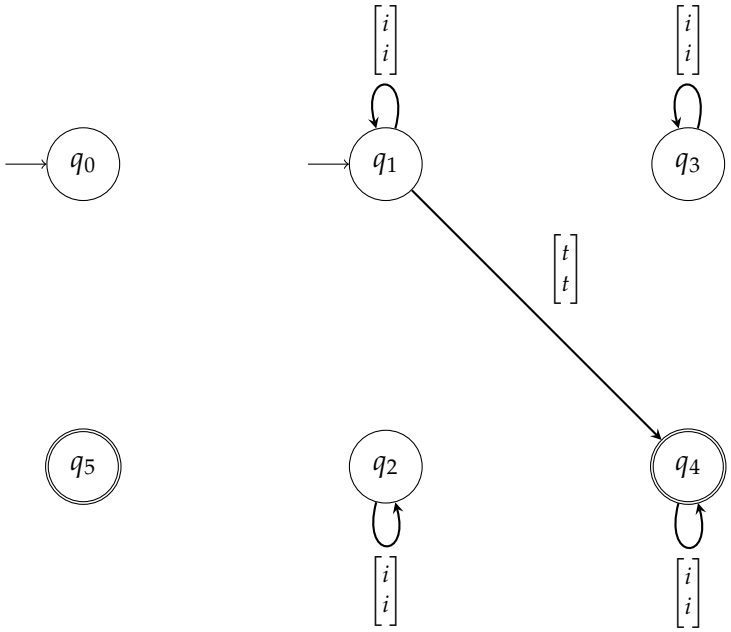
4.5.1 Reducing the alphabet to find a counterexample

The following optimization aims to find a counterexample for a property more quickly by reducing the alphabet Σ of the corresponding RTS. As a side effect, the procedure might yield false positives if it does not achieve to disprove the property. We demonstrate this in an example:

Example 4.5.1| Reducing the alphabet.

Recall the voting token passing protocol 2.2.1. Consider the property $\mathcal{B}$, where one agent is in state t and the other agents are in state i. The language capturing this property coincides with the language of initial configuration $\mathcal{I}$ depicted by the automaton in 2.2.1. In other words, this property tries to answer if, from an initial configuration, a configuration can be reached where all agents remain in the same state. We observe that this is, in fact, the case since there exists a path in the transducer $\mathcal{T}$ (portrayed in 2.2.2) between the states q_1 and q_4 that captures the transition mentioned above of the system. It follows that a final state in $\mathcal{G}_{trap}$ exists, and the property can be disproved.

The optimization idea is now to only use transition symbols that appear in the automatons $\mathcal{I}$ and $\mathcal{B}$ (namely the symbols i and t). First, we greedily restrict the alphabet of $\mathcal{T}$ to $\Sigma' = \{i, t\}$. Next we construct an automaton $\mathcal{T}'$ in which we only keep transitions that are in $\Sigma' \times \Sigma'$:



If we now try to proof the property by constructing $\mathcal{G}'_{trap}$ from $\mathcal{T}'$ and check if $\mathcal{I} \circ \mathcal{G}'_{trap} \circ \mathcal{B}$ is indeed not empty, we have successfully found a counterexample. On the other hand, if $\mathcal{I} \circ \mathcal{G}'_{trap} \circ \mathcal{B}$ is empty, then we are not able to draw any conclusions about the property since there could exist a final state that we were not able to construct due to the restricted alphabet.

We have implemented and tested this mode of *oneshot* in our tool [5, pytraps]. We illustrate the real-world performance of the method in section 5.2.3.

5 Experimental results

5.1 Implementations and Benchmarks

In this section, we will present and evaluate the experimental results of our tool [5, pytraps] and compare them to the tool [4, dodo] for the *oneshot* algorithm. Recall, oneshot constructs two transducers lazily, namely $\mathcal{G}_{trap}$ and $\mathcal{Z}$, where $\mathcal{Z}$ is derived from the intersection of $\mathcal{I} \times \mathcal{B}$ and $\mathcal{G}_{trap}$. To ease the distinction between the construction of both transducers, we will refer to the construction of states from $\mathcal{Z}$ as an *exploration* (of states) and to the construction of states from $\mathcal{G}_{trap}$ as a *generation* (of states). We have implemented two different mechanisms for the construction of states for the transducer $\mathcal{G}_{trap}$ and the transducer $\mathcal{Z}$, namely the distinction between a *breath first search (bfs)* and *depth first search (dfs)*. This means that after having generated/explored a new direct successors state in $\mathcal{G}_{trap}$ or $\mathcal{Z}$ respectively, (with a dfs) we either consider it immediately or (with a bfs) delay its consideration until all direct successors have been generated/explored. Distinguishing between a breadth-first search or depth-first search (in the two-state generating algorithms) is motivated by the fact that we try to determine if the choice of search algorithm can improve the run time of oneshot or, in particular, lead to a counterexample more quickly by considering fewer states and transitions in $\mathcal{Z}$.

Reference implementation

From now on, we will refer to the *oneshot* implementation (with a cache for previously played step games) from [4, dodo] as the *reference implementation*.

Standart implementations

In the following, we will refer to the implementation in the tool [5, pytraps]:

- Oneshot in a breadth-first search as *BfsOS*
- Oneshot in a depth-first search as *DfsOS*
- The lazy step game in a breadth-first search as *BfsGame*
- The lazy step game in a depth-first search as *DfsGame*

Note that for a oneshot implementation, be it *BfsOS* or *DfsOS*, we require a lazy step game implementation *BfsGame* or *DfsGame*. We symbolize this by joining them with a "+". For example, "BfsOS + DfsGame" symbolizes that we execute oneshot in a breadth-first search

with the lazy step game in a depth-first search. Moreover, we call the set of all combinations *standard implementations*.

For the lazy step game implementations *BfsGame* and *DfsGame*, we differentiate between ignoring or not ignoring ambiguous game states as explained in 4.3.1. In the following, we call this distinction *ambigious* and *no ambigious* (in the latter, we don't consider ambiguous game states for further computations).

System specification and benchmarks

All algorithms have been executed on an Apple Macbook Air M1 2020 with 8GB of RAM and macOS Ventura 13.4. For regular transition systems, we use the same encodings first tested with the implementation [4, dodo] in the work [3]. The following benchmarks have been tested:

- Burns
- Bakery
- MESI
- MOESI
- Synapse
- Dining cryptographers
- Voting token passing (from 2.2.1)
- Token passing

The results of the benchmarks can be found in [9] with the properties tested for the respective benchmarks.

5.2 Benchmark results

5.2.1 Average time

Firstly, all tested properties could be established; in other words, there were no timeouts. On average (across all tests), the standard implementations of oneshot (without the improved step game) require roughly the same time to execute the benchmarks. In other words, there is no significant difference in exploring $\mathcal{Z}$ in a bfs or dfs and similarly exploring $\mathcal{G}_{trap}$ in a bfs or dfs.
On the other hand, we could significantly reduce the average time by allowing the exclusion of ambiguous states in the lazy step game. This optimization yields, on average, a speed-up of about **1.6** in comparison to the pure standard implementations.

The table below contains the average time of the different *standard implementations*.

	BfsOS + BfsGame	DfsOS + BfsGame	BfsOS + DfsGame	DfsOS + DfsGame
ambiguous	26.894s	27.952s	27.553s	28.414s
no ambiguous	16.735s	17.398s	16.878s	17.397s

Table 5.1: Average Time in seconds to execute all benchmarks

The average time of the *reference implementation* was **24.918s**. Consequently (for this set of benchmarks), we could improve the state of the art by ignoring ambiguous states in the step game.

5.2.2 Disproving a property

One disparity between the use of different search algorithms becomes visible if we take a closer look at the property "gamewon" for the voting token protocol (in example 2.2.1). This property captures if we can potentially reach (from the the set of initial states illustrated in figure 2.2.1) a configuration in which all agents are marked. The property indicates whether all regions can access the token to enter their votes. One can infer from the transition transducer $\mathcal{T}$ (in 2.2.2) of the RTS that this is indeed the case.
Consequently, we correctly capture the property with oneshot if the algorithm *disproves* the property. Recall that disproving a property corresponds to finding an accepting state in $\mathcal{Z}$ and, as a result, concluding that a (bad) configuration is potentially reachable from an initial configuration. The table 5.2 below depicts the required time to disprove the property "gamewon" for all standard implementations.

	BfsOS + BfsGame	DfsOS + BfsGame	BfsOS + DfsGame	DfsOS + DfsGame
ambiguous	0.0322s	21.554s	0.0305s	21.514s
no ambiguous	0.0317s	14.043s	0.0311s	13.963s

Table 5.2: Time in seconds to disprove the property "gamewon"

If $\mathcal{Z}$ is explored with *BfsOS*, we need significantly less time to prove the property. In comparison, constructing $\mathcal{G}_{trap}$ with *BfsGame* or *DfsGame* is negligible. Moreover, ignoring ambiguous states in the step game only improves the inferior *DfsOS* implementations. For completion, the *reference implementation* took **0.095s** to disprove the property.
We can explain this enormous gap between the *standard implementations* by looking at the exploration of $\mathcal{Z}$.

The following two figures depict the number of explored states and transitions (in $\mathcal{Z}$) constructed while disproving the property. In the case of transitions, we do not represent the number of unique transitions but rather the number of successful computations that yield a transition in $\mathcal{Z}$.

	BfsOS + BfsGame	DfsOS + BfsGame	BfsOS + DfsGame	DfsOS + DfsGame
ambiguous	1293	1078	1293	1078
no ambiguous	1293	1078	1293	1078

Table 5.3: Number of explored states in $\mathcal{Z}$

Even though the *BfsOs* implementations of oneshot are significantly faster, they explore slightly more states until they find a final state in $\mathcal{Z}$. The real difference lies in the number of transitions computed by the bfs and dfs implementations.

	BfsOS + BfsGame	DfsOS + BfsGame	BfsOS + DfsGame	DfsOS + DfsGame
ambiguous	1293	50359309	1293	50359309
no ambiguous	1293	18027861	1293	18027861

Table 5.4: Number of explored transitions in $\mathcal{Z}$

With *DfsOS*, disproportionately more transitions are explored in $\mathcal{Z}$ compared to the number of explored states. On the other hand, with *BfsOs*, the number of transitions directly correlates with the number of explored states. We can conclude that, for this property, the implementation *BfsOs* is favorable. Notably, the exploration of transitions in $\mathcal{Z}$ directly depends on the generation of transition in $\mathcal{G}_{trap}$. As such, the time to disprove the property is primarily spent in the *lazy step game* procedure to generate transitions in $\mathcal{G}_{trap}$.
Moreover, observe that without ignoring ambiguous game states in the lazy step game, we potentially construct the same transition multiple times. For this reason, the number of transitions differ in the rows "ambiguous" and "no ambiguous" in the table 5.4.

Lastly, two implementations are similarly fast when *proving* a property because, in this case, all possible transitions that contribute to establishing the property have to be computed. Consequently, the search does not terminate preemptively, as seen in the case of *disproving* a property.

5.2.3 Performance of reducing the alphabet

Recall the optimization to disprove a property presented in section 4.5.1. In short, we reduce the alphabet Σ to find a minimal counterexample (a final state in $\mathcal{Z}$). The smaller alphabet eliminates transitions in the transducer $\mathcal{T}$, reducing the complexity of the lazy step game to compute states in $\mathcal{G}_{trap}$ (we just consider fewer states). We can draw no conclusions about the property if the search does not yield a counterexample. Note that the procedure successfully

returned a counterexample for the property discussed in the following.
Consider the property "gamewon" for the token voting protocol explained in the previous section 5.2.2. In this instance, we can reduce the alphabet to $\{i, t, m\}$ (in comparison to $\Sigma = \{i, r, t, m\}$) because the initial configurations only contain the letters i and t and the configurations of property "gamewon" only contain the letter m. The table below compares the implementation minSigma to the standard implementation BfsOs. For both, we ignore ambiguous states in the lazy step game:

	BfsOS	minSigma
BfsGame	31.7ms	0.0669ms
DfsGame	31.1ms	0.0579ms

Table 5.5: Time in milliseconds to disprove the property "gamewon"

Once more, performing the construction of $\mathcal{G}_{trap}$ in the mode BfsGame or DfsGame does not make a difference. Although minSigma is faster than the standard implementation BfsOS, the RTS (and property) needs to be more complex to enable us to draw accurate conclusions about the performance of minSigma. It remains open to test minSigma for an RTS and a property where the *reduced* Σ is significantly smaller than the actual Σ.
Nevertheless, the procedure allows us to quickly perform first speculations about a large batch of properties and regular transition systems. Independently of the modes BfsGame or DfsGame, minSigma performed all benchmarks in roughly **0.85ms** (with, of course, unable to *prove* properties).

6 Conclusion

In this thesis, we analyzed the practical possibilities of reasoning over regular transition systems with *inductive statements* and the *trap-interpretation* presented in [3]. These techniques yield a *PSPACE*-complete algorithm that was first implemented in the tool [4, dodo] under the name *oneshot*. Comparing the two sources, we explained in detail how the theory of inductive statements was applied in practice in oneshot. Moreover, we have improved the state of the art by re-implementing oneshot in our tool [5, pytraps]. Additionally, we embarked on a quest to improve the algorithm's runtime (in practice) with different experimental approaches.

6.1 Further work

We want to conclude this thesis with a list of experimental approaches whose viability must still be proven in practice.

Parallelizing the construction of $\mathcal{Z}$. The intersection transducer $\mathcal{Z}$ can be constructed in parallel. After exploring a successor state z' for a state z in $\mathcal{Z}$, we can simultaneously explore the successors of z' in a new thread. It is uncertain if the overhead of initializing and synchronizing threads outweighs the benefits of parallel construction. Additionally, it remains unclear for which successors a new thread should be initialized because the number of successors for a state in $\mathcal{Z}$ vastly outnumbers the number of physical cores on an ordinary processor.

Disproving properties by heuristically exploring $\mathcal{Z}$. This method aims to quickly find a counterexample for a property. For this, we first consider a *viable* path to a final state in $\mathcal{Z}$, and only *afterward* prove that such a path can be constructed with the method explained in this work. The problem lies in determining what states and transitions in $\mathcal{Z}$ constitute a viable path. A possible method of solving this problem would be to heuristically exclude successors for states (that likely are not winning in the step game) in $\mathcal{Z}$ and, as such, reduce the number of possible successors in every link of the path. It is questionable if good heuristics for this method exist.

Computing the entire transducer $\mathcal{G}_{trap}$. Recall that we determine if $[\![\mathcal{I} \times \mathcal{B}]\!] \cap [\![\mathcal{G}_{trap}]\!] \overset{!}{=} \varnothing$ to solve the *potential reachability problem 2*. In *oneshot*, we compute the statement lazily, which means that we only generate transition and states in $\mathcal{G}_{trap}$ that are reachable in the intersection and additionally terminate the computation of $\mathcal{G}_{trap}$ preemptively if we find a final state in the intersection. For a large batch of properties $[\![\mathcal{B}]\!]$ (and possibly different initial languages $[\![\mathcal{I}]\!]$), it could be beneficial to first compute the entire transducer $\mathcal{G}_{trap}$. Subsequently we compute

$[\![\mathcal{I} \times \mathcal{B}]\!] \cap [\![\mathcal{G}_{trap}]\!] \stackrel{!}{=} \varnothing$ for the batch of initial languages $[\![\mathcal{I}]\!]$ and properties $[\![\mathcal{B}]\!]$. This has the benefit that we only perform the costly computation of $\mathcal{G}_{trap}$ once and afterward only conduct comparatively cheap intersections. On the other hand, we might compute large parts of $\mathcal{G}_{trap}$ that are not necessary to prove/disprove any properties from this batch. Moreover, this method would require us to not only store the states of $\mathcal{G}_{trap}$ but also the transitions. It has to be proven in practice if this method can be superior to the established lazy implementations and for which classes of problems this could be the case.

Bibliography

[1] P. A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena. "A Survey of Regular Model Checking". In: *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*. Ed. by P. Gardner and N. Yoshida. Vol. 3170. Lecture Notes in Computer Science. Springer, 2004, pp. 35–48. DOI: 10.1007/978-3-540-28644-8_3. URL: https://doi.org/10.1007/978-3-540-28644-8%5C_3.

[2] J. Esparza, M. Raskin, and C. Welzel. *Regular Model Checking Upside-Down: An Invariant-Based Approach*. 2023. arXiv: 2205.03060 [cs.DC].

[3] C. Welzel. "Inductive Statements for Regular Transition Systems (under review)". PhD thesis. TUM, 2023.

[4] C. Welzel. *Dodo*. https://gitlab.lrz.de/i7/dodo/-/tree/main/src/main/java/de/tum/in/model/dodo?ref_type=heads. 2023.

[5] M. Kamps. *pytraps*. https://github.com/maximiliankamps/Traps. 2023.

[6] J. Esparza and M. Blondin. *Automata theory An algorithmic approach*. Addison-Wesley Professional, 1994.

[7] P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. "Regular model checking made simple and effcient". In: *International Conference on Concurrency Theory*. Springer. 2002, pp. 116–131.

[8] R. Bloem, S. Jacobs, A. Khalimov, I. Konnov, S. Rubin, H. Veith, and J. Widder. "Decidability in Parameterized Verification". In: *SIGACT News* 47.2 (2016), pp. 53–64. URL: https://publications.cispa.saarland/1409/.

[9] M. Kamps. *Benchmark results for dodo and pytraps*. Zenodo, Feb. 2024. DOI: 10.5281/zenodo.10615624. URL: https://doi.org/10.5281/zenodo.10615624.