POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Formal Verification of Parametric Timed Automata with TABEC

Tesi di Laurea Magistrale in
Computer Science Engineering
Ingegneria Informatica

Author: **Andrea Manini**

Student ID: 995318
Advisor: Prof. Pierluigi San Pietro
Co-advisors: Prof. Matteo Rossi
Academic Year: 2023-24

Ringraziamenti

Desidero innanzitutto rivolgere il mio ringraziamento più profondo ai miei genitori, i quali mi hanno sempre supportato nelle mie scelte di vita e nei periodi di difficoltà. È grazie alla loro dedizione e ai loro sacrifici se ho potuto raggiungere questo traguardo.

Un grazie di cuore anche alla mia fidanzata, la quale si è sempre schierata a mio sostegno, accompagnandomi durante tutto il percorso universitario, facendomi crescere come persona oltre che come studente.

Un sentito ringraziamento anche ai miei supervisori, i professori Pierluigi San Pietro e Matteo Rossi. È grazie ai loro preziosi consigli e ad un pizzico di ironia se il lavoro contenuto in questo manoscritto ha potuto raggiungere il suo apice.

Infine, desidero porgere un merito anche a Ridley, il migliore amico che un essere umano possa desiderare. Grazie per avermi rallegrato e tenuto compagia durante le lunghe giornate passate davanti al computer.

Abstract in lingua italiana

Nel contesto della verifica formale dei sistemi, specialmente dei sistemi digitali, il Model Checking (MC) rappresenta una delle tecniche automatiche più efficaci. Considerando un modello formale del sistema in analisi e una specifica proprietà formale, il MC si prefigge di verificare la validità di tale proprietà all'interno del sistema in questione. Uno dei formalismi più rilevanti nel MC è rappresentato dagli Automi Temporizzati, altresì noti come Timed Automata (TA), i quali possono assumere la funzione di accettori di linguaggi, la natura di questi ultimi dipendente dalla definizione adottata di TA. Un problema di notevole rilevanza associato ai TA riguarda la determinazione della vacuità del loro linguaggio, conosciuto come il problema della emptiness. È noto che per TA convenzionali, tale problema risulta essere generalmente decidibile, permettendo di conseguenza la creazione di un algoritmo decisionale risolutivo. La questione assume un carattere generale di indecidibilità qualora la definizione dei TA contempli la presenza di parametri. La presente tesi si dedica all'analisi del problema della emptiness per una specifica categoria di TA nota come non-resetting test TA (nrtTA) parametrici. Per una sottoclasse particolare di tale modello, il problema della emptiness risulta essere decidibile. Questo ha comportato la creazione di un algoritmo decisionale di cui è stata effettuata l'implementazione attraverso uno strumento appositamente sviluppato. Inoltre, la generazione randomica di TA da parte dello strumento ha consentito l'arricchimento del medesimo con funzionalità di testing randomico, incluso l'inserimento di abilità di predizione da oracolo. Tale circostanza implica la capacità dello strumento di anticipare l'esito dei test prima che questi vengano eseguiti. La validità dei risultati teorici e pratici ottenuti è stata infine confermata mediante un'esaustiva fase di testing, sia per quanto concerne la correttezza delle risposte fornite dall'algoritmo, sia per quanto riguarda la scalabilità dello strumento, sia per quanto concerne la correttezza delle previsioni da oracolo fornite.

Parole chiave: verifica formale, model checking, timed automata, testing, oracolo

Abstract

In the context of formal verification of systems, particularly digital systems, Model Checking (MC) stands out as one of the most effective automated techniques. Considering a formal model of the system under analysis and a specific formal property, MC aims to verify the validity of this property within the given system. One of the most relevant formalisms in MC is represented by Timed Automata (TAs), which can act as acceptors of languages, the nature of which depends on the adopted definition of TAs. A significantly relevant problem associated with TAs concerns determining the emptiness of their language, known as the emptiness problem. It is known that, for conventional TAs, this problem is generally decidable, allowing for the creation of a decision algorithm. The problem becomes generally undecidable when the definition of TAs allows parameters. This thesis focuses on examining the emptiness problem for a specific category of TAs known as parametric non-resetting test TAs (nrtTAs). For a particular subclass of this model, the emptiness problem proves to be decidable. This led to the creation of a decision algorithm, which was implemented inside a specifically developed tool. Furthermore, the tool's capacity of randomly generating TAs has allowed its enrichment with random testing functionalities, including the incorporation of oracle prediction capabilities. This implies the tool's ability to anticipate the outcomes of tests before their execution. The validity of both theoretical and practical obtained results was ultimately confirmed through a comprehensive testing phase, encompassing the correctness of algorithmic responses, the scalability of the tool, and the accuracy of provided oracle predictions.

Keywords: formal verification, model checking, timed automata, testing, oracle

Contents

1 | Introduction

Society is nowadays heavily dependent on technology for accomplishing a great variety of tasks in daily life scenarios. The benefits brought by machines can be compared to the benefits brought by the First and Second Industrial Revolutions, if not outclassing the latter. Due to this deep permeation of technology in society, particular attention must be put into guaranteeing the correct behavior of digital systems, ranging from simple scenarios (e.g., sending an e-mail) to high-risk, life-threatening scenarios (e.g., an airplane autopilot securely and reliably transporting passengers to destination). If some design flaws in these systems are not detected before deployment, disasters may occur, as emphasized by the following examples. In 1996, the Ariane-5 (a now retired European heavy-lift space launch vehicle) crashed 36 seconds after its launch, due to a 64-bit floating point into a 16-bit integer value conversion [1]. Fortunately, there were no injuries, but the failure cost hundreds of millions of dollars. Another example, that happened between 1985 and 1987, is the Therac-25, a radiation therapy machine which, due to poor software design, caused the death of six cancer patients as a consequence of radiation overdose [2].

It is thus fundamental to introduce in the whole systems' development life cycle some formal verification techniques to ensure the correctness of such systems. The application of formal verification techniques ranges over several domains, e.g., microcode and firmware, national infrastructures, and biometric-based security applications.
Some successful examples of formal verification techniques use cases are the following: the Transputer series of microprocessor chips, in which the formal development of a correct-by-construction floating-point unit was required; the Maeslant Kering Storm Surge Barrier (a movable barrier protecting the port of Rotterdam from flooding), which explicitly required the use of formal verification techniques due to the high-security threat level of its application; the Mondex Smart Card, deployed in the early 1990s by the National Westminster Bank and Platform Seven, which required to ensure correctness and security of transactions to avoid counterfeiting of money.
These, and additional examples of formal verification techniques use cases, can be found in [3] and [4], the latter being a survey on the application of formal verification techniques in the railway signaling domain, a field in which they have traditionally been applied.

Due to the intricacy and difficulty of proof of correctness based on mathematical logic, automating verification procedures is of uttermost importance. A particularly effective automated technique is Model Checking, supported by tools called Model Checkers. Quoting the definition given in [5], Model Checking is *"an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model"*.
Model Checking is carried out in three main phases: the modeling phase (in which the model of the system is defined), the running phase (in which the Model Checker tries to check the validity of the model), and the analysis phase (in which results are collected and examined). From the above examples and considerations, it can be inferred that, in scenarios requiring the highest level of system correctness, it is fundamental to build a correct model of the system under analysis, accurately selecting the proper formalism used to describe it.

The work presented in this thesis deals with a particular formalism used in Model Checking: Timed Automata (from now on, also referred to as TAs), a mathematical formalism introduced since concurrent and real-time applications are among the most difficult to develop and test. Anticipating what will be explained in the following chapters, the reference TA model of this thesis is a parametric TA with two clocks and one parameter, satisfying the non-resetting test (nrt) property and subject to a Büchi acceptance condition.

The first main goal of this thesis is to find and implement a decision algorithm able to solve the emptiness problem for the TA model just introduced, i.e., the algorithm must be able to tell if the language accepted by a particular TA (satisfying the aforementioned properties) is empty or not. This led to the creation of a tool: TABEC (Timed Automata Builder and Emptiness Checker) which, by leveraging an already existing external tool (namely tChecker), can solve the emptiness problem for TAs, even in the case where TAs under analysis do not satisfy the properties of this thesis' reference TA model. During TABEC's implementation, some theoretical concepts have been defined and introduced, in particular tiles and Tiled TAs, used to create TAs capable of forcing the range of values in which the parameter can fall.
The second main goal of this thesis is to address a common problem in Model Checking, which can be summarized in the sentence: *"Who tests the tester?"*. In different words, how can programs used to carry out tests of a model be considered correct? And also, how can the correctness of the behavior exhibited by programs undergoing testing be assessed? By exploiting its ability to randomly generate TAs and the property of tiles allowing to ensure that the parameter's value must be bounded inside a predetermined interval, TABEC also tackles this issue, providing random test cases and forecasting

on their results. Performance indices containing information about physical resource utilization (i.e., memory and computing time) are also measured while executing tests.

This thesis is organized as follows: in Chapter 2 a survey exploring the current State of the Art on topics related to TAs is presented. Then, in Chapter 3 a theoretical background on TAs lays the foundations to understand the results obtained in subsequent chapters. Chapter 4 focuses on the thesis' first main goal, providing both a theoretical and practical description of how it has been achieved. Next, Chapter 5 directs attention toward the thesis' second main goal, by making a theoretical introduction about the concepts of tiles and Tiled TAs, subsequently describing their actual implementation. In Chapter 6 some experimental results and considerations are reported. Finally, Chapter 7 concludes the thesis.

2 | State of the Art

This chapter aims to give a brief overview of the current state-of-the-art topics about TAs. In particular, results regarding the emptiness problem for non-parametric and parametric TAs are first analyzed. No decidability results about the emptiness problem for the TA model used in this thesis were found in the published literature. Since TABEC can serve testing purposes, a survey of this field is also presented. Lastly, two auxiliary tools used in conjunction with TABEC are shortly introduced.

2.1. Decision Problems for Timed Automata

In Computer Science, there exists a particular class of problems, namely decision problems, which are distinguished by the characteristic that their solution hinges on a binary decision: the determination of either a positive ("yes") or negative ("no") answer.
Problems belonging to this class can be categorized as decidable problems (for which an algorithm computing the correct answer exists or can be constructed), undecidable problems (for which no algorithm exists or can be constructed to compute the correct answer), and semi-decidable problems (for which there exists an algorithm that, given any input instance which satisfies a certain property, halts and outputs "yes" but, if the input does not satisfy the property, the algorithm may either run forever or not halt at all).
Decision problems have been the focus of many researchers since the early stages of Computer Science, contemplating a great variety of mathematical formalisms, TAs included.

Decision problems about TAs arose since the first paper which introduced this formalism [6] and have been extensively studied since then. In [7], a survey of the main decision problems for non-parametric TAs is given: reachability and emptiness are decidable in general, while language inclusion, language equivalence, and the universality problem (i.e., the problem of determining if a given TA accepts all possible timed traces) are undecidable in general. By restricting the expressiveness of the classic TA formalism, some of these latter problems might become decidable (e.g., the language inclusion problem becomes decidable when considering a TA and a bounded 2-way deterministic TA).
When TAs are enriched with parameters (i.e., real values to be determined), obtaining

parametric TAs, decision problems turn out to be undecidable in general. A relatively concise work containing some proofs about the (un)decidability of the emptiness problem for parametric TAs can be found in [8], where the authors provide a set of results obtained by changing the time semantics of the given TAs (either discrete or continuous) and the parameters' domain (either integer or real). They prove emptiness to be decidable for TAs having 1 parametric clock (out of n total clocks), with integer parameters (considering a total of m parameters) in either discrete or continuous time semantics.
A more extensive survey on decision problems for parametric TAs is given in [9], where not only several flavors of emptiness problems are addressed (one of which is the EF-emptiness problem, asking whether the set of parameter valuations such that a given location is reachable is empty), but other problems like the synthesis problem or the membership problem for parametric TAs are considered. This work is a more exhaustive version of a previous survey authored by one of the same contributors [10], where a meticulous summary of the main decidability results about parametric TAs is reported. In particular, the focus is on EF-emptiness, showing various cases where it is decidable and cases where it is undecidable. These different cases are obtained by varying the characteristics of the given TAs, such as the number of parametric and non-parametric clocks, the number of parameters, and the semantics of TAs. Only in a few particular cases, the emptiness problem becomes decidable; in the others, it is undecidable or still an open problem to solve.

Despite the vast quantity of material regarding the emptiness problem for parametric TAs, no results were found in the published literature about the class of TAs considered in this thesis, i.e., parametric TAs with two clocks and one parameter, satisfying the nrt property and subject to a Büchi acceptance condition. A proof for the decidability of the emptiness problem for this specific class of TAs is given in [11]. It is important to notice that in this work, differently from the ones reported within this section, the focus is on timed ω-words and thus on timed ω-language emptiness, which differs from, e.g., the EF-emptiness, the latter being a sort of reachability problem.

2.2. Software Testing

Despite the main focus of this thesis being on TAs and in particular on their emptiness problem, it is necessary to give a brief overview of the most popular verification technique, namely testing, since as it will be explained in subsequent chapters, TABEC is strictly correlated with it.

First, it is necessary to make a distinction between Model Checking and testing (in par-

ticular, the focus in this thesis is on software testing). Both are automated verification techniques that aim to assess the correctness of the system under study and, for this reason, may inadvertently be construed as synonyms.
However, this is not the case, since they exhibit some differences, as pointed out in [5]. Testing aims at verifying a piece of software's correctness by actively running it and forcing it to traverse a set of determined execution paths. Trying to simulate all possible execution paths causes an unfeasible blow-up in computational complexity and, for this reason, only a limited set of execution paths are examined. Paraphrasing a famous quote from Edsger W. Dijkstra: *"Program testing can be used to show the presence of bugs, but never to show their absence!"*.
Model Checking, on the other hand, explores all the possible states of a system using a brute-force approach. This is essential in showing that a system fully satisfies (or doesn't satisfy) a certain property. Model Checking is bound with the given system's model, the latter affecting how well the satisfaction of properties is detected by the former. Hence, techniques like testing may complementarily be used to detect errors.

Due to the ever-changing nature of technology and digital systems, testing techniques evolved and some completely new techniques even entered the scene. There are plenty of surveys in the literature regarding the evolution of testing over the years, a good example of which is given by [12]. As pointed out by the authors of this survey, there now exist several flavors of testing, each with its strengths and weaknesses. Among the most notorious ones, it is worth mentioning regression testing (performed exploiting regression techniques given a program P, a modified version P' and a test suite T to test P'), combinatorial testing (a technique which addresses the problem of considering different configurations needed during testing), model-based testing (in which test suites are derived from a model of the system) and random testing (a traditional technique where testing inputs are randomly and automatically generated).

In particular, random testing can be classified under the Automated Test Input Generation set of testing techniques. These methods have been existing in the testing scenario even before the 2000s, continuing to elicit the attention of professionals and researchers. One of the most significant problems concerning random testing is assuring both the correctness of the system under study and of the results themselves. This is because, since inputs are randomly generated, there is no clue if the correct solution may be reached or if undefined behavior will occur. This problem is known in the literature as the (testing) oracle problem and it first appeared in 1978 in the work of William E. Howden about program testing [13].

From that moment on, the oracle problem has become of public interest and several works

on this topic are now available. In [14] an introduction to the oracle problem is given, as well as a taxonomy of the different types of existing oracles. The three main oracle classes identified in the aforementioned work are: specified test oracles (defined through a specification language, e.g., through a model-based language, through assertions, or algebraic languages), derived test oracles (which use information derived from some artifacts to determine the correctness of the system's behavior) and implicit test oracles (where the correctness of tests can be assessed using implicit knowledge, e.g., considering buffer overflows always as errors).
Of particular interest in this thesis are derived test oracles. Among the possible artifacts that can be used to build a derived test oracle, there is textual documentation, particularly important since professionals are typically more accustomed to natural languages rather than formal languages. There are two ways in which textual documentation can be used to build derived test oracles: one way consists of using techniques to derive a formal specification directly from one written in a natural language; another way concerns the restriction of a natural language in a semi-formal subset which can be automatically processed by machines (e.g., as it has been done for the English language in [15]).

To end the discussion on testing presented in this section, it is possible to anticipate a result that will be developed later in this thesis.
Thanks to its random TAs generation capabilities, from a software engineering point of view, TABEC can serve the purpose of performing random testing both on itself and on tChecker. In addition, tiles can force the interval in which the parameter's value may fall, hence making it possible to integrate oracle capabilities inside TABEC: more precisely, TABEC is enriched with derived test oracle capabilities, where the used artifacts are the descriptions of tiles themselves, containing a specification of the parameter's intervals defined by using a formal language.

2.3. Tools for Timed Automata

Various tools with different features are available for application in tasks related to TAs. This section focuses on the ones used in the development of this thesis. Further considerations will be introduced or picked up again when needed in subsequent chapters.

The de-facto standard tool for working with TAs is Uppaal [16], released in 1995 and developed in collaboration between the Department of Information Technology at Uppsala University (Sweden) and the Department of Computer Science at Aalborg University (Denmark). The tool is freely distributed and actively kept updated. At the current time of writing, it has reached the version 5.0.

Uppaal aims to allow professionals and researchers to be able to model, validate, and verify real-time systems using the formalism of TAs, which can be put in communication with one another by using channels or shared data structures.
Bundled with a clean Graphical User Interface (GUI), Uppaal is extremely easy to use and is composed by four main panels: the System Editor (where the system model is manually composed), the Symbolic Simulator (which allows to manually simulate the execution of the model), the Concrete Simulator (similar to the Symbolic Simulator, but allowing to choose the time interval in which to fire a transition), and the Verifier.
The Verifier allows to express and verify liveness and safety properties [5]. The query language used in Uppaal to formulate properties is a subset of TCTL (Timed Computation Tree Logic) and it can express properties like $A\Box p$, meaning that "it is always true that property p will hold in the considered system". A more detailed explanation of Uppaal's functioning is contained in its official documentation [17].
Some relevant real-life use cases of Uppaal involve Philips Audio Protocol, Bang&Oulfsen Audio/Video Protocol, and Schedulability Analysis [18].

Despite being powerful in checking the validity of safety and liveness properties, the query language used in Uppaal is not able to deal with the emptiness problem for a given TA: for this reason, an inquiry on tools able to satisfy this requirement was carried out, resulting in only one tool suitable for this purpose: tChecker [19].

The tChecker tool is more like a sort of prototype, compared to more popular solutions like Uppaal. It is an open-source tool freely distributed under the MIT license and available on GitHub for download. At the current time of writing, it has reached the version 0.8 and is actively kept updated. The tool is used to implement and test verification algorithms by several research teams: Irisa (Rennes, France), LaBRI (Bordeaux, France), LIF (Marseille, France), and CMI (Chennai, India) [18].
The prime principle leading the creation of tChecker was to allow researchers to experiment with data structures and algorithms when trying to synthesize and verify timed systems. The formalism used is the one of TAs.
The tChecker tool is a collection of four tools: a tool for checking the syntax of the given system specification (tck-syntax), a tool for simulating the execution of the system under analysis (tck-simulate), a tool for performing reachability analyses on TAs (tck-reach) and a tool for checking the emptiness condition of Büchi TAs (tck-liveness).
The main workflow that may be followed when using tChecker consists of giving a representation of the system under analysis by using tChecker's grammar, then checking the syntax of the given model (eventually trying to manually simulate it), and finally checking the reachability or emptiness of the given model.

The most relevant of the above tools for the goals of this thesis is the tck-liveness tool, where two algorithms are available for checking the emptiness of the given TAs: one is based on the concept of Strongly Connected Components (SCCs), while the other uses a nested depth-first search approach.
Since tck-liveness is extensively used by TABEC, a more detailed description of these algorithms and on tChecker itself is delayed to Chapter 5.

3 | Theoretical Background

Finite State Machines, also known as Finite Automata (FAs), are one of the most used tools in engineering to model a system as a set of states and a set of events. Events control how the system changes from one state to another. When modeling real-time systems, which require timing constraints to be met, the formalism of FAs shows its limitations, not being able to effectively model time. This led Rajeev Alur and David L. Dill to introduce, in the early 1990s, a new mathematical formalism able to provide a way for modeling time, i.e., Timed Automata (TAs) [6].
Several TAs definitions exist in the literature: in the following, the most relevant to the work presented in this thesis are introduced. However, it is first necessary to present some concepts about infinite languages to provide a better understanding of TAs.

3.1. (Timed) ω-languages and Büchi Automata

Let Σ be a (finite) set of symbols, i.e., $\Sigma = \{a_0, a_1, a_2, ..., a_n, ...\}$ and let this set be called an alphabet. It is possible to define a formal language over this alphabet as a set of finite words on Σ. This is the typical scenario when working with mathematical formalisms such as FAs. It is possible to extend this notion to consider words of infinite length over the same alphabet Σ. This is done by defining a so-called ω-language over Σ (denoted as Σ^ω), i.e., it is the set of infinite sequences of symbols $\sigma = a_0 a_1 a_2 ...$ such that $a_i \in \Sigma$. An interesting result is obtained by extending ω-languages with the concept of time, obtaining so-called timed ω-languages. A prior clarification about time sequences [6] is however needed before defining timed ω-languages.

Definition 3.1. *A* time sequence $\tau = \tau_0 \tau_1 \tau_2 ...$ *is an infinite sequence of time values* $\tau_i \in \mathbb{R}_{>0}$*, satisfying the following constraints:*

- *Monotonicity:* τ *increases strictly monotonically:* $\tau_i < \tau_{i+1}$, $\forall i \geqslant 1$.
- *Progress: for every* $t \in \mathbb{R}_{>0}$*, there is some* $i \geqslant 1$ *such that* $\tau_i > t$.

It is now possible to define a timed ω-word (sometimes simply called a timed word) over

a given alphabet Σ as a pair (σ, τ), where $\sigma = a_0 a_1 a_2 ...$ is an infinite word and τ is a time sequence. A timed ω-language over Σ is a set of timed words defined over the same alphabet Σ, i.e., it corresponds to the set $L_{\Sigma,\mathbb{R}} = \{(\sigma, \tau) \mid (\sigma, \tau) \in (\Sigma \times \mathbb{R}_{>0})^\omega\}$.
For an ω-language to be accepted by a suitable formalism, an adequate acceptance condition must be specified. The concept of ω-automata is now introduced as an acceptor for ω-languages. Due to the preliminary nature of this chapter, only Nondeterministic Büchi Automata are introduced [5].

Definition 3.2. *A* Nondeterministic Büchi Automaton *(from now on shortened as NBA)* $\mathcal{A}$ *is a tuple* $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ *where:*

- Q *is a finite set of states,*
- Σ *is an input alphabet,*
- $\delta : Q \times \Sigma \to 2^Q$ *is a transition function,*
- $Q_0 \subseteq Q$ *is a set of initial states,*
- $F \subseteq Q$ *is a set of accepting states, called the acceptance set.*

An NBA accepts an infinite word $\sigma = a_0 a_1 a_2 ...$ by performing an infinite run $\rho = q_0 q_1 q_2 ...$ over the set Q of its states. The run ρ is said to be an accepting run if it can visit infinitely often (at least) one accepting state, i.e., it must hold that in a run $\rho = q_0 q_1 q_2 ...$, $q_i \in F$ for infinitely many indices $i \in \mathbb{N}$. The concept of NBA is used to define the acceptance condition of the TA model used in this thesis.

3.2. Timed Automata

It is now possible to introduce the formalism of TAs that, as will become clear in this section, can act as an acceptor of timed ω-languages. To obtain the TA model used in this thesis, two preliminary definitions of TAs are given. These definitions are adapted from [11]. Notice that Definition 3.3 is slightly different from the original one presented in [6].

Definition 3.3. *A* Timed Automaton $\mathcal{A}$ *is a tuple* $\mathcal{A} = (\Sigma, Q, q_0, B, X, T)$*, where:*

- Σ *is a finite input alphabet,*
- Q *is a finite set of control locations (also called states),*
- q_0 *is the initial location,*

- $B \subseteq Q$ *is a subset of control locations used as accepting locations,*
- X *is a set of clocks,*
- $T \subseteq Q \times Q \times \Gamma(X) \times \Sigma \times 2^X$ *is a transition relation.*

In Definition 3.3, each clock $x \in X$ can assume only positive real values belonging to the set $\mathbb{R}_{\geqslant 0}$. Clocks always have an initial default value equal to 0, from which they start to grow. The value a clock assumes in a given moment in time is defined by a function called clock valuation, i.e., a function $v : X \to \mathbb{R}_{\geqslant 0}$. A clock can therefore be thought of as a sort of variable handled by the TA, whose value can either be reset to 0 or grow boundlessly until a reset occurs (like a stopwatch).
An example of the evolution in time of a clock x is shown in Figure 3.1. Here, the labels on the x-axis indicate time instants in which the clock is reset: this entails the re-initialization of the clock's value to 0, from which it starts increasing again. Taking a closer look at this figure, at the beginning clock x is initialized having a value equal to 0. Then, its value is continuously incremented until a reset occurs: the first reset happens in the time instant t_0. At that moment in time, the value of x is instantly set to 0, from where it can start to increase again until a new reset is encountered.
$\Gamma(X)$ is the set of clock constraints $\gamma \in \Gamma(X)$ (also called clock guards) defined over X and specified by the following grammar: $\gamma := x < c \mid x = c \mid \neg\gamma \mid \gamma \wedge \gamma$, where $c \in \mathbb{N}$ is a positive natural constant and where $x \in X$ is a clock. Clock constraints appear in transitions and a transition can fire only if its clock constraints are met by all clock valuations for which the constraints are defined. When a transition fires, all clocks specified in the last element of its cartesian product (2^X) are reset to 0.
A couple (q, v), where $q \in Q$ and v is a clock valuation, is called a configuration for a given TA $\mathcal{A}$. A run $\rho = (q_0, v_0)(q_1, v_1)(q_2, v_2)...$ for $\mathcal{A}$ is an infinite sequence of configurations starting from the initial location q_0. Given a run ρ, $inf(\rho)$ is used to represent the locations visited infinitely often by $\mathcal{A}$, i.e., the set of locations $q \in Q$ such that $q = q_i$, for

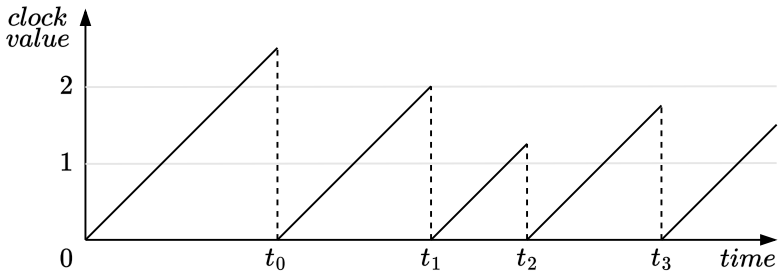

Figure 3.1: Representation of the evolution in time of the value for a given clock x. The x-axis represents time, while the y-axis represents the clock's value in a given time instant.

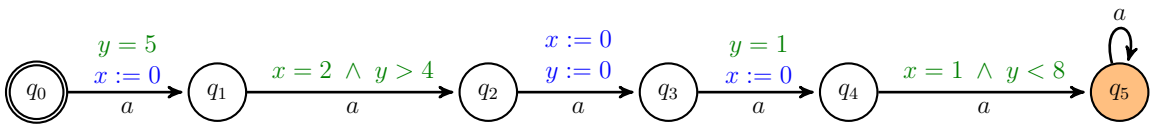

Figure 3.2: Example of TA. The input alphabet is $\Sigma = \{a\}$, the set of clocks is $X = \{x, y\}$. Clock constraints are represented in green; clock resets are represented in blue. The initial location is q_0 (represented with a double circle), while the (only) accepting location is q_5 (represented with a colored background).

infinitely many indices $i \in \mathbb{N}$. The set B of accepting locations denotes a Büchi acceptance condition. As specified in the previous section, this means that the given TA, to accept a timed w-language, must be able to visit locations specified in B infinitely often. More formally, the language of $\mathcal{A}$ is not empty iff $inf(\rho) \cap B \neq \varnothing$.

TAs can be conveniently represented as directed graphs, where nodes represent locations and where edges represent transitions between locations. In Figure 3.2, an example of TA is given. Starting from location q_0, the first transition of that TA can fire only when the value of clock y is exactly equal to 5. The second transition only requires clock x to be exactly equal to 2 since the condition on clock y is always satisfied (due to the first transition). Then, exiting from location q_2, both clocks are reset. A similar reasoning can be done for the fourth and fifth transitions, where in the latter the condition on clock y is always satisfied. Since there exist some clock valuations allowing the depicted TA to reach location q_5, in which it cycles forever, its language is not empty.

The TA model presented up to this point can characterize concrete real-time systems, i.e., systems whose specification is completely known at modeling time. The reality however is not always so straightforward: systems may be embedded in larger environments and some details may not be known when designing the model, or even after the model is deployed. To deal with these cases, the notion of TAs is extended with the concept of parameters, obtaining so-called parametric TAs. Parametric TAs first appeared in [20], along with some results on decidability problems.

Definition 3.4. *A* Parametric Timed Automaton *(also referred to as PTA)* $\mathcal{A}$ *is a tuple* $\mathcal{A} = (\Sigma, Q, q_0, B, X, T, P)$*, where* Σ*,* Q*,* q_0*,* B*,* X*,* T *are defined as in Definition 3.3 and where* P *is a set of parameters.*

The introduction of parameters has an impact on the structure of clock constraints. The set $\Gamma(X)$ now allows constraints of the form $\gamma := x < \mu$ and $\gamma := x = \mu$, where $\mu \in P$ is a parameter. The value a parameter can assume is determined by a mapping $\mathcal{I} : P \to \mathbb{R}$, meaning that a specific parameter is associated with a specific real-valued number.

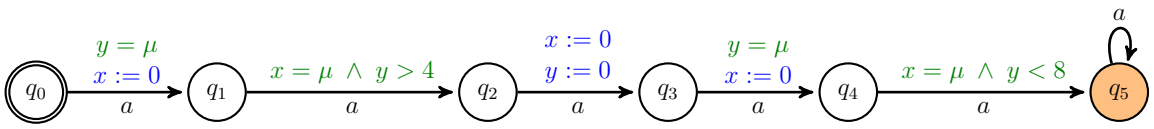

Figure 3.3: Example of PTA. The input alphabet is $\Sigma = \{a\}$, the set of clocks is $X = \{x, y\}$, the set of parameters is $P = \{\mu\}$. Clock constraints are represented in green; clock resets are represented in blue. The initial location is q_0 (represented with a double circle), while the (only) accepting location is q_5 (represented with a colored background).

$\mathcal{I}$ is called a parameter evaluation. For this reason, for a clock constraint $\gamma \in \Gamma(X)$ to be satisfied, both clock valuations and parameter evaluations must be considered. In this case, it is convenient to write $v, \mathcal{I} \models \gamma$ to denote that the constraint γ is satisfied by clock valuation v and parameter evaluation γ. In the context of PTAs, a run $\rho = (q_0, v_0)(q_1, v_1)(q_2, v_2)...$ is said to be a parametric run if clock constraints admit parameters and thus a parameter evaluation $\mathcal{I}$ must also be considered.

Figure 3.3, a revised version of Figure 3.2, represents an example of PTA. In this case, the first transition influences the execution of the second (e.g., if $\mu = 1$, the second transition would never fire, since clock x would be required to be exactly equal to 1 while clock y would be exactly equal to 2). A similar reasoning can be carried out for the fourth and fifth transitions. In this example, emptiness is subject to the value assumed by parameter μ: if the parameter evaluation $\mathcal{I}$ is such that $\mathcal{I}(\mu) \in (2, 4)$, then the language of the depicted PTA is not empty; otherwise, it would not be possible to reach the accepting location q_5.

The TA model studied in this thesis can now be formally introduced. In this model, it is forbidden to test and reset the same clock on the same transition: this property falls under the name of non-resetting test and TAs satisfying this property are called non-resetting test TAs [11].

Definition 3.5. *Let $\mathcal{A} = (\Sigma, Q, q_0, B, X, T)$ be a TA. For each transition $u \in T$ of $\mathcal{A}$ of the form $u = (q_u, q'_u, \gamma_u, a_u, S_u)$, let $X(\gamma_u)$ be the set of clocks that appear in constraint γ_u. Then, $\mathcal{A}$ is called a* non-resetting test Timed Automaton *(shortened as nrtTA) if the following holds: $\forall u \in T,\ X(\gamma_u) \cap S_u = \varnothing$.*

The notion of nrtTAs can be extended to obtain parametric nrtTAs, simply adding a set P of parameters to Definition 3.5. In the following, the considered nrtTA model has a singleton set as input alphabet, a set of clocks comprising two elements, and a singleton set of parameters. An example of such a parametric nrtTA was already given in Figure 3.3: notice that in the depicted example, in each transition either both clocks x, y are reset, or

both clocks x, y are tested, or one clock among x, y is tested and the other reset; hence, no transition allows clocks to be both tested and reset.

It was proven in [11] that for any (parametric) TA $\mathcal{A}$ whose set of clocks is X, there exists an equivalent (parametric) nrtTA $\mathcal{A}'$ whose set of clocks X' has size $|X'| = |X| + 1$. This implies that the results presented in the following chapters also apply to traditional PTAs (not satisfying the nrt property) having only one clock and one parameter. Lastly, the most important result, concerning the variant of parametric nrtTAs having two clocks and one parameter, is that it makes the emptiness problem decidable.

3.3. Regions and Zones

Having introduced the concept of TAs, a semantic characterization of this formalism can be done using Transition Systems (TSs) [5], a sort of Finite State Machines.
Given the infinite nature of words accepted by TAs, the underlying TS is composed of an infinite number of states, where a TS state is a couple (q, v) in which $q \in Q$ represents a state and v is a clock valuation. In practice, it is better to consider a finite version of such TS. It is possible to obtain a finite TS by introducing an equivalence relation between clock valuations. This equivalence relation considers both the integral and fractional part of clocks' values [6].

Definition 3.6. *Let $\mathcal{A}$ be a TA and, for each clock $x \in X$, let c_x be the largest integer constant with which clock x is compared. For each real value $t = \lfloor t \rfloor + frac(t) \in \mathbb{R}$, let $\lfloor t \rfloor$ denote its integral part and $frac(t)$ its fractional part. Then, two clock valuations v and v' are equivalent under the relation $\sim$ (denoted as $v \sim v'$) iff all the following holds:*

- *$\forall x \in X$, it either holds that $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$ or $v(x) > c_x \wedge v'(x) > c_x$,*
- *$\forall x, y \in X : v(x) \leqslant c_x \wedge v(y) \leqslant c_y$ it must hold that $frac(v(x)) \leqslant frac(v(y))$ iff $frac(v'(x)) \leqslant frac(v'(y))$,*
- *$\forall x \in X$ with $v(x) \leqslant c_x$, $frac(v(x)) = 0$ iff $frac(v'(x)) = 0$.*

A clock region *for $\mathcal{A}$ is an equivalence class induced by $\sim$ over the set of clock valuations.*

The notion of clock regions is used to define a Region Automaton (RA), essentially an FA able to mimic the behavior of the TA from which it is derived. A state of an RA is a couple (q, α), where $q \in Q$ is a state and α is a clock region. In particular, given a TA $\mathcal{A}$, its underlying RA can recognize the language $Untime(\mathcal{L}(\mathcal{A}))$, i.e., the language obtained by dropping the time sequence from the timed ω-language recognized by $\mathcal{A}$, hence considering only input alphabet symbols.

Emptiness checking of a given TA can therefore be carried out on the underlying RA since, if a timed ω-language $\mathcal{L}$ is timed regular (i.e., there exists a TA accepting it), then $\mathit{Untime}(\mathcal{L})$ is ω-regular (i.e., there exists an (N)BA accepting it).
Some considerations about the computational complexity of the emptiness checking carried out by relying on an RA are noteworthy. It was proven that the number of regions $\mathcal{R}$ is such that $[|X|! \cdot \Pi_{x \in X}(c_x)] \leqslant \mathcal{R} \leqslant [|X|! \cdot 2^{|X|} \cdot \Pi_{x \in X}(2c_x + 2)]$, where X is a set of clocks and c_x is the maximum integer constant with which clock $x \in X$ is compared. This causes the emptiness problem performed using regions to be PSPACE-complete [6].

Despite being an utterly important theoretical result, proving that the emptiness problem for TAs is decidable, the above technique based on regions is not practically feasible. For this reason, high-end tools working with TAs (like Uppaal or tChecker) use a different abstraction, which involves the concept of zones.
A *zone* over a set of clocks X is the set of clock valuations defined by a general clock constraint $\gamma \in \Gamma(X)$ [18]. Several operations can be defined over zones, including:

- Intersection of two zones,
- Reset of a zone given a subset $Y \subseteq X$ of clocks,
- Zone's future operation, which tells the temporal evolution of a zone.

The above operations preserve the nature of a given zone, i.e., after being applied to a given set of zones, the result will still be a zone. The concept of zones allows the introduction of Zone Graphs (ZGs) [21]. A ZG for a given TA $\mathcal{A}$ is a directed graph having as nodes couples of the form (q, Z), where $q \in Q$ is a location of $\mathcal{A}$ and Z is a zone. The number of zones is still infinite, hence after constructing the ZG, further abstractions need to be performed. It was proven that determining whether a TA $\mathcal{A}$ admits a Büchi acceptance condition, given a subsumption graph for $\mathcal{A}$ (i.e., a ZG that is further abstracted), is still PSPACE-complete, as in the case of regions. However, zones provide coarser (and hence smaller) abstractions with respect to regions, turning out to be more manageable in practice.

4 | Checking nrtTAs emptiness

This chapter focuses on the first main goal of this thesis: the development of a decision algorithm able to solve the emptiness problem for the parametric nrtTA model studied in this thesis. Starting with a short theoretical introduction about the algorithm, the chapter continues with a description of some of the tools available in TABEC. The chapter ends with a detailed description of how the algorithm has been implemented in the TABEC's checker tool.

4.1. Solving the nrtTAs emptiness problem

Before deriving a decision algorithm able to solve the emptiness problem for nrtTAs, recall that the considered nrtTA model is a parametric nrtTA presenting a Büchi acceptance condition, with a single parameter $\mu \in P$ and two clocks $x, y \in X$.
Let $\mathcal{A} = (\Sigma, Q, q_0, B, X, T, P)$ be a parametric nrtTA satisfying the aforementioned requirements and let C denote the biggest constant appearing in $\mathcal{A}$. It was proven in [11] that the emptiness problem for this nrtTA model is decidable.
The two main theoretical results which led to this conclusion are as follows:

- There exists a value $\Xi > 2C$ such that, $\forall \bar{\mu} \in \mathbb{R} : \bar{\mu} > 2C$, with $\bar{\mu} \neq \Xi$, if there is a parametric run ρ for $\mathcal{A}$ over a timed word (σ, τ) having a parameter evaluation $\mathcal{I}(\mu) = \bar{\mu}$, then there is also a parametric run $\hat{\rho}$ for $\mathcal{A}$ over a timed word $(\sigma, \hat{\tau})$ such that $\hat{\mathcal{I}}(\mu) = \Xi$ holds.
- There exists a value $0 < \alpha < \frac{1}{2}$ such that, $\forall n \in \mathbb{N}_{\geqslant 0} : n < 4C$, if there is a parametric run ρ for $\mathcal{A}$ over a timed word (σ, τ) having a parameter evaluation $\mathcal{I}(\mu) = \bar{\mu}$ (where $\frac{n}{2} < \bar{\mu} < \frac{n+1}{2}$), then there is also a parametric run $\hat{\rho}$ for $\mathcal{A}$ over a timed word $(\sigma, \hat{\tau})$ such that $\hat{\mathcal{I}}(\mu) = \hat{\mu} = \frac{n}{2} + \alpha$ holds.

From these results, the analysis of a parametric nrtTA can be split into two different scenarios: one in which $\mu > 2C$ holds and one in which $\mu \leqslant 2C$ holds. In particular, thanks to the first result, to check emptiness it is sufficient to replace parameter μ with a sufficiently large value (any value larger than $1 + C(1 + |Q|)$ is acceptable), and then

check the obtained non-parametric nrtTA. Similarly, it is possible to check all the cases in which μ is a multiple of $\frac{1}{2}$ and less than or equal to $2C$. The second result instead suggests that emptiness can be checked by replacing parameter μ with a value equal to $\frac{n}{2} + \alpha$, where α can be any value less than $\frac{1}{4(1+C \cdot max\{|Q|,4C\})}$ (this entails the value of α is always less than $\frac{1}{20}$), for every value $n < 4C$.
The fractional values mentioned above ($\frac{1}{2}$ in particular) are due to a characterization of clock values given using a region graph, where in this case regions are computed up to $2C$ and not up to C, as instead was done in [6].

From the above considerations, it is now possible to derive a decision algorithm that, receiving a parametric nrtTA $\mathcal{A}$ as input, outputs "true" if the language of $\mathcal{A}$ is not empty, hence admitting a Büchi acceptance condition, "false" otherwise.
A pseudocode version is shown in Algorithm 1. Here, two auxiliary procedures are leveraged: `SubstituteParameterValue()` (reported in Algorithm 2) is used to substitute every occurrence of parameter μ inside $\mathcal{A}$ with a value $\bar{\mu}$; this is necessary since the auxiliary procedure `CheckNonParametricEmptiness()` works only with non-parametric TAs. The latter procedure was not manually implemented, since tChecker already provides an implementation using zones. It is not mandatory to use the implementation provided by tChecker: other procedures may as well be created. For this reason, this procedure is considered a general black-box procedure, returning "true" if the non-parametric TA it receives as input admits a Büchi acceptance condition and "false" otherwise.
Algorithm 1 starts by considering the case in which the parameter's value may be greater than $2C$. In line 3 a suitable value $\mu > 2C$ (chosen as seen at the beginning of this section) is assigned to the parameter. In line 4 that value is substituted inside the parametric nrtTA given as input obtaining a non-parametric nrtTA, which emptiness is checked in line 5. Then, from line 7 to line 12, the analysis continues considering the case in which the parameter's value may be less than or equal to $2C$ and a multiple of $\frac{1}{2}$. Next, line 13 checks if after the previous loop, a positive acceptance condition is found. If a solution is still missing, the case in which the parameter's value may be less than $2C$ is tested from line 15 to line 21, assigning a suitable value to α as explained at the beginning of this section. Notice the $\vee$ operator applied in lines 11 and 20 to the result: this has the purpose of returning "true" even if only one acceptance condition is found, regardless if previous or subsequent results turn out to be "false". Lines from 22 to 25 return the obtained result.

It is now possible to make some observations about the computational complexity of Algorithm 1. These observations are based on a parametric nrtTA $\mathcal{A} = (\Sigma, Q, q_0, B, X, T, P)$. It is trivial to see that the `SubstituteParameterValue()` procedure has linear complex-

Algorithm 1: Emptiness check for parametric nrtTAs with 2 clocks and 1 parameter

Data: $(\mathcal{A})$: a parametric nrtTA with 2 clocks and 1 parameter.
Result: *True* if the language of $\mathcal{A}$ is not empty, *False* otherwise.

begin
  // Case in which the parameter is $\mu > 2C$
  $\bar{\mu} \leftarrow$ any value greater than $1 + C(1 + |Q|)$
  $\bar{\mathcal{A}} \leftarrow$ `SubstituteParameterValue`$(\mathcal{A}, \bar{\mu})$
  $isAccepting \leftarrow$ `CheckNonParametricEmptiness`$(\bar{\mathcal{A}})$
  // Now trying all the cases where $\mu \leqslant 2C$ *and* μ *is a multiple of* $\frac{1}{2}$
  $n \leftarrow 0$
  while $n \leqslant 2C$ **do**
    $\bar{\mu} \leftarrow n$
    $\bar{\mathcal{A}} \leftarrow$ `SubstituteParameterValue`$(\mathcal{A}, \bar{\mu})$
    $isAccepting \leftarrow isAccepting \vee$ `CheckNonParametricEmptiness`$(\bar{\mathcal{A}})$
    $n \leftarrow n + \frac{1}{2}$
  if $isAccepting == False$ **then**
    // Now trying all the cases where $\mu < 2C$ *using value* α
    $\alpha \leftarrow$ any value less than $\frac{1}{4(1+C \cdot max\{|Q|,4C\})}$
    $n \leftarrow 0$
    while $n < 4C$ **do**
      $\bar{\mu} \leftarrow \frac{n}{2} + \alpha$
      $\bar{\mathcal{A}} \leftarrow$ `SubstituteParameterValue`$(\mathcal{A}, \bar{\mu})$
      $isAccepting \leftarrow isAccepting \vee$ `CheckNonParametricEmptiness`$(\bar{\mathcal{A}})$
      $n \leftarrow n + 1$
  if $isAccepting == True$ **then**
    return $True$
  else
    return $False$

Algorithm 2: SubstituteParameterValue (used in Algorithm 1 and Algorithm 3)

Data: $(\mathcal{A}, \bar{\mu})$: a 1-parametric TA $\mathcal{A}$, a parameter value $\bar{\mu}$.
Result: A copy of $\mathcal{A}$ where each parameter occurrence μ is substituted with $\bar{\mu}$.

begin
  $P \leftarrow$ a copy of all parameter occurrences of $\mathcal{A}$
  for $\mu \in P$ **do**
    $\mu \leftarrow \bar{\mu}$
  return $\bar{\mathcal{A}}$ as a copy of $\mathcal{A}$ having P as new set of parameter occurrences

ity concerning the number of parameter occurrences contained in the clock guards of $\mathcal{A}$ due to the single for loop (copy operations in lines 2 and 5 are assumed to be linear as well). Let's denote the number of transitions of $\mathcal{A}$ as $\eta = |T|$ and let's assume that, for each transition $t \in T$, the number of parameter occurrences in the guard of t is bounded by the integer constant $\rho \in \mathbb{N}$.
It follows that the `SubstituteParameterValue()` procedure's complexity is $\mathcal{O}(\rho \cdot \eta)$. Next, since the `CheckNonParametricEmptiness()` procedure is considered as a black-box procedure, its complexity is denoted as $T_n(\bar{\mathcal{A}})$.
Finally, since the first while loop executes $4C + 1$ iterations whilst the second while loop executes $4C$ iterations, the overall worst-case scenario resulting complexity is:
$c + \mathcal{O}(\rho \cdot \eta) + T_n(\bar{\mathcal{A}}) + (4C + 1) \cdot \mathcal{O}(\rho \cdot \eta) + (4C + 1) \cdot T_n(\bar{\mathcal{A}}) + 4C \cdot \mathcal{O}(\rho \cdot \eta) + 4C \cdot T_n(\bar{\mathcal{A}}) =$
$c + (1 + 4C + 1 + 4C) \cdot \mathcal{O}(\rho \cdot \eta) + (1 + 4C + 1 + 4C) \cdot T_n(\bar{\mathcal{A}}) = c + (8C + 2) \cdot (\mathcal{O}(\rho \cdot \eta) + T_n(\bar{\mathcal{A}}))$,
where c is a constant accounting for any additional overhead or instructions not explicitly considered (such as assignments or comparisons).
The overall complexity is driven by $T_n(\bar{\mathcal{A}})$, since the emptiness problem is PSPACE-complete, either using regions [6] or zones [21]. This leads to the conclusion of Algorithm 1 being a PSPACE-complete algorithm as well.

It is possible to obtain a slightly improved version of Algorithm 1 by directly returning a positive answer as soon as a suitable value for the parameter is found. The pseudocode of this new version is reported in Algorithm 3. The only difference with respect to Algorithm 1 concerns the addition of checks on the obtained emptiness results. These checks (added in lines 6, 15, and 25) stop the algorithm execution if a positive Büchi acceptance condition is found. In addition, when calling the `CheckNonParametricEmptiness()` procedure, the $\vee$ operator is not present anymore since a positive result is returned immediately once found.
Despite being able to save some iterations in best-case scenarios, this improved version still exhibits the same complexity as the one found for Algorithm 1 in the worst case. Furthermore, Algorithm 1 is more general allowing, with trivial adjustments, to detect and save all the parameter's values which yield a positive or negative result. This may be useful both in debugging the implementation and in testing a given parametric nrtTA. In the actual implementation, both Algorithm 1 and Algorithm 3 are available.

Having seen an overview of the theory behind the decidability problem of the parametric nrtTAs emptiness and having provided a decision algorithm to solve it, the next section introduces TABEC, the tool where this algorithm has been practically implemented.

Algorithm 3: Improved version of Algorithm 1

Data: $(\mathcal{A})$: a parametric nrtTA with 2 clocks and 1 parameter.
Result: *True* if the language of $\mathcal{A}$ is not empty, *False* otherwise.

```
1  begin
2      // Case in which the parameter is μ > 2C
3      μ̄ ← any value greater than 1 + C(1 + |Q|)
4      Ā ← SubstituteParameterValue(A, μ̄)
5      isAccepting ← CheckNonParametricEmptiness(Ā)
6      if isAccepting == True then
7          return True
8      else
9          // Now trying all the cases where μ ⩽ 2C and μ is a multiple of 1/2
10         n ← 0
11         while n ⩽ 2C do
12             μ̄ ← n
13             Ā ← SubstituteParameterValue(A, μ̄)
14             isAccepting ← CheckNonParametricEmptiness(Ā)
15             if isAccepting == True then
16                 return True
17             n ← n + 1/2
18         // Now trying all the cases where μ < 2C using value α
19         α ← any value less than 1/(4(1+C·max{|Q|,4C}))
20         n ← 0
21         while n < 4C do
22             μ̄ ← n/2 + α
23             Ā ← SubstituteParameterValue(A, μ̄)
24             isAccepting ← CheckNonParametricEmptiness(Ā)
25             if isAccepting == True then
26                 return True
27             n ← n + 1
28     return False
```

4.2. Introducing TABEC

To automatically check the emptiness of a given parametric nrtTA, a specific tool has been created: TABEC. Born as a simple converter from Uppaal's syntax to tChecker's syntax, TABEC has become, at the time of writing, a tool able not only to perform this conversion task but also to allow users to manually build TAs, to check their emptiness, and to perform random testing on them.

The fundamental principle underpinning the development of TABEC was to provide an academic tool that could be modified as necessary, to experiment with its functionalities and even extend it with new ones. For this reason, dependency on external libraries and tools was reduced as much as possible.
TABEC is a command-line tool written in C++17, occasionally using some Bash scripts to interact with auxiliary external tools and to collect results. It has been developed under a Unix environment and, for this reason, may work only with Unix machines. It has been tested on Linux and MacOS. Windows users may run it inside a Virtual Machine or in a Linux Subsystem, otherwise, the tool may not work properly.
TABEC presents a collection of four tools: the converter (which translates a TA description using Uppaal's syntax into a description using tChecker's syntax), the grapher (which outputs a PDF version of a given TA), the checker (which, by relying on tChecker, determines if the language of a TA is empty or not), and the tester (which performs random testing on TAs, either manually or automatically built). Notice that it is not mandatory to use parametric nrtTAs in TABEC: this is only a particular case since it can work with traditional TAs as well. Each of these tools comes with a set of options, specified as command-line arguments.
The remainder of this section gives a brief overview of the tools useful for performing emptiness checking of TAs with TABEC.

4.2.1. The converter

To be able to perform emptiness checking on a TA, TABEC requires a description of it. Since TABEC is a command-line tool, manually building TAs using a textual representation would be a difficult task. For this reason, it is possible to create TAs descriptions using Uppaal's System Editor and import these descriptions inside TABEC, which then automatically translates them into a format suitable for tChecker.
An example of a TA declared in Uppaal and accepted by TABEC is given in Figure 4.1. Notice that the obtained model does not strictly follow Uppaal's syntax; this is purposely done to specify all the needed details in TABEC which are not available in Uppaal. In particular, in that figure a double circle represents an initial location, a colored circle represents an accepting location, clock guards are specified in green, resets are specified in blue, and the keyword "param" is used to denote the presence of a parameter. Also, clocks x and y need to be specified inside Uppaal's template declarations using the conventional Uppaal's syntax. The grey initialization for clock y depicted on the left means that this clock must have a value equal to 0 before entering the TA, while the "in" and "out" keywords are used to connect TAs: their use will become clear when the concepts

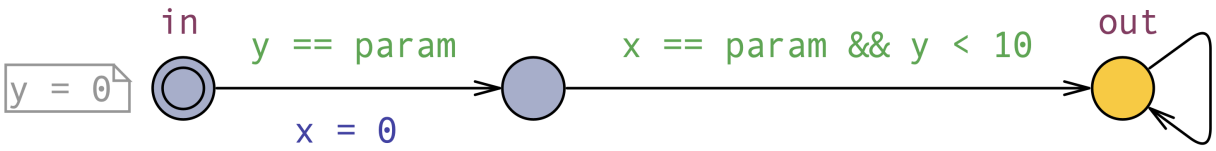

Figure 4.1: Example of TA created in Uppaal.

of tiles and Tiled TAs will be introduced in Section 5.1.
Once the TA model is completed, it can be saved in an `.xml` file. When reading that `.xml` file, the converter first translates it into a JSON intermediate representation. This is done to allow easy manipulation of the given TAs. Then, the JSON intermediate representation is translated into a `.tck` file (the proprietary format used by tChecker). The resulting file can now be either given manually to tChecker or used by other TABEC's tools. If given to tChecker, results about emptiness and resource utilization are obtained. A graphical representation of this sequence of steps is given in Figure 4.2.
The converter can be called using the following shell command:

```
$ ./converter -src /path/to/input/dir -dst /path/to/output/dir -nrt -jsn
```

The meaning of the options used in this command is the following:

- `-src /path/to/input/dir` specifies the input directory from which to take `.xml` files. If omitted, the default input directory embedded in TABEC is used.
- `-dst /path/to/output/dir` specifies the output directory where `.tck` files are written. If omitted, the default output directory embedded in TABEC is used.
- `-nrt` translates only TAs satisfying the nrt property.
- `-jsn` prints the JSON intermediate representation of the given TAs on screen.

It is worth noticing that in every TABEC's tool, command-line options can be specified in any order. The only restriction is that options' arguments need to be specified right after the relative option (e.g., when specifying the input directory path). Command-line options can also be omitted.

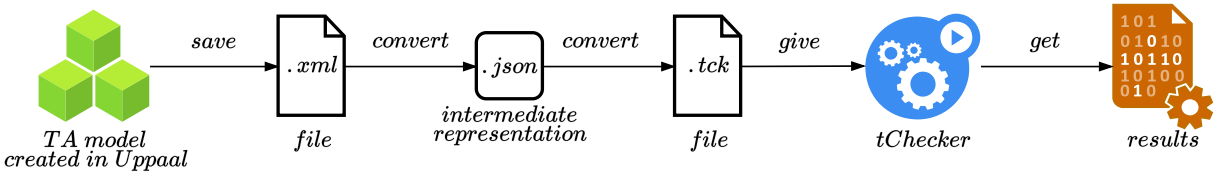

Figure 4.2: Sequence of steps performed to gather results when using the converter tool.

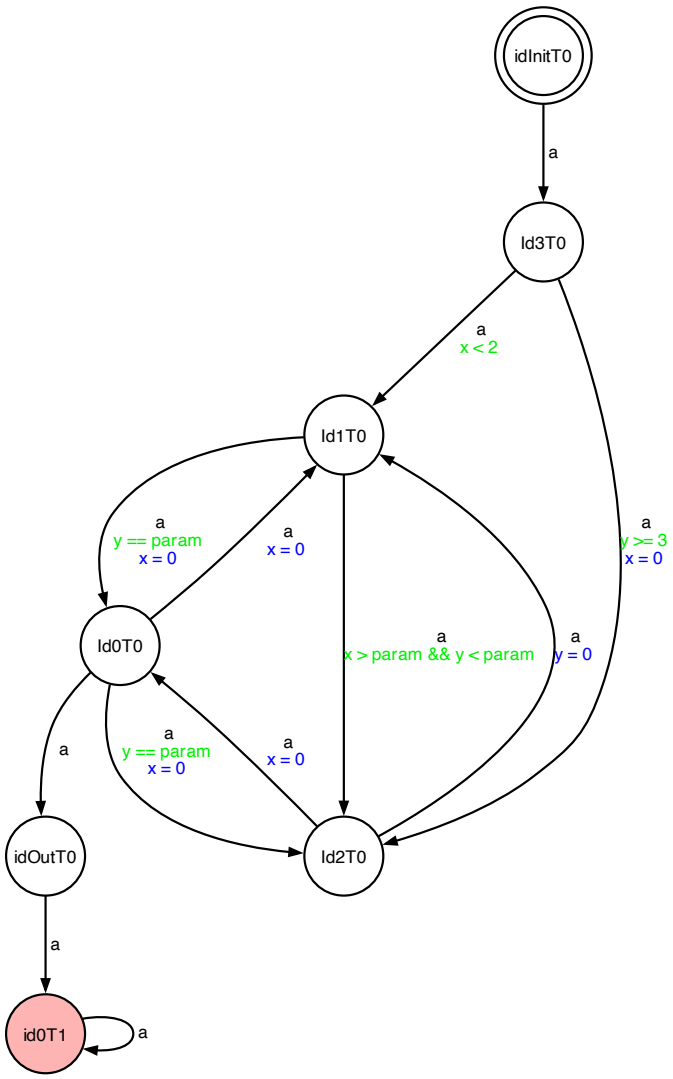

Figure 4.3: Example of a parametric nrtTA generated by TABEC and subsequently transformed in a `.pdf` file by the grapher tool.

4.2.2. The grapher

The ability to visually analyze TAs, either given to or generated by TABEC, is fundamental, especially for debugging a given TA description. To generate a visual representation of a TA, the grapher tool can be used.
This tool takes an `.xml` representation of a TA as input (typically obtained from Uppaal) and generates a `.dot` file as output which, thanks to the open-source graph visualization software Graphviz, is converted into a more understandable `.pdf` representation.
An example of a `.pdf` file the grapher can generate is given in Figure 4.3. Notice that the same graphical conventions are used with respect to the examples shown in the previous chapter (e.g., colors and double circles).

It is possible to run the grapher tool with two different behaviors. The first one is obtained using the following shell command:

```
$ ./grapher -src /path/to/input/dir
```

The meaning of the options used in this command is the following:

- `-src /path/to/input/dir` specifies the input directory from which to take `.xml` files. If omitted, the default input directory embedded in TABEC is used.

When this call to the grapher is invoked, all the `.xml` files contained in the specified input directory are transformed in `.pdf` files inside the default `.pdf` output directory embedded in TABEC. Another possibility consists of calling the grapher as follows:

```
$ ./grapher -rfd
```

The meaning of the options used in this command is the following:

- `-rfd` if set, the grapher only translates all `.dot` files contained in the dedicated `.dot` directory embedded in TABEC.

This latter execution modality is particularly useful when no `.xml` file is provided. A typical scenario in which this happens is when TAs are randomly generated.

4.2.3. The checker

Once a TA model is converted in the `.tck` format, its emptiness can be checked: this is the responsibility of the checker tool. This subsection only provides some hints about the checker tool. Section 4.3 contains a more detailed description of its implementation.
As mentioned in Section 2.3, tChecker is used as an auxiliary tool since it already provides an algorithm implementation for TAs emptiness checking, available in the tck-liveness tool. From a high-level point of view, the checker tool gives to tck-liveness the desired TA for which emptiness must be checked and, once the computation has finished, gathers the results obtained by tck-liveness, presenting them to the user.
A consideration has to be made regarding the presence of parameters: tChecker is not designed to work with parametric TAs, hence the `.tck` files it accepts must contain a non-parametric TA description.
The checker tool can be run by executing the following shell command:

```
$ ./checker -dst /path/to/tcks/dir -lns /path/to/tck-liveness/dir -cls -all
```

The meaning of the options used in this command is the following:

- `-dst /path/to/tcks/dir` path to the directory where `.tck` files are taken as input. If omitted, the default `.tck` files input directory embedded in TABEC is used.
- `-lns /path/to/tck-liveness/dir` path to the directory in which the tck-liveness executable is contained (in tChecker this corresponds to the bin folder). If omitted, the path specified during the build of TABEC is used.
- `-cls` clears directories used to store computation results while executing the checker.
- `-all` the computation is not terminated when the first parameter's value leading to a Büchi acceptance condition is found, but it continues to check all the other possible parameter's values, as specified in Algorithm 1.

4.3. Checking emptiness with TABEC

The TABEC's checker tool is able, given a suitable `.tck` description of a parametric nrtTA $\mathcal{A}$, by relying on tChecker, to provide an answer about the emptiness problem for $\mathcal{A}$. It is worth recalling that with the checker tool it is not mandatory to work only with parametric nrtTAs, but traditional TAs are admitted as well.

The checker tool has been implemented as a combination of C++ code and Bash scripts, the latter being used to communicate with the tChecker's tck-liveness tool, both for giving inputs and for gathering outputs. Bash scripts represent the core of the checker tool since they contain an implementation of the algorithms introduced in Section 4.1. On the other hand, C++ code is used to orchestrate the execution of such scripts and to write logs in `.txt` files containing the results of the execution.

A slightly modified version of Algorithm 3 has been implemented, where the algorithm stops at the first positive answer returning only one parameter's value, otherwise, thanks to the command-line option `-all`, it continues trying all possible parameter's values. Since tChecker works only with non-parametric TAs, Bash scripts also contain an implementation of Algorithm 2 to substitute the keyword "param" with a suitable value $\bar{\mu} \in \mathbb{R}$. This is done by scanning the given `.tck` files, substituting each "param" keyword occurrence when found. Furthermore, when considering the case in which the parameter μ may be less than or equal to $2C$, since tChecker cannot work with fractional values appearing in clock guards, all the values contained in the `.tck` representation are multiplied by an integer factor to get only integer values.

A consideration must be made about the input provided to tChecker. The semantics

of TAs in tChecker does not consider time as strictly monotonic, hence two consecutive transitions may occur at the same time. This requires an adaption, since the semantics of nrtTAs is instead strictly monotonic.
Given a TA $\mathcal{A}$ and a timed word (σ, τ), to force strict monotonicity of the time sequence τ, an additional clock z is added to the set of clocks $X = \{x, y\}$ of $\mathcal{A}$, hence obtaining a TA $\mathcal{A}'$ having as a set of clocks $X' = \{x, y, z\}$. The new clock z is reset on every transition entering a location and introduces an additional guard $(z > 0)$ on every transition exiting from a location. Since clock z is never compared with constants (apart from 0) nor parameters, its presence does not affect the nature of the language accepted by $\mathcal{A}$; in other words, language equivalence holds between $\mathcal{A}$ and $\mathcal{A}'$, i.e., $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$. For this reason, it is permissible to use this slightly modified TA model which uses three clocks, always keeping in consideration that the model under study (i.e., a parametric nrtTA with two clocks, one parameter, and subject to a Büchi acceptance condition) is not affected by this change and thus the obtained results coincide.

There are four Bash scripts in total:

- `gt2C.sh` script considering the case in which parameter μ may be greater than $2C$.
- `lt2CScale.sh` script used to obtain only integer values inside clock guards.
- `lt2CCycle.sh` script testing the cases where the parameter μ may be less than or equal to $2C$ and a multiple of $\frac{1}{2}$ and where the parameter μ may be less than $2C$.
- `tCheckerLiveness.sh` script used to call the tck-liveness tool and to gather results back (used both in `gt2C.sh` and `lt2CCycle.sh` scripts). This script also measures information regarding the execution time of tChecker.

To provide an overview of how the checker tool calls these scripts, a sequence diagram is given in Figure 4.4. In this example, it is assumed that the command-line option `-all` is not provided: this corresponds to a representation of the execution of Algorithm 3. As shown in this figure, the checker tool starts by invoking the `gt2C.sh` script which, after obtaining a non-parametric version of the TA given as input, calls the `tCheckerLiveness.sh` script to check the case in which the parameter may be greater than $2C$. This script, relying on the tck-liveness tool, gathers emptiness results and keeps track of resource utilization. The obtained emptiness results are then given back to the checker: if there is a positive Büchi acceptance condition, the execution stops, otherwise the `lt2CScale.sh` script is called to obtain only integer constants in the input TA. Then, the `lt2CCycle.sh` script performs emptiness checks for both cases where the parameter may be less than or equal to $2C$ and a multiple of $\frac{1}{2}$ and where the parameter may be less than $2C$. Here, the

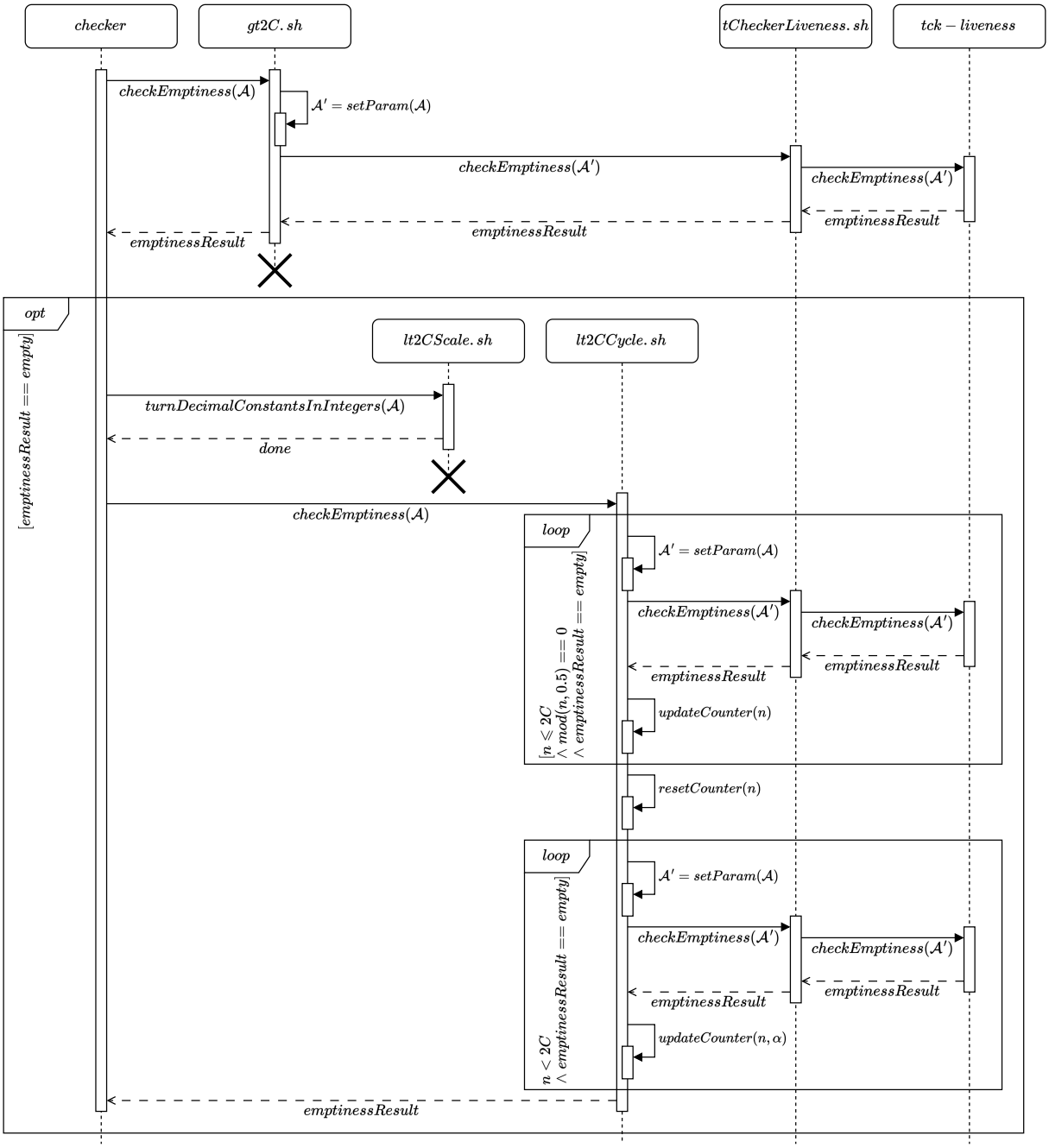

Figure 4.4: Sequence diagram showing the invocation order of Bash scripts used in the checker tool. Scripts are denoted with a `.sh` extension. The `mod()` (modulo) function is used to determine values of n which are multiples of $\frac{1}{2}$. The value α used in the second loop is the same as the one described in Section 4.1.

`tCheckerLiveness.sh` script and the tck-liveness tool are used as well. The condition on loop cycles performed by the `lt2CCycle.sh` script ensures to stop its execution as soon as a positive Büchi acceptance condition is detected. At the end, the obtained results are given back to the checker.

It is still necessary to analyze how the tck-liveness tool can execute the verification of emptiness, providing a positive answer if the given TA admits a Büchi acceptance condition and a false answer otherwise. The tck-liveness tool provides two decision algorithms for TAs emptiness checking, one using a nested Depth First Search (DFS) approach, the other one based on the concept of SCCs.
Let $\mathcal{G}$ be the graph used to check Büchi emptiness (e.g., obtained by using the concept of regions or zones). The DFS algorithm directly looks for accepting cycles in $\mathcal{G}$, requiring little auxiliary memory. The SCC-based algorithm identifies SCCs containing accepting cycles in $\mathcal{G}$, requiring more auxiliary memory but being able to find counterexamples more quickly: this has been implemented as an enhanced version of the Couvreur's SCC-decomposition-based algorithm.
Both these algorithms are classified as explicit algorithms, i.e., they construct and explore states of $\mathcal{G}$ one by one, allowing to solve the Büchi emptiness problem in $\mathcal{O}(n)$ time, where n is the number of needed states. It is however necessary to recall that n doesn't grow linearly (e.g., as reported in Section 3.3, the number of regions $\mathcal{R}$ is such that $[|X|! \cdot \Pi_{x \in X}(c_x)] \leqslant \mathcal{R} \leqslant [|X|! \cdot 2^{|X|} \cdot \Pi_{x \in X}(2c_x + 2)]$). A more detailed description of these algorithms, along with a pseudocode representation, can be found in [22].
The SCC-based algorithm is the one invoked by the `tCheckerLiveness.sh` script. Indeed, this algorithm can work with TAs exhibiting a generalized Büchi acceptance condition, this being compliant with the parametric nrtTA model used in this thesis.

5 | Random testing with TABEC

This chapter focuses on the second main goal of this thesis: the ability to perform random testing with oracle forecasting when dealing with TAs. The chapter starts by giving a theoretical introduction to the concepts of tiles and Tiled TAs, showing how these formalisms can be constructed. It then continues by giving an in-depth description of the tester tool and its implementation, also focusing on how tiles and Tiled TAs can be used to enrich TABEC with oracle capabilities.

5.1. Tiles and Tiled Timed Automata

Tiles have been introduced since it can be proven that, when working with parameters, they're able to constrain the interval in which the parameters' value can fall, a remarkable result heavily used in random testing with TABEC. This effect on the parameters' value is obtained simply by connecting a given set of tiles together. This section offers an introduction to the concepts of tiles and Tiled TAs, along with some practical examples.

5.1.1. Basic theoretical concepts about tiles

Tiles are particular TAs that can be considered as building blocks which, once combined, can generate more complex TAs. More formally:

Definition 5.1. *A* tile $\mathcal{T}$ *is a tuple* $\mathcal{T} = (\Sigma, Q, q_0, B, X, T, In, Out)$*, where* Σ*,* Q*,* q_0*,* B*,* X*,* T *are defined as in Definition 3.3 and where:*

- $In \subseteq Q$ *is the set of input locations,*
- $Out \subseteq Q$ *is the set of output locations,*
- q_0 *may not be defined,*
- B *may be empty.*

Input and output locations are used to connect tiles (e.g., connecting the output locations of a tile with the input locations of another tile). When tiles are connected without

generating cycles among them, they are said to be connected in sequence:

Definition 5.2. *Given a set of n tiles* $\Theta = \{\mathcal{T}_0, \mathcal{T}_1, ..., \mathcal{T}_{n-1}\}$*, let* $\Sigma_\Theta = \bigcup_{\mathcal{T}\in\Theta} \Sigma_\mathcal{T}$ *and* $X_\Theta = \bigcup_{\mathcal{T}\in\Theta} X_\mathcal{T}$*. Tiles contained in* Θ *are said to be* connected in sequence *if transitions between different tiles are of the form:* $Out_{\mathcal{T}_i} \times In_{\mathcal{T}_{i+1}} \times \Gamma(X_\Theta) \times \Sigma_\Theta \times 2^{X_\Theta}$*,* $\forall i \in [0, n-2]$*.*

To have significant tiles, from now on, without loss of generality, given a tile $\mathcal{T}$ its sets *In* and *Out* are assumed to be disjoint, i.e., $In \cap Out = \varnothing$. It is worth emphasizing again that, by Definition 5.1, a given tile may not have an initial location and/or accepting locations. The former condition is formalized as: $\{q_0\} = \varnothing$, while the latter as: $B = \varnothing$. For this reason, the following two definitions are introduced:

Definition 5.3. *A* tile *is called an* initial tile *if it has one and only one initial location.*

Definition 5.4. *A* tile *is called an* accepting tile *if it has one or more accepting locations.*

It is trivial to extend the notion of tiles with the concept of parameters:

Definition 5.5. *A* tile $\mathcal{T} = (\Sigma, Q, q_0, B, X, T, P, In, Out)$ *is called a* parametric tile*, where* Σ*,* Q*,* q_0*,* B*,* X*,* T*,* In*,* Out *are defined as in Definition 5.1 and where* P *is a set of parameters, if it forces each parameter* $\mu \in P$ *to be inside an interval* $\mathcal{I}_\mathcal{T}$ *such that* $\mathcal{I}_\mathcal{T} = \{w \in \mathbb{R}_{\geqslant 0} \mid a \sim w \sim b \wedge a \leqslant b\}$*, where* $a \in \mathbb{R}_{\geqslant 0}$*,* $b \in (\mathbb{R}_{\geqslant 0} \cup \{\infty\})$ *and* $\sim \in \{<, \leqslant\}$*.*

Having understood what tiles are, it is now possible to introduce Tiled TAs. Since tiles are used to force the parameters' value interval, only the definition of parametric Tiled TAs is given (the definition of non-parametric Tiled TAs can be obtained from the following one by using only non-parametric tiles and by dropping the set Π of parameters).

Definition 5.6. *A* parametric Tiled Timed Automaton *(also shortened as parametric TTA)* $\mathcal{A}$ *is a tuple* $\mathcal{A} = (\Theta, \mathcal{T}_0, \Xi, \mathcal{B}, \mathcal{X}, \Pi, \Upsilon)$*, where:*

- Θ *is a set of parametric tiles,*
- $\mathcal{T}_0 \in \Theta$ *is the initial tile,*
- Ξ *is a finite input alphabet defined as:* $\Xi = \bigcup_{\mathcal{T}\in\Theta} \Sigma_\mathcal{T}$*,*
- $\mathcal{B} \subseteq \Theta$ *is a set of accepting tiles,*
- $\mathcal{X}$ *is a set of clocks defined as:* $\mathcal{X} = \bigcup_{\mathcal{T}\in\Theta} X_\mathcal{T}$*,*
- Π *is a set of parameters defined as:* $\Pi = \bigcup_{\mathcal{T}\in\Theta} P_\mathcal{T}$*,*
- $\Upsilon \subseteq \Theta \times \Theta \times \Xi \times 2^\mathcal{X}$ *is a transition relation.*

The language accepted by a parametric TTA $\mathcal{A}$ is denoted as $\mathcal{L}(\mathcal{A})$. In Definition 5.6, the last element appearing in the cartesian product of a transition ($2^{\mathcal{X}}$) denotes the set of clocks that are reset over such transition. A parametric TTA having a set of parameters such that $|\Pi| = 1$ is called a 1-parametric TTA.

Since the focus of this thesis is on parametric nrtTAs with two clocks and one parameter, from now on only tiles satisfying the nrt property having $|X| = 2$ and $|P| = 1$ will be considered. Henceforth, the term *tiles* will be used to denote this class of tiles.

In the following it is assumed that, for a given parametric TTA $\mathcal{A} = (\Theta, \mathcal{T}_0, \Xi, \mathcal{B}, \mathcal{X}, \Pi, \Upsilon)$, each tile $\mathcal{T} \in \Theta$ has the same set $X = \{x, y\}$ of clocks (hence $\mathcal{X} = X = \{x, y\}$), each tile $\mathcal{T} \in \Theta$ has the same set of parameters $P = \{\mu\}$ (hence $\Pi = P = \{\mu\}$), and each tile $\mathcal{T} \in \Theta$ has the same input alphabet $\Sigma = \{a\}$ (hence $\Xi = \Sigma = \{a\}$).

For a tile $\mathcal{T}$ to force its parameter's value inside an interval, clocks must have particular initial values before a run $\widetilde{\rho} = (q_{<0,\mathcal{T}_0>}, v_{<0,\mathcal{T}_0>})(q_{<1,\mathcal{T}_0>}, v_{<1,\mathcal{T}_0>})...(q_{<n,\mathcal{T}>}, v_{<n,\mathcal{T}>})...$ over $\mathcal{A}$ enters $\mathcal{T}$. These values are referred to as clock preconditions. Notice that, since the run $\widetilde{\rho}$ is defined over a parametric TTA, locations q and clock valuations v contained in $\widetilde{\rho}$ are enriched with the name of the currently visited tile.
The clock precondition function $\mathcal{P}_{\mathcal{T}} : X \to \mathbb{R}_{\geqslant 0}$, when applied to a clock $x \in X$ for tile $\mathcal{T}$, tells the value clock x must assume for its clock precondition to hold. Hence, when connecting tiles, preconditions must be respected: as will be clear later in this chapter, for the considered tiles in this thesis, clock preconditions are always satisfied by inserting clock resets on transitions used to connect tiles.

An example of parametric TTA is given in Figure 5.1. The depicted parametric TTA is obtained by connecting four tiles in sequence. Its execution starts in tile $\mathcal{T}_0$. After exiting from the initial tile, it enters the second tile $\mathcal{T}_1$, then the third tile $\mathcal{T}_2$, and lastly the fourth tile $\mathcal{T}_3$. The latter tile is an accepting tile meaning that, for the parametric TTA's language to be non-empty, a Büchi acceptance condition must hold inside this tile. Notice also that, before entering a tile, clock values must be set as specified by the clock precondition functions reported over each transition.

Although Definition 5.6 allows the construction of generic parametric TTAs, in this thesis the considered model is a slightly restricted version, which does not allow cycles between different tiles (cycles can still be defined inside tiles themselves), resulting in parametric TTAs structured as trees. This is done since it can be proven that the connection of tiles in a sequence can force the parameter's value inside the intersection of all the intervals provided by each tile, as stated in the following lemma.

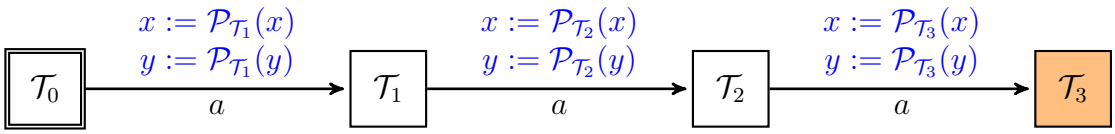

Figure 5.1: Example of parametric TTA obtained by connecting four tiles in sequence. The input alphabet is $\Xi = \{a\}$, the set of clocks is $\mathcal{X} = \{x, y\}$, the set of parameters is $\Pi = \{\mu\}$. The initial tile $\mathcal{T}_0$ is represented with a double square, and the accepting tile $\mathcal{T}_3$ is represented with a colored background. Clock preconditions are represented in blue. On each transition, before entering a tile $\mathcal{T}_i$, both clocks x, y are set to the values specified by the $\mathcal{T}_i$'s clock precondition function $\mathcal{P}_{\mathcal{T}_i}$.

Lemma 5.1. *Let $\mathcal{A}_n = (\Theta, \mathcal{T}_0, \Xi, \mathcal{B}, \mathcal{X}, \Pi, \Upsilon)$ be a 1-parametric TTA obtained by connecting in sequence a set $\Theta = \{\mathcal{T}_0, \mathcal{T}_1, ..., \mathcal{T}_{n-1}\}$ of n tiles, $\forall n \in \mathbb{N}_{>0}$, where $\mathcal{B} = \{\mathcal{T}_{n-1}\}$.*
It then follows that: $\mathcal{L}(\mathcal{A}_n) \neq \varnothing \wedge (\bigcap_{i=0}^{n-1} \mathcal{I}_{\mathcal{T}_i} \neq \varnothing) \Rightarrow \mu \in \bigcap_{i=0}^{n-1} \mathcal{I}_{\mathcal{T}_i},\ \mu \in \Pi$.

Proof. In the following, it is assumed that $\mathcal{L}(\mathcal{A}_n) \neq \varnothing \wedge \bigcap_{i=0}^{n-1} \mathcal{I}_{\mathcal{T}_i} \neq \varnothing$, $\mathcal{T}_i \in \Theta$ holds, i.e., the intersection of all the intervals $\mathcal{I}_{\mathcal{T}_i}$, $\forall i \in [0, n-1]$, is not empty, otherwise $\mu \in \bigcap_{i=0}^{n-1} \mathcal{I}_{\mathcal{T}_i}$, $\mathcal{T}_i \in \Theta$ would trivially hold, due to the logic equivalence $a \Rightarrow b \equiv \neg a \vee b$. The proof is then conducted by induction over the number n of tiles contained in the set Θ of $\mathcal{A}_n$.
Base case: it must be proven that the statement holds for $n = 1$, i.e., $\Theta = \{\mathcal{T}_0\}$, $\mathcal{B} = \{\mathcal{T}_0\}$. Since here Θ is composed of only one tile, a single interval $\mathcal{I}_{\mathcal{T}_0}$ has to be considered. The fact that $\mu \in \mathcal{I}_{\mathcal{T}_0}$ directly follows from the definition of parametric tiles.
Induction step: assuming that the statement holds for a generic value $n \in \mathbb{N}_{>0}$, i.e., when considering a set of tiles $\Theta = \{\mathcal{T}_0, \mathcal{T}_1, ..., \mathcal{T}_{n-1}\}$, it will be proven that it also holds for $n+1$, i.e., when considering a parametric TTA $\mathcal{A}_{n+1}$ with a set of tiles $\Theta' = \{\mathcal{T}_0, \mathcal{T}_1, ..., \mathcal{T}_{n-1}, \mathcal{T}_n\}$ and a set $\mathcal{B}' = \{\mathcal{T}_n\}$. In order for $\mathcal{L}(\mathcal{A}_{n+1})$ to be non-empty, tile $\mathcal{T}_n$ must be reached by a run $\tilde{\rho}$ traversing all tiles composing $\mathcal{A}_{n+1}$.
Let $\mathcal{I}_\Theta$ be the interval obtained by intersecting all the intervals forced by tiles contained in Θ, i.e., $\mathcal{I}_\Theta = \bigcap_{i=0}^{n-1} \mathcal{I}_{\mathcal{T}_i}$, $\mathcal{T}_i \in \Theta$. Since Θ' can be written as $\Theta' = (\Theta \cup \mathcal{T}_n)$, the intersection of all the intervals forced by its tiles is $\mathcal{I}_{\Theta'} = (\mathcal{I}_\Theta \cap \mathcal{I}_{\mathcal{T}_n})$. Since by induction hypothesis it holds that $\mu \in \mathcal{I}_\Theta$ and since, by construction, it holds that $\mu \in \mathcal{I}_{\mathcal{T}_n}$, parameter μ must satisfy the following formula: $\varphi := \mu \in \mathcal{I}_\Theta \wedge \mu \in \mathcal{I}_{\mathcal{T}_n}$. Recalling the assumption made at the beginning of the proof (in the case of $n+1$ tiles rewritten as $\mathcal{L}(\mathcal{A}_{n+1}) \neq \varnothing \wedge \bigcap_{i=0}^{n} \mathcal{I}_{\mathcal{T}_i} \neq \varnothing$, $\mathcal{T}_i \in \Theta'$), the satisfaction of the previous formula φ is possible only if μ belongs to the intersection of $\mathcal{I}_\Theta$ and $\mathcal{I}_{\mathcal{T}_n}$, i.e., it must hold that $\mu \in \mathcal{I}_{\Theta'} = \bigcap_{i=0}^{n} \mathcal{I}_{\mathcal{T}_i}$, $\mathcal{T}_i \in \Theta'$. □

By the above lemma and the fact that the considered parametric TTA model exhibits a tree structure, TABEC can be enriched with oracle capabilities on the parameter's value.

5.1.2. Elementary tiles and examples

Before continuing the discussion on tiles, one additional definition is still needed, since it introduces a concept used for creating tiles able to force arbitrary intervals.

Definition 5.7. *Two parametric tiles $\mathcal{T}_1$ and $\mathcal{T}_2$ are called* elementary tiles *if, when connected in sequence, they can generate any arbitrary interval in which to constrain the parameter's value.*

In the following, without loss of generality, only open intervals will be considered (to obtain closed intervals, trivial adjustments can be made on tiles). A clarification about the arbitrariness of intervals that can be created is necessary. Due to the parametric nrtTA model under study having two clocks, and due to the region characterization of such model as carried out in [11], the finest interval granularity that can be created is at multiples of $\frac{1}{2}$, i.e., the most general interval that can be created is of the form $(\frac{a}{2}, \frac{b}{2})$, where $a \in \mathbb{N}$, $b \in (\mathbb{N}_{>0} \cup \{\infty\})$, and $a < b$.

The following propositions give two examples of elementary tiles. The integer value n introduced in these examples must not be confused with a parameter: its purpose is to generalize the interval that can be generated with a particular tile. In practice, n must be substituted by an appropriate value when creating a tile.

Proposition 5.1. *The tile $\mathcal{T}$ defined as $\mathcal{T} = (\Sigma = \{a\}, Q = \{q_1, q_2, q_3\}, \{q_0\} = \varnothing, B = \varnothing, X = \{x, y\}, T = \{(q_1, q_2, y = \mu, a, x), (q_2, q_3, x = \mu \wedge y > n, a, \varnothing)\}, P = \{\mu\}, In = \{q_1\}, Out = \{q_3\})$ is able to force the parameter inside the interval $(\frac{n}{2}, \infty)$, with $n \in \mathbb{N}$.*

Proof. The proof is conducted by analyzing the behavior of $\mathcal{T}$, reported for convenience in Figure 5.2a. For the first transition to fire, clock y is required to have a value equal to the parameter μ. After the first transition fires, clock x is reset to 0. This means that, in location q_2, by applying the clock valuation function to both clocks x, y, it holds that $v(x) = 0$ and $v(y) = \mu$. For the second transition to fire, clock x is required to have a value equal to the parameter μ, while clock y is required to be greater than n (notice that clock y is also equal to $\mu + \mu = 2\mu$). This means that the following inequality must be

(a) Elementary tile forcing the interval $(\frac{n}{2}, \infty)$ (b) Elementary tile forcing the interval $(0, \frac{n}{2})$

Figure 5.2: Example of elementary tiles used in Proposition 5.1 and in Proposition 5.2.

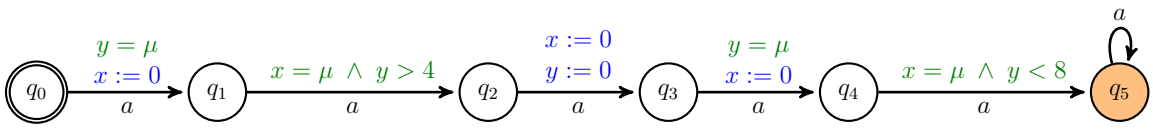

Figure 5.3: Tile forcing the interval $(2, 4)$. This is a case of a tile being both an initial tile (thanks to location q_0) and an accepting tile (thanks to location q_5).

satisfied for the second transition to fire: $y = 2\mu > n$, i.e., $\mu > \frac{n}{2}$. □

Proposition 5.2. *The tile $\mathcal{T}$ defined as $\mathcal{T} = (\Sigma = \{a\}, Q = \{q_1, q_2, q_3\}, \{q_0\} = \varnothing, B = \varnothing, X = \{x, y\}, T = \{(q_1, q_2, y = \mu, a, x), (q_2, q_3, x = \mu \wedge y < n, a, \varnothing)\}, P = \{\mu\}, In = \{q_1\}, Out = \{q_3\})$ is able to force the parameter inside the interval $(0, \frac{n}{2})$, with $n \in \mathbb{N}_{>0}$.*

Proof. The proof is conducted by analyzing the behavior of $\mathcal{T}$, reported for convenience in Figure 5.2b. The considerations regarding the first transition are the same as those pointed out in Proposition 5.1. For the second transition to fire, clock x is required to have a value equal to the parameter μ, while clock y is required to be less than n (notice that clock y is also equal to $\mu + \mu = 2\mu$). This means that the following inequality must be satisfied for the second transition to fire: $y = 2\mu < n$, i.e., $\mu < \frac{n}{2}$. □

A more careful observation of the tile presented in Proposition 5.2 reveals that the actual interval forced by this tile is $[0, \frac{n}{2})$. However, the interval obtained in Proposition 5.2 has its left bound open due to the strictly monotonic semantics of the considered time sequences. Indeed, for this tile to force parameter μ to be exactly 0, both transitions should fire immediately, without letting even an infinitesimal amount of time pass. Since this is forbidden by the TA model used in this thesis, considering the interval as open on the left is still considered correct.
It is also interesting to derive clock preconditions for the tiles presented in Proposition 5.1 and Proposition 5.2. Since the first transition from q_1 to q_2 is the same in both cases, clock preconditions coincide. In particular, $\mathcal{P}_{\mathcal{T}}(x) \in \mathbb{R}_{\geqslant 0}$, i.e., clock x is not constrained to have a particular value when entering tile $\mathcal{T}$ (it is reset when the first transition fires), while $\mathcal{P}_{\mathcal{T}}(y) = 0$ (since clock y has to act like a counter).

By Proposition 5.1, Proposition 5.2, and Lemma 5.1 it is now possible to provide an example of a tile able to force the parameter's value inside a specific interval. The tile (denoted as $\mathcal{T}_{12}$) was already presented in Section 3.2 when introducing the concept of PTAs and is reported for convenience in Figure 5.3. $\mathcal{T}_{12}$ is obtained by connecting in sequence tiles presented in Proposition 5.1 and Proposition 5.2: the former (denoted as $\mathcal{T}_1$) can be recognized in locations q_0, q_1, and q_2 of $\mathcal{T}_{12}$, where n is substituted with 4,

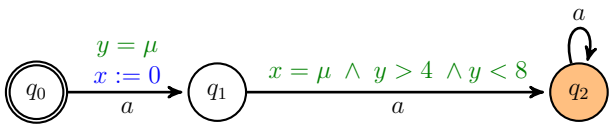

Figure 5.4: A revised version of the tile forcing the interval $(2, 4)$ presented in Figure 5.3.

while the latter (denoted as $\mathcal{T}_2$) can be recognized in locations q_3, q_4, and q_5 of $\mathcal{T}_{12}$, where n is substituted with 8. Since $\mathcal{I}_{\mathcal{T}_1} = (2, \infty)$ and $\mathcal{I}_{\mathcal{T}_2} = (0, 4)$, Lemma 5.1 ensures that $\mu \in (\mathcal{I}_{\mathcal{T}_1} \cap \mathcal{I}_{\mathcal{T}_2}) = ((2, \infty) \cap (0, 4)) = (2, 4) = \mathcal{I}_{\mathcal{T}_{12}}$. Notice that for this condition to hold, it is also necessary to set clocks according to the clock precondition function $\mathcal{P}_{\mathcal{T}_2}$ of $\mathcal{T}_2$, as it is done in the transition from q_2 to q_3.
By this construction, it can be inferred that connecting only elementary tiles, to create tiles able to force arbitrary intervals, results in the creation of tiles having a total number of locations and transitions growing linearly with respect to the number of locations and transitions of the elementary tiles used. It is possible to create by hand smaller tiles forcing arbitrary intervals, though this is a more difficult process.
An example of a tile forcing the same interval $(2, 4)$ of the tile presented in Figure 5.3 is given in Figure 5.4. Here, only three locations are used instead of six, but the guard appearing on the transition from location q_1 to location q_2 is now more complex, since it embodies the interval that was previously determined by the connection of two elementary tiles. It is easy to see that, for the transition from location q_1 to location q_2 to fire, clock y must satisfy the following inequality: $4 < y = 2\mu < 8 \Rightarrow \frac{4}{2} < \mu < \frac{8}{2} \Rightarrow 2 < \mu < 4$, hence it must hold that $\mu \in (2, 4)$.

5.2. The tester tool: an in-depth overview

This section shows how the concepts of tiles and Tiled TAs have been used to enrich TABEC with oracle capabilities when performing random testing, also giving a detailed overview of its implementation. In the following, four classes of tiles are considered, as they are the ones that have been used in TABEC:

- Accepting tiles have one input location and no output locations.
- Binary tiles have one input location and one output location.
- Ternary tiles have one input location and two output locations.
- Randomly generated tiles have one input location and one output location.

It is possible to add new classes of tiles in TABEC by performing minor adjustments in its implementation since it heavily relies on the factory method design pattern.

$\langle TiledTA \rangle \longrightarrow \langle Tile \rangle$
| $\langle TiledTA \rangle$ ($\langle BinOp \rangle$ $\langle TiledTA \rangle$ | $\langle TriOp \rangle$ $\langle TiledTA \rangle$ $\langle TiledTA \rangle$)*
| '(' $\langle TiledTA \rangle$ ')'

$\langle Tile \rangle \longrightarrow \mathcal{T}$ | $\mathcal{T}$ '[' $\langle Integer \rangle$ ']'

$\langle BinOp \rangle \longrightarrow$ '+' | '+1'

$\langle TriOp \rangle \longrightarrow$ '++'

$\langle Integer \rangle \longrightarrow \epsilon$ | (1..9)(0..9)*

Figure 5.5: Tl0's context-free grammar.

5.2.1. Tl0: a language for connecting tiles

Given a set $\Theta = \{\mathcal{T}_0, \mathcal{T}_1, ..., \mathcal{T}_n\}$ of tiles, the tester tool must automatically connect them to build parametric TTAs, which are then checked for emptiness. To connect tiles in a semantically significant manner, several operators have been introduced in TABEC. At the current time of writing, they are:

- $\mathcal{T}_0$ + $\mathcal{T}_1$: each output location of $\mathcal{T}_0$ is connected to an input location of $\mathcal{T}_1$ based on the order these locations are declared inside tiles, only if $|Out_{\mathcal{T}_0}| = |In_{\mathcal{T}_1}|$.
- $\mathcal{T}_0$ +1 $\mathcal{T}_1$: the first output location of $\mathcal{T}_0$ is connected to the first input location of $\mathcal{T}_1$ based on the order these locations are declared inside tiles.
- $\mathcal{T}_0$ ++ $\mathcal{T}_1$ $\mathcal{T}_2$: the first output location of $\mathcal{T}_0$ is connected to the first input location of $\mathcal{T}_1$ while the second output location of $\mathcal{T}_0$ is connected to the first input location of $\mathcal{T}_2$, based on the order these locations are declared inside tiles. In this case, $\mathcal{T}_0$ is required to be a ternary tile.

Operators have been implemented using the factory method design pattern: adding new operators is only a matter of adding new C++ classes. Operator + and +1 are also called binary operators since they require two tiles as operands, while operator ++ is also called a ternary operator since it requires three tiles as operands.
A simple Domain Specific Language (DSL) called Tl0 has been introduced to generate a description of how tiles contained in Θ can be connected using the above operators. A context-free grammar for Tl0 is given in Figure 5.5. Non-terminals are enclosed between angular brackets. The starting non-terminal is $\langle TiledTA \rangle$. Non-terminals $\langle BinOp \rangle$ and $\langle TriOp \rangle$ represent respectively binary and ternary operators. $\langle Integer \rangle$ represents an integer value which can also be omitted, thanks to the production rule $\langle Integer \rangle \longrightarrow \epsilon$. Parentheses allow to nest production rules, specifying precedence levels for the operators.

$\langle TA \rangle \longrightarrow$ '(' $\langle TiledTA \rangle$ ')' '+1' $\langle AccTile \rangle$

$\langle TiledTA \rangle \longrightarrow \langle BinTile \rangle \mid \langle RngTile \rangle$
$\mid (\langle BinTile \rangle \mid \langle RngTile \rangle) \langle BinOp \rangle \langle TiledTA \rangle$
$\mid \langle TriTile \rangle \langle TriOp \rangle$ '(' $\langle TiledTA \rangle$ ')' '(' $\langle TiledTA \rangle$ ')'
$\mid$ '(' $\langle TiledTA \rangle$ ')'

$\langle BinOp \rangle \longrightarrow$ '+' | '+1'

$\langle TriOp \rangle \longrightarrow$ '++'

$\langle AccTile \rangle \longrightarrow \mathcal{T}_{acc}$

$\langle BinTile \rangle \longrightarrow \mathcal{T}_{bin}$

$\langle TriTile \rangle \longrightarrow \mathcal{T}_{tri}$

$\langle RngTile \rangle \longrightarrow \mathcal{T}_{rng} \mid \mathcal{T}_{rng}$ '[' $\langle Integer \rangle$ ']'

$\langle Integer \rangle \longrightarrow \epsilon \mid (1..9)(0..9)^*$

Figure 5.6: Tl0's context-free grammar, strict version.

The terminal symbol $\mathcal{T}$ represents a generic tile contained in Θ, while terminals `+`, `+1`, and `++` represent the operators introduced at the beginning of this subsection. The integer value that can be associated with a tile represents the maximum number of locations a randomly generated tile can have. An example of a syntactically correct string specified in Tl0 using a set of tiles $\Theta' = \{\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3, \mathcal{T}_4\}$ is $\mathcal{T}_0$ `++ (` $\mathcal{T}_1$ `+ (` $\mathcal{T}_0$ `++` $\mathcal{T}_2$ $\mathcal{T}_3$ `) )` $\mathcal{T}_4$, where $\mathcal{T}_0$ is a ternary tile, $\mathcal{T}_4$ is an accepting tile and the others are binary tiles.
Since the tester tool needs to automatically generate Tl0 strings, production rules are randomly chosen and expanded until a string containing only non-terminals is obtained. However, due to the grammar reported in Figure 5.5 being too general, in most of the cases the obtained strings are not syntactically correct, since operators require specific classes of tiles to work with, which are randomly selected. For this reason, the actual implementation is based on a stricter version of the grammar presented in Figure 5.5. This new context-free grammar is shown in Figure 5.6.
Some new non-terminals and production rules have been added in this new version, ensuring that each operator receives the correct class of tiles to work with. Here the starting non-terminal is $\langle TA \rangle$, appearing in a new production rule enclosing within parentheses the declaration of the parametric TTA, forcing its connection with an accepting tile. This is done to ensure the obtained parametric TTA has at least one accepting location, as the tiles used in its construction are not required to have one. Operator `+1` guarantees that a connection is generated regardless of the number of output and input locations between its operands. Non-terminals $\mathcal{T}_{acc}$, $\mathcal{T}_{bin}$, $\mathcal{T}_{tri}$ and $\mathcal{T}_{rng}$ account respectively for accepting,

binary, ternary and randomly-generated tiles.
In the implementation, the `TATileRegExGeneratorStrict` class is in charge of generating random strings according to the strict version of Tl0.

5.2.2. A digression on randomly generated tiles

The ability to randomly generate tiles is essential for random testing to be effective, otherwise, the same pre-built tiles would be continuously combined, leading to uninteresting results. TABEC's architecture can be extended with various methodologies for creating random tiles, since the choice of which one to use is based on the factory method design pattern. At the current time of writing, only one methodology has been implemented inside the `RandomCreatorBarabasiAlbert` class, enabling TABEC to build tiles based on the Barabási-Albert model [23].
In the remainder of this subsection, tiles are considered as graphs $\mathcal{G} = (V, E)$ (also called networks), where V is a set of nodes and E is a set of edges. Barabási-Albert networks exhibit the so-called scale-free property, i.e., few nodes (called hubs) have a very high degree, while a large number of nodes have a much lower degree. The degree of a node $v \in V$ counts how many edges $e \in E$ are connected to v.
A peculiarity of scale-free networks consists of modeling the behavior of real-life networks, e.g., power grid networks, biological networks, transportation networks, and so forth. From this perspective, having tiles able to mimic the structure of these networks may become useful if a TA model of such systems is required.

The pseudocode of the implemented Barabási-Albert algorithm is given in Algorithm 4. The algorithm starts by performing a check on the input parameters in line 2. If everything is correct, an initial network $\mathcal{G}$ is built in line 5 by randomly connecting a given number of nodes as specified by the m_0 parameter. In the `RandomCreatorBarabasiAlbert` class implementation, this initial network is created as a complete graph. Then, in line 7 a while loop used to create and insert new nodes in $\mathcal{G}$ starts its execution until the maximum number of nodes as specified by the n parameter is reached. Once a new node is created, the inner while loop starting in line 10 ensures that it will be connected to a specified number of different nodes, as specified by the m parameter. Notice that at this stage, node v is not yet inserted inside $\mathcal{G}$: this avoids the creation of self-cycles. Line 11 specifies a set keeping track of the nodes that are randomly chosen during this connection procedure. Lines 13 and 14 initialize two probabilities used to determine if the node v created in line 8 can be connected to the node $\bar{v}$ extracted in line 12. The function `deg()` returns the degree of its input: `deg(`$\bar{v}$`)` returns the degree of node $\bar{v}$, while `deg(`$\mathcal{G}$`)` returns the degree of the entire network $\mathcal{G}$. Line 15 specifies the condition for nodes v and

Algorithm 4: Barabási-Albert algorithm implemented in TABEC

Data: (n, m_0, m): the number of nodes the resulting network will have, the initial number of nodes, the number of edges added when inserting a new node.
Result: A Barabási-Albert network $\mathcal{G}$.

```
1  begin
2      if m > m_0 then
3          // It must hold that m ⩽ m_0
4          return error
5      Create an initial network G = (V, E) having |V| = m_0 nodes arbitrarily connected
6      i ← |V|
7      while i < n do
8          Create a new node v
9          createdEdges ← 0
10         while createdEdges < m do
11             Γ = ∅
12             v̄ ← a node randomly chosen from G
13             pLink ← a random number within the interval [0, 1]
14             pNode ← deg(v̄)/deg(G)
15             if pLink ⩽ pNode ∧ v̄ ∉ Γ then
16                 Γ = Γ ∪ v̄
17                 Connect v to v̄
18                 createdEdges ← createdEdges + 1
19         Insert node v in G
20         i ← i + 1
21     return G
```

$\bar{v}$ to be connected. If the condition is satisfied, lines 16, 17, and 18 respectively update the set of chosen nodes, connect v and $\bar{v}$, and increment the number of created edges. In particular, when a new edge e is created in line 17, since in this thesis the graph generated by this algorithm needs to be a parametric nrtTA with two clocks and one parameter, clock guards and resets are randomly generated for e considering two clocks x, y and a parameter keyword "param", in a way such that the nrt condition is always satisfied.

It is trivial to see that the complexity in the worst case is $\mathcal{O}(m_0 \cdot (m_0 - 1) + (n - m_0) \cdot m)$ when considering a complete graph in line 5 and due to the initialization of i in line 6. Being the provided algorithm a randomized algorithm belonging to the class of Las Vegas algorithms, i.e., it is a randomized algorithm ensuring that the correct solution is always found, thanks to the inner loop starting in line 10 being executed until exactly m new edges are inserted, it should be taken into account that the overall complexity may be influenced by the random choices made during the execution of such loop.

5.2.3. From Tl0 to Tiled Timed Automata

Once a Tl0 string $\mathcal{S}$ has been generated by the `TATileRegExGeneratorStrict` class, the tester tool needs to interpret it to understand how tiles must be connected as specified in $\mathcal{S}$. For this purpose, a lexer (implemented in the `TATileInputLexer` class) and a parser (implemented in the `TATileInputParser` class) for the Tl0 language have been developed. Recalling the experimental academic context in which TABEC was conceived to be used and given the simple nature of the Tl0's grammar, the lexer and the parser have been developed by hand, to ensure the highest level of customization.

When a string $\mathcal{S}$ is received, the lexer scans it and replaces each symbol found in $\mathcal{S}$ with a suitable token that can subsequently be interpreted by the parser. Concerning tiles, the token with which their symbol is associated is simply the name of the file in which they are saved. Indeed, for tiles to be used, they must be saved in an `.xml` file (this can be easily done using the converter tool, as already explained in Section 4.2), paying attention to save them in the right TABEC's directory based on the class in which they belong, i.e., accepting, binary or ternary tiles. Tokens associated with operators and parentheses are instead specified inside the lexer itself.
Once the lexer has tokenized the input string $\mathcal{S}$, the parser scans it from left to right and performs actions based on the current token it reads. Actions have been implemented by using the factory method design pattern: this makes it easy to add new actions simply by creating new classes associated with new actions. This also enhances maintainability, since to change an already existing action only its class needs to be modified.
Actions affect an internal data structure used by TABEC to manipulate and keep track of tiles used during the generation process. This data structure has been implemented as a doubly linked list and an example of one of its nodes is reported in Figure 5.7. The elements contained in this node are:

- `next`: a pointer to the next node of the list (by default it points to `null`).
- `prev`: a pointer to the previous node of the list (by default it points to `null`).
- `nesting level`: an integer variable tracking the current nesting level due to the use of parentheses. A higher value implies a higher precedence level for a node.
- `tile stack`: a stack containing a JSON representation of the tiles used in the node. The choice of JSON is due to the ease with which this format can be manipulated, thanks to the APIs provided by the `nlohmann` JSON library (e.g., it is immediate to add new locations or to add new transitions inside an already existing tile).
- `operator stack`: a stack containing the operators used in the node (as strings).

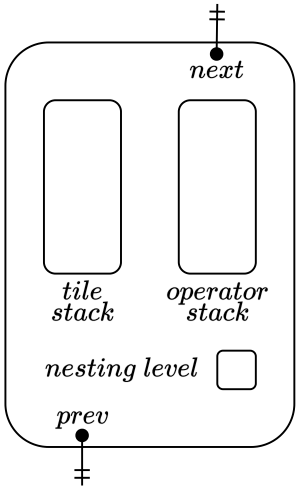

Figure 5.7: Graphical representation of a node contained in the doubly linked list data structure used by TABEC during the parsing process.

The possible actions that can be performed based on the tokens encountered while parsing the tokenized input string $\mathcal{S}$ involve: pushing a tile to the top of the tile stack when a tile token is encountered, pushing an operator to the top of the operator stack when an operator token is encountered, creating a new node incrementing the nesting level when a left parenthesis token is encountered or consuming and deallocating the current node when a right parenthesis token is encountered. The functioning of this mechanism is better explained using an illustrative example.

Let's focus on Figure 5.8, where the input string $\mathcal{S} = \mathcal{T}_0$ ++ ($\mathcal{T}_1$ + ($\mathcal{T}_2$ + $\mathcal{T}_3$)) $\mathcal{T}_4$ is given, considering a set of tiles $\Theta = \{\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3, \mathcal{T}_4\}$. The example is described considering only symbols as appearing in $\mathcal{S}$ but it is worth recalling that, in the actual implementation, before the parsing process can start, symbols have first to be transformed into tokens by the lexer. In the beginning, the list presented in this example is empty and composed of only one node having a nesting level value equal to 0. The node with the highest nesting level value is also called the head of the list and denoted in Figure 5.8 with a $\mathcal{H}$ symbol.
In step 1, the $\mathcal{T}_0$ symbol is encountered: this makes the parser execute an action that pushes the tile corresponding to $\mathcal{T}_0$ on top of the tile stack in the head node. Next, in step 2 the symbol ++ is read: since this is an operator, its corresponding string representation is pushed on top of the operator stack in the head node. Then, in step 3 a left parenthesis symbol (is read. This means that a new precedence level is specified and hence a new node must be inserted inside the list. The new node has a nesting level value equal to 1 since it is one level above the previous node (which has a nesting level value equal to 0) and, for this reason, it becomes the new head of the list. Steps 4 and 5 read respectively

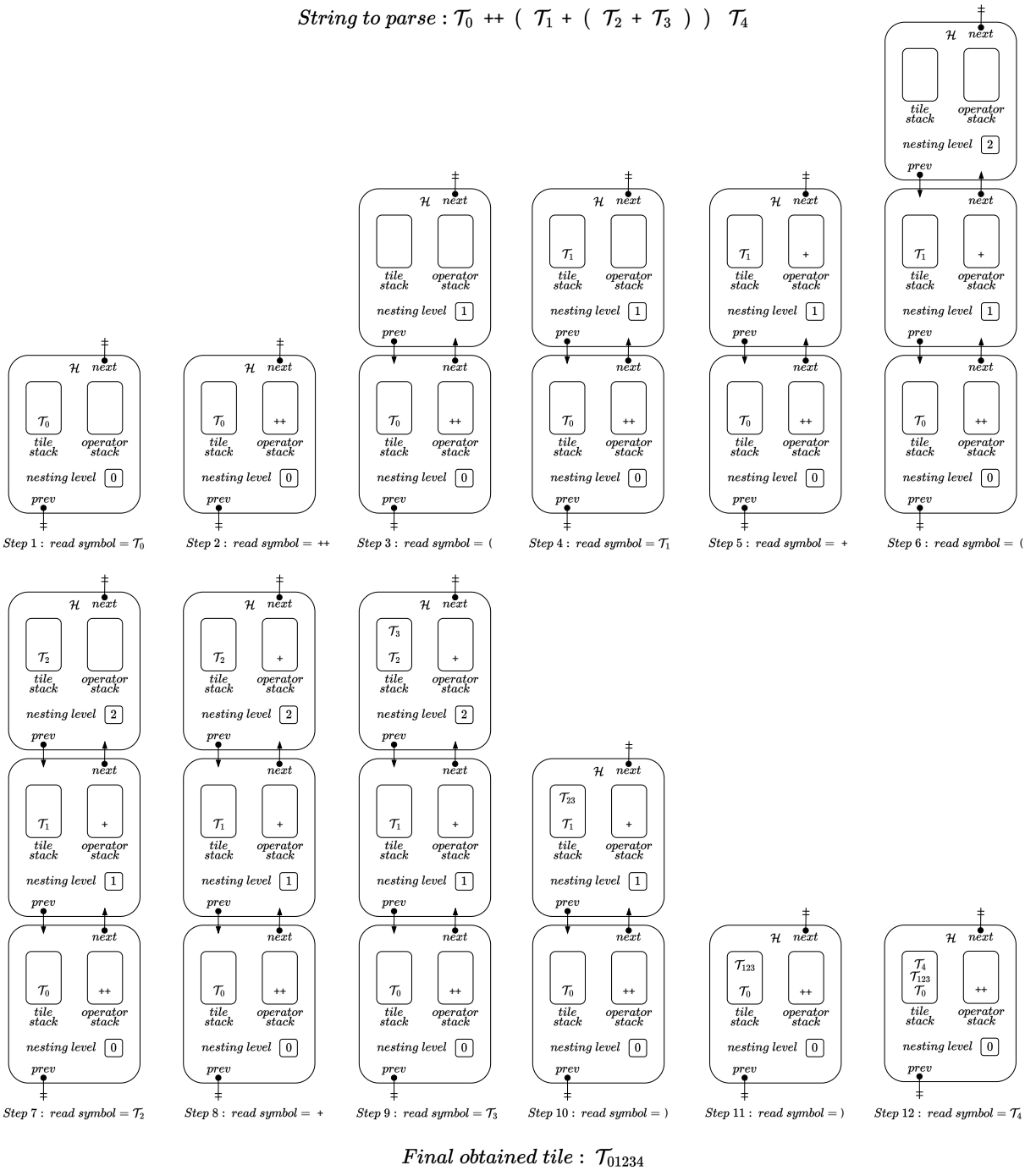

Figure 5.8: Example of how the internal data structure of TABEC is manipulated when parsing a given Tl0 string. A $\mathcal{H}$ symbol inside a node denotes it as the current head of the list.

the tile symbol $\mathcal{T}_1$ and the operator symbol +, pushing the relative tile and operator string on top of the tile stack and operator stack of the head node. In step 6 a new precedence level is specified due to another (symbol: the list grows with a new node having a

nesting level value equal to 2, which also becomes the new head of the list. Steps from 7 to 9 read the substring $\mathcal{T}_2$ + $\mathcal{T}_3$, pushing the relative tiles and operator string on top of the respective stacks in the head node. Then, in step 10 a right parenthesis symbol) is read. This causes the current head node to be consumed: its tiles are connected using the operators available in its operator stack until this stack becomes empty. To do this, the number of operands required by the operator on top of the operator stack is first acknowledged. Then, the needed tiles are taken from the tile stack extracting them from top to bottom, subsequently combining them according to the considered operator. Notice that the last extracted tile is used on the left-hand side of the operator, while the other tiles are used on the right-hand side of the operator. The resulting intermediary tile obtained in the process is pushed again on top of the tile stack of the head node so that it can be reused and connected with other tiles. Once all tiles are connected by using all the available operators, if the head's nesting level value is greater than 0, the resulting tile is pushed on top of the tile stack of the previous node (the node reachable following the head's prev pointer), the previous node becomes the new head of the list, and the old head node is deallocated. Step 11 is similar to step 10: the current head node is consumed and the resulting tile is pushed on top of the tile stack contained in the previous node, the previous node becomes the new head of the list, and the old head node is deallocated. Finally, in step 12 the tile symbol $\mathcal{T}_4$ is read and the relative tile is pushed on top of the tile stack in the head node. If no further symbols are read, one last step involves the connection of the tiles still contained in the head node. The performed actions are the same as the ones executed when a) symbol is encountered.
The final result in the example is the tile $\mathcal{T}_{01234}$, obtained by properly connecting all tiles contained in Θ and used in $\mathcal{S}$ (with the specified operators).

As a final note, it is interesting to point out that, due to this stack-based implementation, operators are right-associative. For instance, the following sequence of tiles connected by using the + operator: $\mathcal{T}_0$ + $\mathcal{T}_1$ + ... + $\mathcal{T}_{n-2}$ + $\mathcal{T}_{n-1}$ + $\mathcal{T}_n$ in which explicit precedence levels are omitted, is equivalent to: $\mathcal{T}_0$ + ($\mathcal{T}_1$ + ... + ($\mathcal{T}_{n-2}$ + ($\mathcal{T}_{n-1}$ + $\mathcal{T}_n$))) in which precedence levels are specified by explicitly inserting parentheses.

Since the random Tl0 strings generated by the `TATileRegExGeneratorStrict` class are syntactically correct and thanks to the strict version of Tl0 ensuring to obtain parametric TTAs having at least one accepting location, the process described in this subsection always generates meaningful parametric TTAs. These are provided as a JSON intermediate representation which can be further manipulated or converted into other file formats, like `.tck` and `.dot`, to be used by other tools.

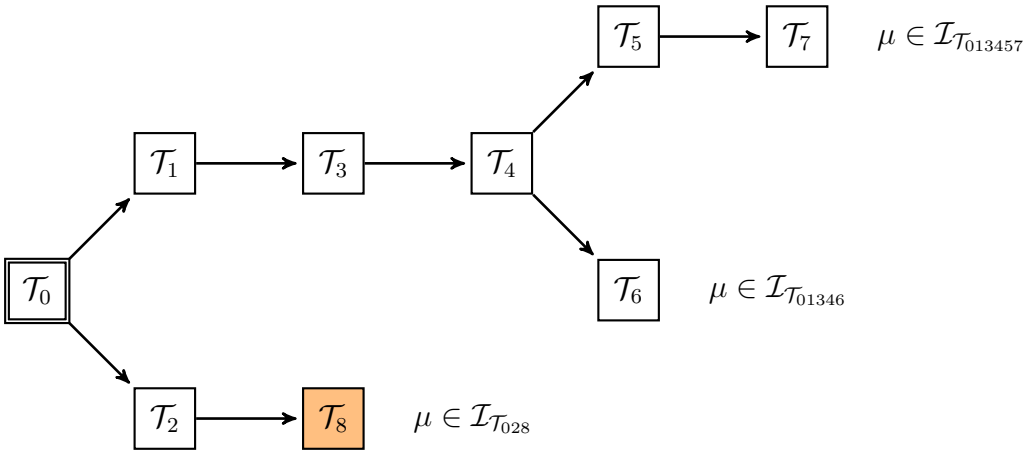

Figure 5.9: Example tree structure of a parametric TTA used in the tester. Graphical details are omitted and only transitions and tiles' names are shown.

5.2.4. Enriching TABEC with oracle capabilities

It is now clear how the tester tool creates random parametric TTAs by combining different tiles, i.e., by generating random strings given in Tl0 (using its strict version) and subsequently parsing these strings while acting on an internal doubly linked list data structure. What is still missing is how TABEC can predict the outcomes deriving from random tests, i.e., how it can predict the interval in which the parameter's value may fall before tests are executed, hence becoming an oracle.
Given a parametric TTA $\mathcal{A} = (\Theta, \mathcal{T}_0, \Xi, \mathcal{B}, \mathcal{X}, \Pi, \Upsilon)$ and a tile $\mathcal{T} \in \Theta$, the term *child* is used to denote a tile $\bar{\mathcal{T}} \in \Theta$ such that $(\mathcal{T}, \bar{\mathcal{T}}, ...) \in \Upsilon$, while the term *leaf* is used to denote a tile $\tilde{\mathcal{T}} \in \Theta$ such that there are no transitions of the form $(\tilde{\mathcal{T}}, \mathcal{T}, ...)$ in Υ.
In the remainder of this subsection, parametric TTAs are assumed to have a binary tree structure with tiles as nodes, where no nodes have any children in common.
The considerations that are going to be illustrated can be generalized in the case of parametric TTAs having a tree structure where each node can have an arbitrary number of children, always with the restriction of avoiding nodes having any children in common.

An example of a parametric TTA $\mathcal{A}$ structured as a binary tree is given in Figure 5.9, obtained from the Tl0 string: $\mathcal{T}_0$ ++ ($\mathcal{T}_1$ + $\mathcal{T}_3$ + $\mathcal{T}_4$ ++ ($\mathcal{T}_5$ + $\mathcal{T}_7$) ($\mathcal{T}_6$)) ($\mathcal{T}_2$ + $\mathcal{T}_8$).
The root of $\mathcal{A}$ corresponds to tile $\mathcal{T}_0$, which also is an initial tile. In this figure, on the right of each leaf node ($\mathcal{T}_6$, $\mathcal{T}_7$ and $\mathcal{T}_8$) the strictest interval forced by the path leading from the root to a leaf is reported. These intervals contain inside their subscript the numbers corresponding to the tiles traversed to reach a specific leaf.
Since there are no children in common in the tree structure of $\mathcal{A}$, each path leading from the root to a leaf is unique (this also holds considering trees with nodes having any number

of children, if children are not shared by any node). Each path from the root to a leaf can be considered as a set of tiles connected in sequence. Hence, by Lemma 5.1, to predict the interval in which the parameter's value can fall, it is sufficient to determine the interval obtained when reaching a leaf, for each leaf contained in $\mathcal{A}$.
Considering Figure 5.9, three intervals are available as possible candidates in which the parameter's value can fall: $\mathcal{I}_{\mathcal{T}_{013457}}$, $\mathcal{I}_{\mathcal{T}_{01346}}$ and $\mathcal{I}_{\mathcal{T}_{028}}$. Since $\mathcal{T}_8$ is an accepting tile, in order for $\mathcal{L}(\mathcal{A})$ to be non-empty, a run $\widetilde{\rho}$ must enter in tile $\mathcal{T}_8$. For this reason, it is easy to see that the only interval obtained by satisfying this condition is $\mathcal{I}_{\mathcal{T}_{028}}$ and, if the language admits a Büchi acceptance condition, the parameter's value is contained inside that interval.
A more careful observation reveals that a parametric TTA needs not to traverse all tiles up to a leaf to admit a Büchi acceptance condition, e.g., if in Figure 5.9 tile $\mathcal{T}_3$ is assumed to be an additional accepting tile, $\mathcal{A}$ could stay in that tile forever, without reaching any leaf. Indeed, the computation of the interval on the path leading to a leaf only returns the strictest possible interval but, being this interval computed before tests are executed, it is not known whether a parametric TTA has to reach a leaf to exhibit a Büchi acceptance condition. For this reason, when computing the resulting interval on a path from the root to a leaf, intermediate intervals must be kept in consideration as well.

Intervals can be specified when creating a tile in Uppaal by inserting a string corresponding to the following regular expression: *bound:l:r*(| *bound:l:r*)* inside the Declaration section of Uppaal's System Editor, where l and r have to be substituted respectively with a value $l \in \mathbb{R}_{\geqslant 0}$, $r \in (\mathbb{R}_{\geqslant 0} \cup \{\infty\})$ (infinity is represented by the keyword *inf*) or with the *nan* keyword, the latter meaning an interval is not required (e.g., for accepting tiles with only one location). Since the interval of randomly generated tiles cannot be known a priori, this is set to the loosest admissible interval, i.e., $(0, \infty)$. To have meaningful intervals, it is required that $l \leqslant r$.
In the implementation, the `TABoundsCalculator` class monitors the possible parameter's intervals allowed in a given parametric TTA. The class only considers proper intervals, i.e., empty intervals or *nan* intervals are discarded. The class starts by keeping track of every intermediate interval found starting from the root. When a node having more than one child is found, the intervals found up to that node are duplicated, to be used in the new paths that originate from such node. When a leaf node is reached through a path, all the intervals corresponding to that specific path are stored inside the `TABoundsCalculator` class. The check on the condition determining whether a leaf node is reached is carried out when consuming a node of the doubly linked list due to a `)` symbol. Once the parsing of the given Tl0 string is concluded, if the language of the relative parametric TTA is not

empty, the parameter's value is checked against the intervals found during computation, returning the strictest interval in which the parameter's value falls. Results are then saved to a `.txt` log file.

5.3. Running the tester

This section provides a final summary of the content presented in this chapter. In particular, a description of how the TABEC's executables, C++ classes, and Bash scripts interact when running the tester is given, aided by the sequence diagram reported in Figure 5.10.

First, to perform random testing with TABEC, the tester tool executable needs to be run. This is done with the following shell command:

```
$ ./tester -tst t -sup s -nbt n -lns /path/to/tck-liveness/dir -atc -bds -all
```

The meaning of the options used in this command is the following:

- `-tst t` specifies to generate a string in the strict version of Tl0 by performing a total number of productions equal to `t`.
- `-sup s` specifies the max number `s` of locations a randomly-generated tile will have.
- `-nbt n` specifies the number `n` of tests randomly generated and executed by the tester. If the given value of `n` is equal to 0, an interactive prompt allows the manual insertion of a Tl0 string.
- `-lns /path/to/tck-liveness/dir` path to the directory containing the tck-liveness executable (in tChecker this corresponds to the bin folder). If omitted, the path specified during the build of TABEC is used.
- `-atc` the checker automatically runs after the random test generation has finished.
- `-bds` used to print all the parameter intervals found when executing tests.
- `-all` the computation is not terminated when the first parameter's value leading to a Büchi acceptance condition is found, but it continues to check all the other possible parameter's values, as specified in Algorithm 1.

The tester's execution starts by entering a first loop. Given an integer variable named $totalTests$ representing the total number of random tests that must be generated and a counter n initialized to 0, the loop is executed as long as the condition $n < totalTests$ holds. In every iteration, a new parametric TTA is generated. In particular, the tester starts by resetting the data structure used to compute the parameter's intervals of the

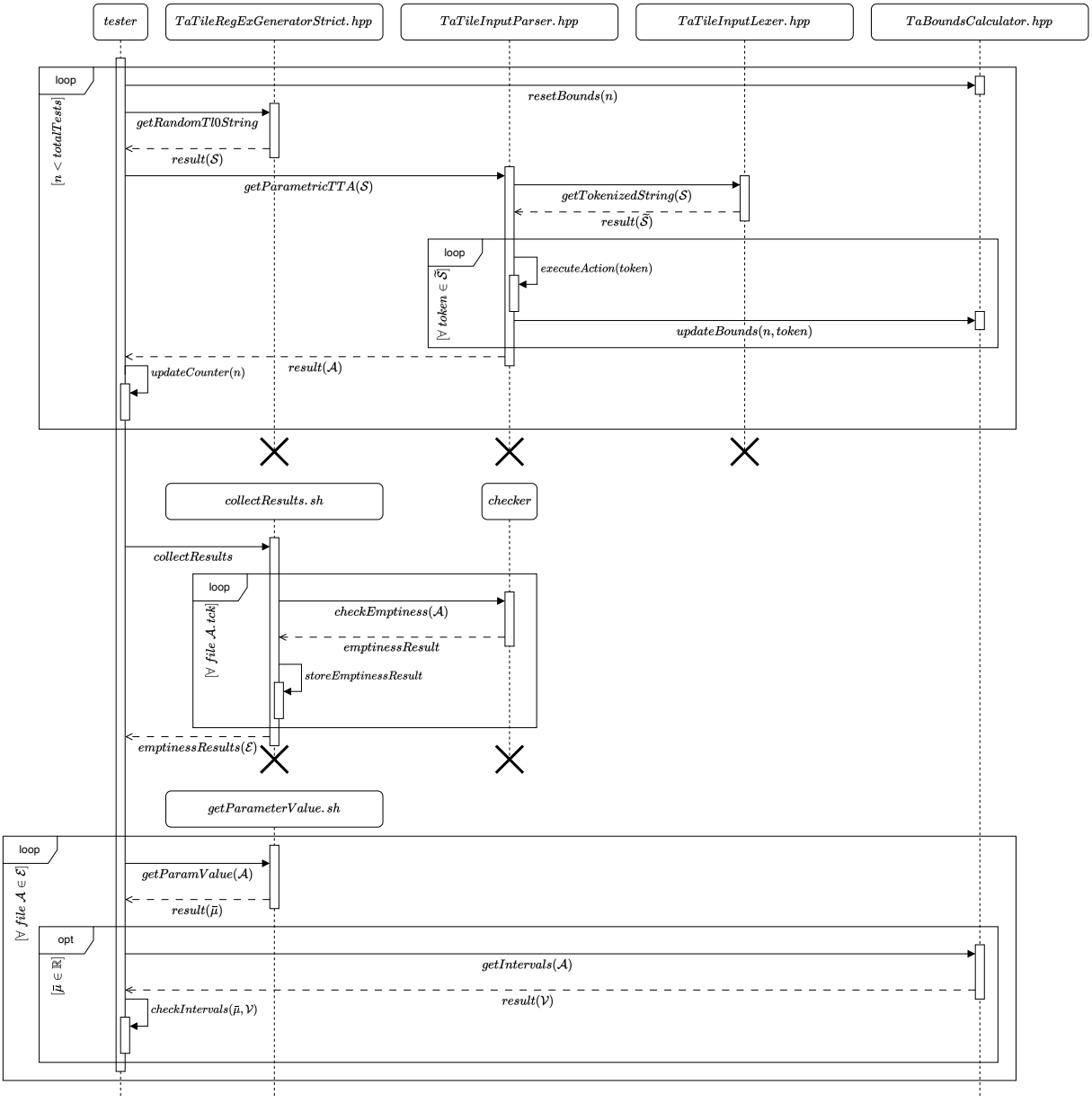

Figure 5.10: Sequence diagram showing the order of invocation of TABEC's executables (tester and checker), C++ classes (saved with a `.hpp` extension) and Bash scripts (saved with a `.sh` extension) when using the tester tool to perform random testing. Checker's execution was already provided in the sequence diagram depicted in Figure 4.4.

current parametric TTA, which is represented by the iteration number n. This data structure is contained in the `TaBoundsCalculator.hpp` class. Then, the tester asks the `TaTileRegExGeneratorStrict.hpp` class to generate a string containing a parametric TTA description using the strict version of Tl0. The resulting Tl0 string $\mathcal{S}$ is then given to the `TaTileInputParser.hpp` class to get a valid JSON intermediate representation of the described parametric TTA. To accomplish this task, the parser first invokes the `TaTileInputLexer.hpp` class to obtain a tokenized version of $\mathcal{S}$ (denoted by $\widetilde{\mathcal{S}}$). Then, for each token inside $\widetilde{\mathcal{S}}$, the parser executes the action corresponding to that token and, based on the token's class (i.e., a tile token, an operator token, or a parenthesis token), the parameter's intervals for the parametric TTA corresponding to iteration number n are updated. Once the parsing of $\widetilde{\mathcal{S}}$ has finished, the JSON intermediate representation of the obtained parametric TTA $\mathcal{A}$ is given back to the tester, which can now increment by 1 the counter n, starting a new iteration of the loop.
Once the JSON intermediate representation of $\mathcal{A}$ is received, the tester converts and saves it both in a `.dot` file (by relying on the `TADotConverter.hpp` class) and in a `.tck` file (by relying on the `TAutotTranslator.hpp` class). These conversion steps are omitted in the sequence diagram reported in Figure 5.10 to enhance clarity and conciseness.
After obtaining a valid `.tck` representation of the randomly generated parametric TTAs, their emptiness is checked by exploiting the `collectResults.sh` script. This script invokes the checker tool for each $\mathcal{A}$`.tck` file, asking it to check $\mathcal{A}$`.tck`'s emptiness. Checker's execution was already presented in the sequence diagram of Figure 4.4. Recall that the actual emptiness checking is performed by the external tck-liveness tool of tChecker. The `collectResults.sh` script stores the obtained results, which are subsequently saved inside `.txt` log files.
When all $\mathcal{A}$`.tck` files have been processed, control goes back to the tester: it now enters a second loop, in which it uses the previously obtained emptiness results (for convenience denoted as $\mathcal{E}$) to verify if the parameter's value for a given parametric TTA $\mathcal{A}$ falls inside the respective precomputed intervals, given that $\mathcal{L}(\mathcal{A})$ is not empty. For each `.txt` log file, the tester asks the `getParameterValue.sh` script to retrieve the parameter's value $\bar{\mu}$ (obtained after checking emptiness over $\mathcal{A}$) contained in that file. If $\mathcal{L}(\mathcal{A})$ admits a Büchi acceptance condition (i.e., $\bar{\mu} \in \mathbb{R}$), the precomputed intervals $\mathcal{V}$ for $\mathcal{A}$ are retrieved from the `TaBoundsCalculator.hpp` class. Finally, the tester checks, for each interval $v \in \mathcal{V}$, if the parameter's value $\bar{\mu}$ falls inside one of these intervals (i.e., if $\bar{\mu} \in v$, for some $v \in \mathcal{V}$). A new iteration of this loop is executed until all `.txt` log files contained in $\mathcal{E}$ have been analyzed.

6 | Experimental Results

This chapter presents experimental results for validating both the correctness of the algorithm used to check parametric nrtTAs emptiness and TABEC's oracle forecasting. Experiments have been carried out on two different machines: an Apple MacbookPro laptop with an M1 Pro CPU and a Linux server with an AMD EPYC 7282 CPU. The CPU specifications of these machines are reported in Table 6.1.

Characteristic	Apple M1 Pro	AMD EPYC 7282
Number of cores	10	16
Number of threads	10	32
Base clock	2.064 GHz	2.800 GHz
Maximum boost clock	3.220 GHz	3.200 GHz
L3 cache	24 MB	64 MB
Architecture	ARM	x86

Table 6.1: CPU specifications of the machines used to execute experiments.

In the remainder of this chapter, the size of a given TA is measured as the sum of the number of its locations and its transitions. Tiles reported in Table 6.2 are used to build parametric nrtTAs, i.e., every tile is a parametric tile built in a way to satisfy the nrt property. In this table, the first column contains the tiles' type (either accepting, binary, ternary, or random), the second column contains the tiles' name, i.e., the name of the `.xml` file where tiles are saved (recall that these also correspond to the tokens used by the parser), the third column contains the symbols used to specify tiles inside a Tl0 string, and the fourth column contains the interval forced by the given tiles. Also, recall that accepting tiles do not force any interval by construction and that randomly generated tiles are assumed to force the loosest admissible interval.

It is interesting to notice that the `bin_acc_2_8` tile is a particular case of binary tile having both an accepting location and an output location. Due to the classification of tiles presented at the beginning of Section 5.2, this tile correctly identifies as a binary tile.

Tile Type	Tile name	Tile symbol	Tile interval
Accepting	acc	t1	$\varnothing$
Binary	bin_0_5	t2	$\mathcal{I}_{\texttt{t2}} = (0, 5)$
	bin_1_1	t3	$\mathcal{I}_{\texttt{t3}} = [1, 1]$
	bin_2_8	t4	$\mathcal{I}_{\texttt{t4}} = (2, 8)$
	bin_3_inf	t5	$\mathcal{I}_{\texttt{t5}} = (3, \infty)$
	bin_acc_2_8	t6	$\mathcal{I}_{\texttt{t6}} = (2, 8)$
Ternary	tri_3_inf_2_2	t7	$\mathcal{I}_{\texttt{t7}} = ((3, \infty) \cup [2, 2])$
	tri_4_6_0_2	t8	$\mathcal{I}_{\texttt{t8}} = ((4, 6) \cup (0, 2))$
Random	t_barabasi_albert	t:BA[n]	$\mathcal{I}_{\texttt{t:BA}} = (0, \infty)$

Table 6.2: Tiles used in experiments.

6.1. Two guided examples

This section provides two ad-hoc-built examples to prove the correctness of TABEC's implementation and to provide a description of TABEC's outputs. The first example shows that, by connecting a set of tiles in a way that the resulting parametric nrtTA exhibits a Büchi acceptance condition and the resulting interval is not empty, the parameter's value falls inside that interval. The second example shows that by connecting a set of tiles in a way that the resulting interval is empty, the resulting parametric nrtTA's language is empty. Both these examples have been executed on the M1 Pro CPU and with the tester's `-all` option enabled.

6.1.1. Positive acceptance example

The following example is based on a parametric nrtTA $\mathcal{A}$ built according to the following Tl0 string: `t7 ++ ( t2 + t1 ) ( t5 + t1 )`. A graphical depiction of $\mathcal{A}$ (generated by the grapher tool) is reported in Figure 6.1. As can be inferred from Table 6.2, the branch on the top (starting due to the transition from location id1T0 to location id2T0) composed by tiles `t7`, `t2`, and `t1`, forces the interval $\mathcal{I}_{\texttt{t7t2t1}} = (3, 5)$ due to `t7` forcing $\mathcal{I}_{\texttt{t7}} = (3, \infty)$ on this branch. The branch on the bottom (starting due to the transition from location id1T0 to location id3T0) composed by tiles `t7`, `t5`, and `t1`, generates an empty interval due to `t7` forcing $\mathcal{I}_{\texttt{t7}} = [2, 2]$ on this branch. Hence, in order for $\mathcal{A}$ to exhibit a Büchi acceptance condition, the parameter's value $\bar{\mu}$ should be such that $\bar{\mu} \in \mathcal{I}_{\texttt{t7t2t1}} = (3, 5)$. Furthermore, by the construction of the used tiles, the language of $\mathcal{A}$ should be non-empty. These claims are proven by looking at `.txt` log files output by TABEC. The first

one is the `Results.txt` log, which contains the following information:

```
1 RegExTA_1
2 Language is not empty with parameter value: 4.50000
3 Tiles used: bin_0_5 bin_3_inf tri_3_inf_2_2
```

Line 2 tells that the language of $\mathcal{A}$ is not empty, also showing the value found for the parameter. Line 3 reports the tiles used to build $\mathcal{A}$ (excluding accepting tiles and randomly generated tiles).
Next, forecasting on the parameter's value $\bar{\mu}$ being in the interval $\mathcal{I}_{\texttt{t7t2t1}} = (3, 5)$ is proven to be correct. To do this, it is sufficient to analyze the content of the `ParameterBounds.txt` log file:

```
1  RegExTA_1
2  ---------
3  The following bounds have been found:
4  Bound: { 3.0 - 5.0 }
5  Bound: { 2.0 - 2.0 }
6  Bound: { 3.0 - 5.0 }
7  Bound: { 2.0 - 2.0 }
8  Bound: { 3.0 - inf }
9  Disjoint bound :: Bound: { 3.0 - 2.0 }
10 Bound: { 3.0 - inf }
11 Disjoint bound :: Bound: { 3.0 - 2.0 }
12 The parameter value is: 4.5
13 Parameter value: 4.5 is within found bound: { 3.0 - 5.0 }
14 true
```

In line 13 the interval corresponding to $\mathcal{I}_{\texttt{t7t2t1}} = (3, 5)$ is correctly determined. It is also interesting to notice in lines 9 and 11 a detected empty interval, due to the aforementioned reasoning about the bottom branch of $\mathcal{A}$. Recall that forecasting is done by TABEC before

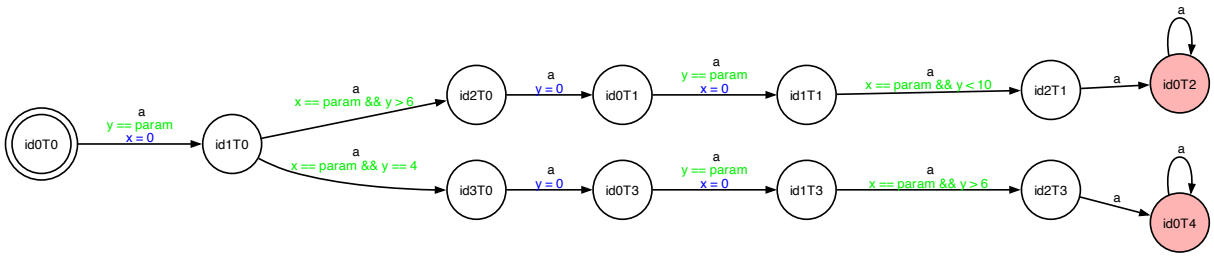

Figure 6.1: Parametric nrtTA used in the first guided example.

actually checking the given TA, hence without knowing the parameter's value a priori. This confirms the claims stated above: the language of $\mathcal{A}$ is not empty and the interval in which the parameter's value falls is correctly determined.
Lastly, it is interesting to inspect resource utilization throughout the entire execution. Since, as stated in Section 4.1, the complexity of the algorithm is driven by the check carried out by tChecker exploiting zone graphs, resource utilization focuses on tChecker and not on TABEC. This information is contained inside the `ResourceUsage.txt` log:

```
1  RegExTA_1
2  ---------
3  Total number of locations:                          12
4  Total number of transitions:                        13
5  Total number of runs:                               42
6  Mean algorithm running time [milliseconds]:         0.035122
7  Peak algorithm running time [milliseconds]:         0.075917
8  Mean tChecker running time [milliseconds]:          6.2619
9  Peak tChecker running time [milliseconds]:          10
10 Mean maximum memory utilization [Bytes]:            4.78803e+06
11 Peak maximum memory utilization [Bytes]:            5.17734e+06
12 Mean number of stored zone graph states:            4.69048
13 Mean number of visited zone graph states:           4.69048
14 Mean number of visited zone graph transitions:      3.7619
```

Lines 3 and 4 give information about the structure of the analyzed TA. Line 5 reports the number of times tChecker is invoked during the execution of the algorithms presented in Section 4.1. Lines 6 and 7 provide timing results focusing on the algorithm used by tChecker to check emptiness for a given TA. Lines 8 and 9 provide timing results focusing on the whole tChecker's execution. Lines 10 and 11 provide results concerning memory occupied by tChecker during its execution. Finally, lines 12 to 14 show some details about the zone graph used when performing emptiness checking.

6.1.2. Negative acceptance example

The following example is based on a parametric nrtTA $\mathcal{A}$ built according to the following Tl0 string: `t3 + t5 + t1`. A graphical depiction of $\mathcal{A}$ (generated by the grapher tool) is reported in Figure 6.2. As can be inferred from Table 6.2, the interval obtained by intersecting all the intervals forced by tiles used to build $\mathcal{A}$ is empty. For this reason, the language of $\mathcal{A}$ should be empty and no interval in which the parameter's value may

fall should be computed. These claims are proven by the log files already mentioned in the previous subsection (here reported without explanations). The content of the `Results.txt` log is as follows:

```
1 RegExTA_1.txt
2 Language is empty
3 Tiles used: bin_1_1 bin_3_inf
```

Line 2 confirms that the language is indeed empty. Concerning the `ParameterBounds.txt` log, since the language of $\mathcal{A}$ turned out to be empty, its content is simply:

```
1 RegExTA_1
2 ---------
3 The language of the given TA is empty and thus
  no parameter has been found.
```

For the sake of completeness, the content of the `ResourceUsage.txt` log is reported:

```
1  RegExTA_1
2  ---------
3  Total number of locations:                    7
4  Total number of transitions:                  7
5  Total number of runs:                         50
6  Mean algorithm running time [milliseconds]:   0.0308391
7  Peak algorithm running time [milliseconds]:   0.1235
8  Mean tChecker running time [milliseconds]:    5.54
9  Peak tChecker running time [milliseconds]:    15
10 Mean maximum memory utilization [Bytes]:      4.75005e+06
11 Peak maximum memory utilization [Bytes]:      5.0135e+06
12 Mean number of stored zone graph states:      2.04
13 Mean number of visited zone graph states:     2.04
14 Mean number of visited zone graph transitions: 1.04
```

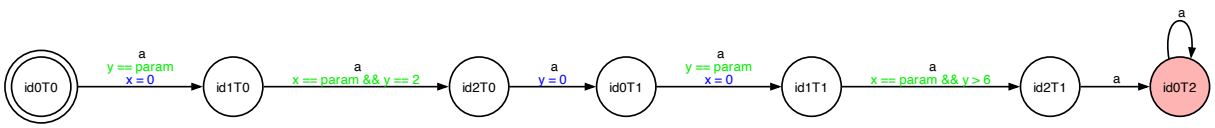

Figure 6.2: Parametric nrtTA used in the second guided example.

6.2. Scalability testing

This section provides scalability testing results for both the tester tool and tChecker. Tests have been executed on the AMD EPYC 7282 CPU.

6.2.1. Scalability of the tester

To measure the scalability of the tester tool, progressively bigger parametric nrtTAs were generated. To do this, Tl0 strings were iteratively constructed by performing one additional production rule for each new iteration (i.e., incrementing the parameter of the `-tst` command-line option by one at each iteration). Randomly generated tiles were set to have six locations each. A first test suite was conducted by setting a timeout equal to 62 minutes for the tester to complete its execution. Next, a second test suite was conducted by setting a timeout equal to 122 minutes for the tester to complete its execution. Since in Section 4.1 two different algorithms are provided, both versions were tested (recall that in the tester implementation, it is possible to switch between them by using the `-all` command-line option). The obtained results are summarized in Table 6.3.

Metric	Algorithm 1 62 minutes	Algorithm 1 122 minutes	Algorithm 3 62 minutes	Algorithm 3 122 minutes	Total
Tests executed	79	103	81	104	367
Biggest TA size	869	1172	911	1199	-
Total non empty TAs	28	38	25	38	129
Total empty TAs	51	65	56	66	238
Prediction accuracy	100%	100%	100%	100%	100%

Table 6.3: Scalability results of the tester tool considering Algorithm 1 and Algorithm 3 when setting timeouts equal to 62 minutes and 122 minutes.

Observing the above table, Algorithm 3 turns out to be faster than Algorithm 1, since it allows bigger parametric nrtTAs checking without violating the specified timeouts. Since the tested parametric nrtTAs are randomly generated, results may exhibit slight variations across multiple trials. In addition, oracle forecasting about the parameter's value carried

out by TABEC had an accuracy of 100% (the accuracy is computed as the ratio between the total number of non-empty TAs and the total number of correct predictions).

6.2.2. Scalability of tChecker

To measure the scalability of tChecker, progressively bigger TAs were generated. The maximum constant appearing in the generated TAs was set to 10. The random TA generation was carried out by starting with a TA having 7 locations and 12 transitions (hence a size equal to 19), iteratively incrementing its size at steps of 100 locations and 200 transitions each. In every iteration, a timeout equal to 62 minutes was set. The maximum obtained TA size satisfying the timeout was equal to 4819, composed of 1607 locations and 3212 transitions. In addition, three additional tests were executed augmenting the number of locations up to 1698 and the number of transitions up to 3394, hence obtaining a TA of size 5092 still satisfying the timeout of 62 minutes. In this latter case, the total tChecker execution time was equal to 3714 seconds = 61.9 minutes.

6.3. Resource utilization testing

In this section, a total of 18 tests of increasing complexity have been generated to measure tChecker resource utilization. The random TA generation was carried out by starting with a TA having 7 locations and 12 transitions (hence a size equal to 19), iteratively incrementing its size at steps of 40 locations and 80 transitions each. The maximum constant appearing in the generated TAs was set to 10. Tests have been executed on the AMD EPYC 7282 CPU by running the tester without the command-line option `-all`. Average measures were computed considering the number of iterations performed by the algorithms presented in Section 4.1. Results were taken from the `ResourceUsage.txt` log files, as mentioned in Section 6.1.
Plots reporting the obtained results are depicted in the remainder of this section. Tables containing the actual measured values corresponding to the ones depicted in the aforementioned plots are contained in Appendix A.
Figure 6.3 shows that the overall tChecker execution time does not increase linearly with the size of the given TAs. Memory consumption and the SCC-based algorithm time, depicted respectively in Figure 6.4 and Figure 6.5, are dependent on the obtained zone graph, in particular on the number of its visited states and its visited transitions, which are reported in Figure 6.6.

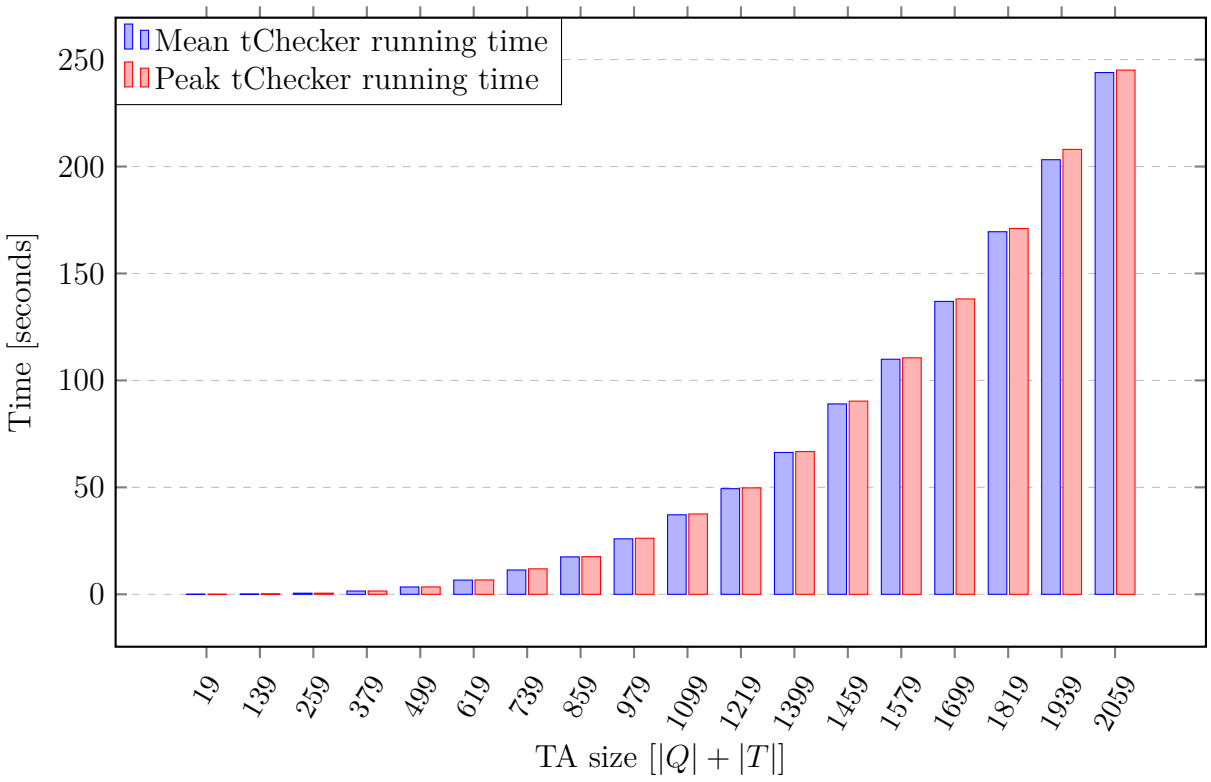

Figure 6.3: Mean and peak running time results for tChecker (in seconds).

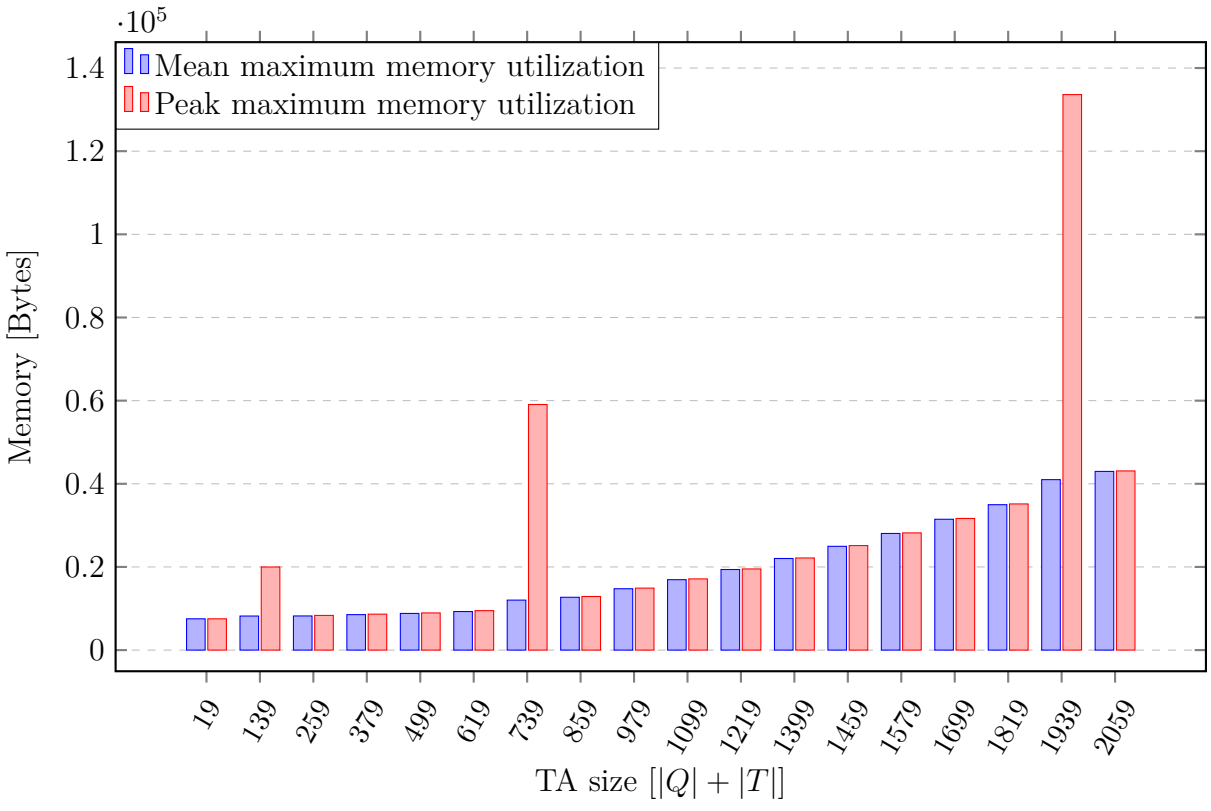

Figure 6.4: Mean and peak occupied memory results for tChecker (in Bytes).

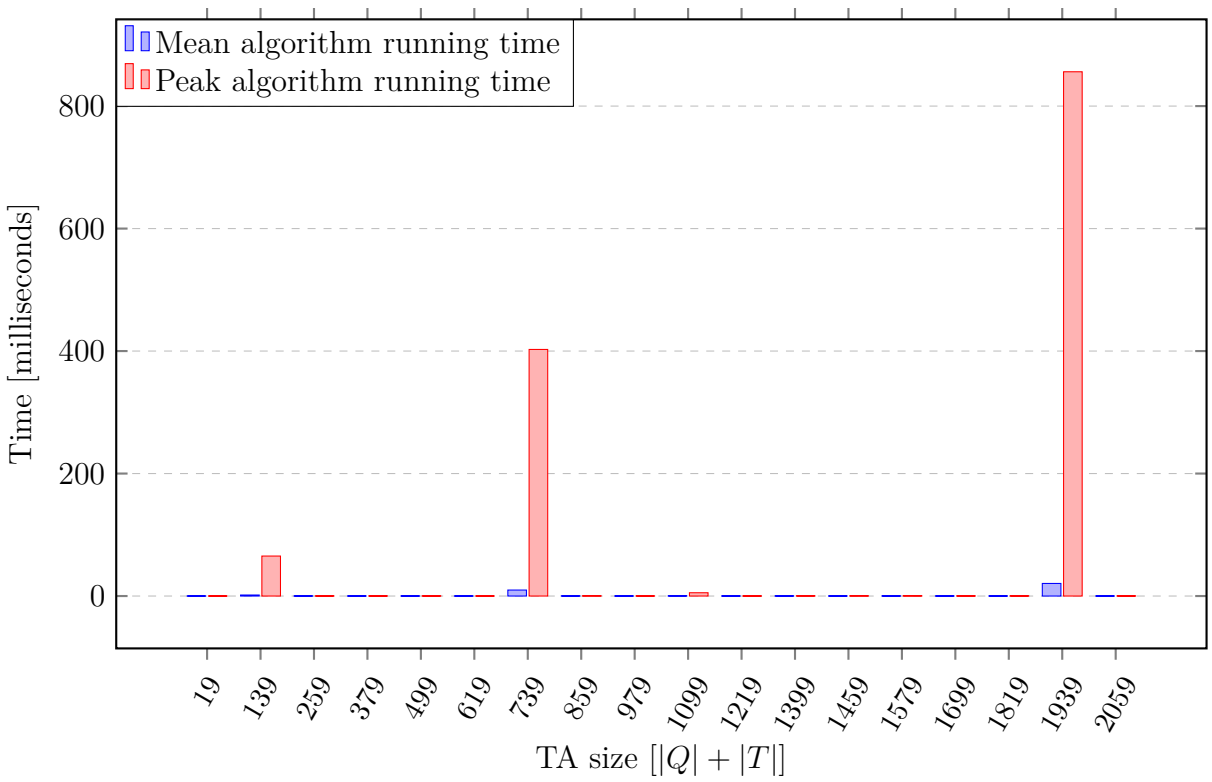

Figure 6.5: Mean and peak running time results for tChecker's algorithm (in milliseconds).

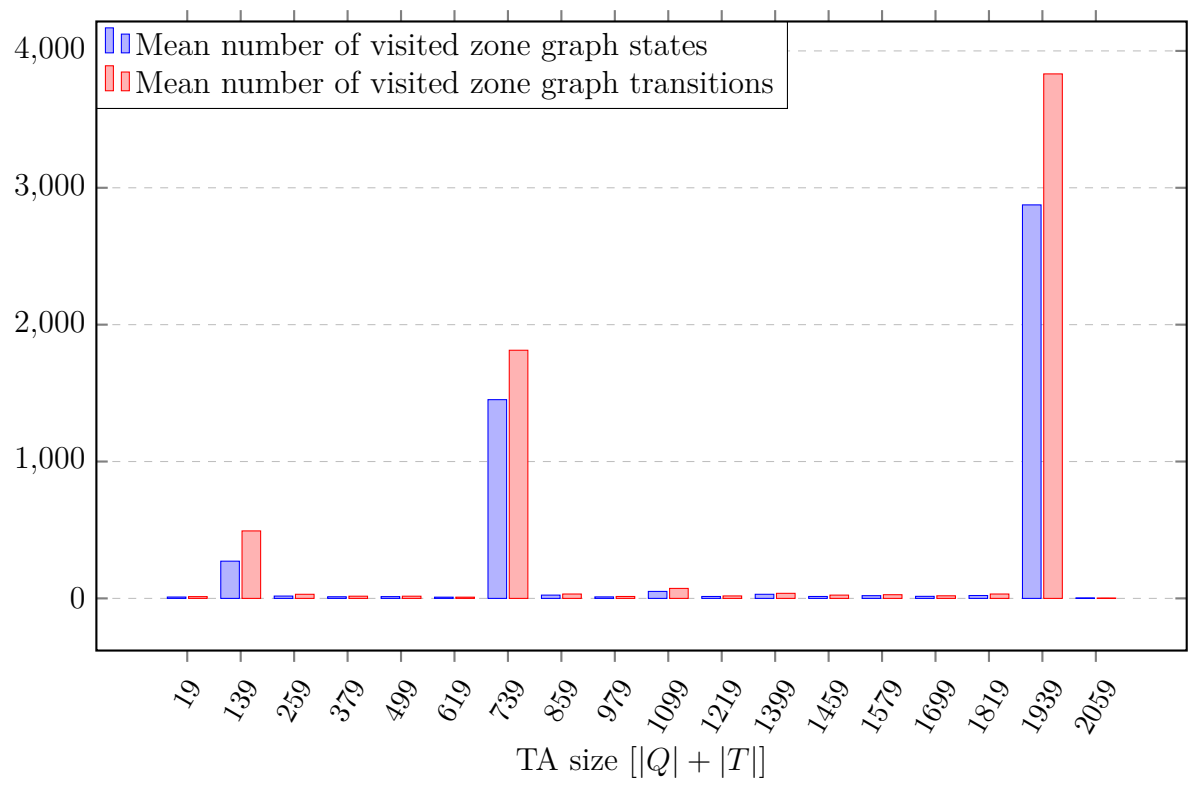

Figure 6.6: Mean number of zone graphs states and transitions visited by tChecker.

7 | Conclusion and future works

This thesis focused on the formal verification of the emptiness problem for parametric nrtTAs with two clocks and one parameter. In particular, the TABEC tool was created to provide an implementation for a decision algorithm able to solve the emptiness problem for this class of TAs. TABEC has also been enriched with oracle capabilities, thanks to the introduction of tiles and Tiled TAs.

Chapter 2 gave an overview of the current State of the Art concerning TAs emptiness results, software testing, and tools used for working with TAs.
Chapter 3 laid down a theoretical background necessary to understand the topics presented in this thesis. Basic definitions of infinite languages, TAs, and PTAs were given. A concise overview of regions and zones was also presented.
Chapter 4 proposed a solution to the parametric nrtTAs emptiness problem by describing both the theoretical results which proved this problem to be decidable and the derived decision algorithm. TABEC was also introduced and the converter, grapher, and checker tools were concisely described. A focus on how the checker tool works was given at the end of the chapter.
Chapter 5 dealt with random testing performed using TABEC. Since the concepts of tiles and TTAs proved to be essential in enriching TABEC with oracle capabilities, a theoretical introduction of these concepts was given, along with some examples. Next, a detailed description of the tester tool implementation was provided, to understand how to obtain a valid parametric TTA starting from a textual description given using the Tl0 DSL. A focus on the oracle capabilities of TABEC was also made. Finally, a description of how the tester tool works was given at the end of the chapter.
Chapter 6 reported the obtained experimental results. Two guided examples were given to understand the meaning of the measured indices. Then, scalability tests were performed to assess the ability of both the tester tool and tChecker to handle inputs of increasing size. Testing about resource utilization was also carried out, to understand how much time and memory were used during test execution.

From experimental results, the proposed algorithm proved to correctly detect the empti-

ness of the given TAs. TABEC proved to correctly forecast the parameter's value interval before executing tests and also to be able to build relatively big TAs without any issues.

While this thesis provided a tool to formally verify the emptiness problem for TAs, in particular for nrtTAs, some future developments and improvements await exploration.
From a practical perspective, TABEC could be upgraded by enriching it with a GUI, enhancing its usability, also losing dependence on Uppaal for building TAs models. In addition, new algorithms for randomly generating TAs could be implemented. Furthermore, TABEC could be tested by using models coming from real-life scenarios rather than relying on randomly generated TAs.
From a theoretical perspective, it could be interesting to study the emptiness decidability for nrtTAs having three clocks and one parameter. These correspond to traditional parametric TAs having two clocks and one parameter, a class for which the emptiness decidability is still an open problem. In addition, theoretical concepts about tiles and parametric TTAs could be further expanded, for example by introducing new tile classes or allowing cycles between different tiles in parametric TTAs.

Bibliography

[1] Rseaux Systmes, Gerard Le Lann, Thme Systmes, and Projet Reflecs. “The Ariane 5 Flight 501 Failure - A Case Study in System Engineering for Computing Systems”. In: *Institut National de Recherche en Informatique et en Automatique* (Feb. 1997).

[2] Nancy G. Leveson. *Safeware: system safety and computers*. Association for Computing Machinery, 1995. ISBN: 0201119722.

[3] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. “Formal methods: Practice and experience”. In: *ACM Comput. Surv.* 41.4 (Oct. 2009).

[4] Alessio Ferrari and Maurice H. Ter Beek. “Formal Methods in Railways: A Systematic Mapping Study”. In: *ACM Comput. Surv.* 55.4 (Nov. 2022).

[5] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008. ISBN: 9780262026499.

[6] Rajeev Alur and David L. Dill. “A theory of timed automata”. In: *Theoretical Computer Science* 126.2 (1994).

[7] Rajeev Alur and P. Madhusudan. “Decision Problems for Timed Automata: A Survey”. In: ed. by Marco Bernardo and Flavio Corradini. Springer Berlin Heidelberg, 2004.

[8] Nikola Beneš, Peter Bezděk, Kim G. Larsen, and Jiří Srba. “Language Emptiness of Continuous-Time Parametric Timed Automata”. In: *Automata, Languages, and Programming*. Ed. by Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann. Springer Berlin Heidelberg, 2015.

[9] Étienne André, Didier Lime, and Olivier Roux. “Reachability and liveness in parametric timed automata”. In: *Logical Methods in Computer Science* Volume 18, Issue 1 (Feb. 2022).

[10] Étienne André. "What's decidable about parametric timed automata?" In: *International Journal on Software Tools for Technology Transfer* 21.2 (2019).

[11] Marcello M. Bersani, Matteo Rossi, and Pierluigi San Pietro. "On Parametric Timed Formalisms". Unpublished manuscript. July 2023.

[12] Alessandro Orso and Gregg Rothermel. "Software testing: a research travelogue (2000–2014)". In: *Future of Software Engineering Proceedings*. Association for Computing Machinery, 2014.

[13] William E. Howden. "Theoretical and Empirical Studies of Program Testing". In: *IEEE Transactions on Software Engineering* SE-4.4 (1978).

[14] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. "The Oracle Problem in Software Testing: A Survey". In: *IEEE Transactions on Software Engineering* 41.5 (2015).

[15] Rolf Schwitter. "English as a formal specification language". In: *Proceedings. 13th International Workshop on Database and Expert Systems Applications*. 2002.

[16] *Uppaal*. URL: `https://uppaal.org`.

[17] *Uppaal documentation*. URL: `https://docs.uppaal.org`.

[18] Patricia Bouyer, Paul Gastin, Frédéric Herbreteau, Ocan Sankur, and B. Srivathsan. "Zone-Based Verification of Timed Automata: Extrapolations, Simulations and What Next?" In: *Formal Modeling and Analysis of Timed Systems*. Ed. by Sergiy Bogomolov and David Parker. Springer International Publishing, 2022.

[19] *tChecker*. URL: `https://github.com/ticktac-project/tchecker`.

[20] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. "Parametric real-time reasoning". In: *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*. Association for Computing Machinery, 1993.

[21] Frédéric Herbreteau, B. Srivathsan, Thanh-Tung Tran, and Igor Walukiewicz. "Why Liveness for Timed Automata Is Hard, and What We Can Do About It". In: *ACM Trans. Comput. Logic* 21.3 (2020).

[22] Andreas Gaiser and Stefan Schwoon. “Comparison of Algorithms for Checking Emptiness on Büchi Automata”. In: *ArXiv* (2009).

[23] Albert-László Barabási. *Network Science*. Cambridge University Press, 2016. ISBN: 9781107076266.

A | Testing results

This appendix contains tables reporting the obtained values when executing resource utilization testing as described in Section 6.3. In particular:

- Table A.1 contains values corresponding to timing results for both tChecker and its SCC-based emptiness check algorithm. These values are depicted in Figure 6.3 and Figure 6.5.
- Table A.2 contains values corresponding to memory occupied by tChecker during its computation. These values are depicted in Figure 6.4.
- Table A.3 contains values corresponding to zone graphs built and exploited by tChecker during its computation. These values are depicted in Figure 6.6.

TA Size	Mean algorithm running time [milliseconds]	Peak algorithm running time [milliseconds]	Mean tChecker running time [milliseconds]	Peak tChecker running time [milliseconds]
19	0.07918	0.07918	4	4
139	1.65716	65.286	84.8095	165
259	0.139585	0.206679	523.167	531
379	0.119486	0.226178	1530.69	1544
499	0.12906	0.213498	3421.1	3444
619	0.0916953	0.11774	6648.17	6722
739	9.76749	402.631	11368.9	11898
859	0.190226	0.406908	17501	17602
979	0.109289	0.171918	25927	26200
1099	0.290151	5.36536	37172.8	37565
1219	0.116995	0.158209	49386.9	49762
1339	0.201608	0.299586	66331.5	66736
1459	0.158296	0.391688	89011.8	90313
1579	0.201357	0.444428	109905	110562
1699	0.134604	0.265487	136941	138104
1819	0.16022	0.255077	169516	171025
1939	20.5466	856.027	203188	208005
2059	0.0935724	0.211579	243948	245095

Table A.1: Timing results of tChecker and its SCC-based algorithm.

TA Size	Mean maximum memory utilization [Bytes]	Peak maximum memory utilization [Bytes]
19	7524	7524
139	8193.81	19988
259	8213.24	8340
379	8530.76	8648
499	8818.29	8940
619	9274.57	9488
739	12023.7	59076
859	12705.1	12896
979	14756.7	14908
1099	16938.3	17124
1219	19384.5	19520
1339	22035.4	22160
1459	24962.7	25140
1579	28052.9	28188
1699	31470.2	31664
1819	34973.2	35156
1939	41000.2	133612
2059	42975.8	43092

Table A.2: Memory results of tChecker.

TA Size	Mean number of visited zone graph states	Mean number of visited zone graph transitions
19	10	13
139	272	493
259	17	30
379	12	16
499	13	16
619	9	9
739	1452	1813
859	24	32
979	11	14
1099	51	73
1219	14	18
1339	30	37
1459	14	24
1579	20	27
1699	15	19
1819	21	32
1939	2875	3832
2059	3	2

Table A.3: Zone graph results of tChecker.

List of Figures

List of Tables