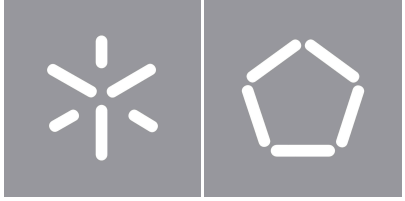
University of Minho
School of Engineering

Alexandre de Jesus Sousa e Silva

Analysis of families of Real Time systems with IMITATOR

september 2025



University of Minho
School of Engineering

Alexandre de Jesus Sousa e Silva

Analysis of families of Real Time systems with IMITATOR

Master's Dissertation in Physics Engineering

Dissertation supervised by
José Miguel Paiva Proença
Renato Jorge Araújo Neves

september 2025

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and context

Model-checking is a complex task that consists of the automatic verification of properties in finite-state systems, such as elevators or vending machines. When applied to real-time systems, this technique becomes even more challenging and often intractable. One of the main limitations of this approach is the large number of states a system can assume, which leads to the so-called state explosion, even when verifying relatively simple properties such as the occurrence of deadlocks. This phenomenon results in very high computational costs, often impractical in the context of the challenges faced by both companies and engineers.

To address this issue, variations of the original model are created, simplifying certain aspects and adjusting requirements according to the objectives of the system. To support and automate this process, we use the Uppex tool **??**, which, through an interface based on Microsoft Excel, reads the configurations chosen by the user. The tool relies on models built in Uppaal and returns an optimized model according to the given specifications. In this way, Uppex extends the applicability of formal verification, making it accessible even to professionals who are not experts in automata theory.

In addition, we use the IMITATOR tool **?**, which is designed for the analysis and verification of real-time systems and serves as an alternative to Uppaal. IMITATOR offers the additional capability of parameter synthesis in real-time systems, enabling the automatic determination of system-specific values that ensure the satisfaction of desired properties. This feature makes the tool particularly valuable, as it provides greater flexibility and allows for further system optimization.

This thesis explored the IMITATOR tool, more specifically by incorporating it as a back-end in the Uppex tool, and investigating whether it could be effectively manipulated through a front-end such as an Excel spreadsheet.

Understanding the syntax of IMITATOR proved to be one of the first crucial steps in the project. The

initial challenges encountered during the development were centered on the conversion of automata into the IMITATOR platform. These challenges were not limited to syntactic disparities but also involved issues related to the specific limitations of the tool itself. The central difficulty was to overcome not only the differences in the way automata are represented, but also to adapt and optimize the use of IMITATOR to meet the needs and peculiarities of the project. By addressing these issues, it became possible to combine the capabilities of IMITATOR with those of Uppex, thereby increasing the efficiency of the integration and expanding the range of problems that could be addressed more comprehensively.

Finally, by enabling the manipulation of IMITATOR through a user-friendly front-end, the tool was made accessible to a broader audience. This approach helped democratize access and simplify the user experience, extending the reach of the tool beyond specialists in the field.

1.2 Organization of this thesis

This thesis is organized as follows: Chapter 2 provides an introduction to model checkers, with several examples, placing particular emphasis on UPPAAL, as it is the model checker used in the Uppex tool. The formal definition of timed automata is presented, complemented by the formal verification of an example using UPPAAL. Next, an introduction to the IMITATOR model checker is given, which is the main focus of this work. This includes the formal definition of parametric timed automata, as well as a detailed explanation of the limitations of this paradigm and the synthesis algorithms provided by IMITATOR. Finally, the concept of Software Product Lines is introduced, followed by a detailed explanation using an example with the Uppex tool.

In Chapter 3, the syntax of IMITATOR is introduced, followed by three examples: Coffee Machine, Worker and Hammer System and an ATM Machine. The aim is to apply the tool to everyday problems and demonstrate its behavior in systems with varying levels of complexity. For each system, a set of properties is verified, highlighting the capabilities of the tool and the types of verification it enables. This section also serves to familiarize the reader with the output file analysis, as well as to explain the information they contain.

Finally, in Chapter 4, the modifications made to the Uppex tool to support IMITATOR as a backend are detailed. Each subsection corresponds to a major change or improvement implemented throughout the development process. To conclude, the Worker and Hammer System example is applied using the updated version of Uppex, demonstrating its extended functionality and the integration of parametric timed automata verification.

Chapter 2

State of the Art

This chapter presents a brief explanation of the Model Checking technique followed by a general overview of some alternative model checkers. Next, the two main model checkers that are the focus of this work: Uppaal and Imitator, will be addressed in greater depth. Finally, a detailed explanation of the Uppex tool is presented, accompanied by an example that demonstrates its functionality.

2.1 Model Checking real-time systems

Model Checking is an automatic verification technique for finite state (concurrent) systems. Its a very important step in the development of a system to find errors and guarantee correct function **?**. A summarized picture of the model checking process is given in Figure 1.

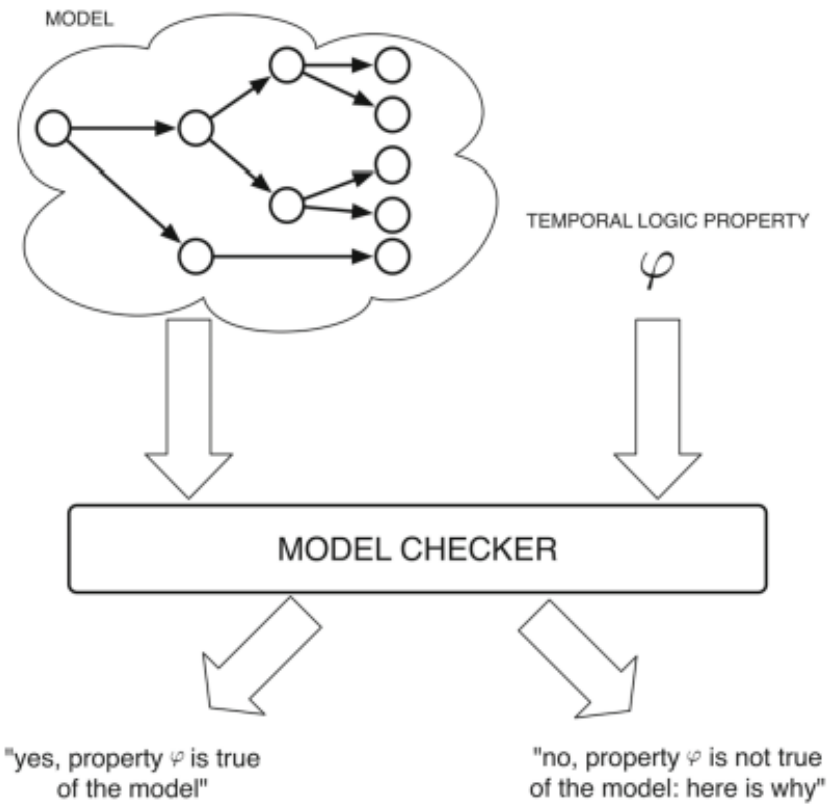


Figure 1: Model Checking Overview **?**.

As presented in the picture, one has a set of requirements that need to be satisfied, usually given in a temporal logic (CTL, LTL), stating how the behavior of the systems evolves over time. One also needs to model the system under analysis in such a way that the model checking tool understands it, typically as a Transition System.

There are many model checkers available, and what mainly distinguishes them is the algorithm they use, since each one adopts different optimization strategies, that is, they all solve the same verification problem but rely on different techniques(add here ref). To start this project, we present different examples of models checkers with**Imitator** and **Uppaal** on the spotlight as previously mentioned. It is within these two programs that the systems in Uppex will be modeled and analyzed.

SPIN

SPIN is an LTL model checker commonly used for modeling concurrent software and asynchronous processes, particularly communication protocols. It utilizes Promela as the modeling language. After we create our model, it offers the option to simulate the model randomly, interactively, or in a guided manner. Additionally, SPIN can create a C program that automatically performs exhaustive model checking **?**.

The verifier conducts exhaustive model checking, identifying violations of specified properties, such as safety and correctness. This methodology proves particularly valuable in scenarios where correctness and security are critical considerations, as is often the case in communication protocol applications (Figure 2).

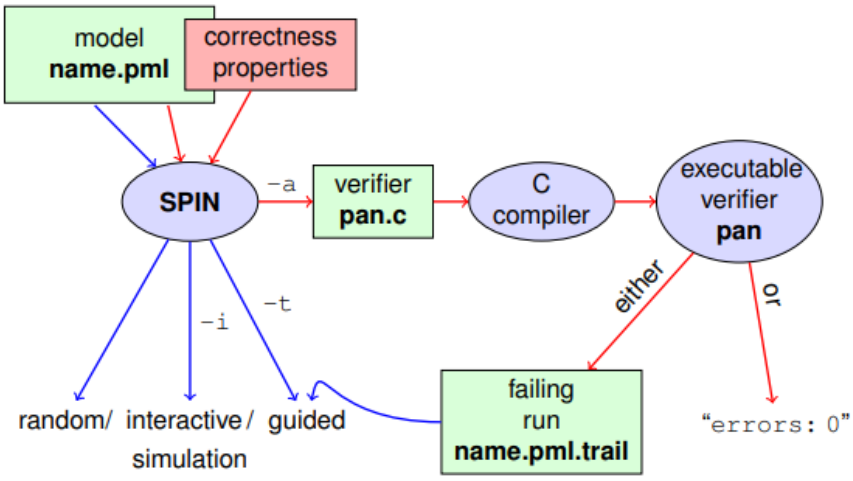


Figure 2: Spin Process **?**.

Prism

Prism is a Probabistic Model Checking tool that deals with systems that exhibit probabilistic behavior. Probabilistic model checking refers to a range of techniques for calculating the likelihood of the occurrence of certain events during the execution of systems **?**. As a input, Prism takes a probabilistic model, including *Continuous-time Markov chains* and *Probabilistic timed automata*, and a probabilistic temporal logic **?**. This is summarized in Figure 3.

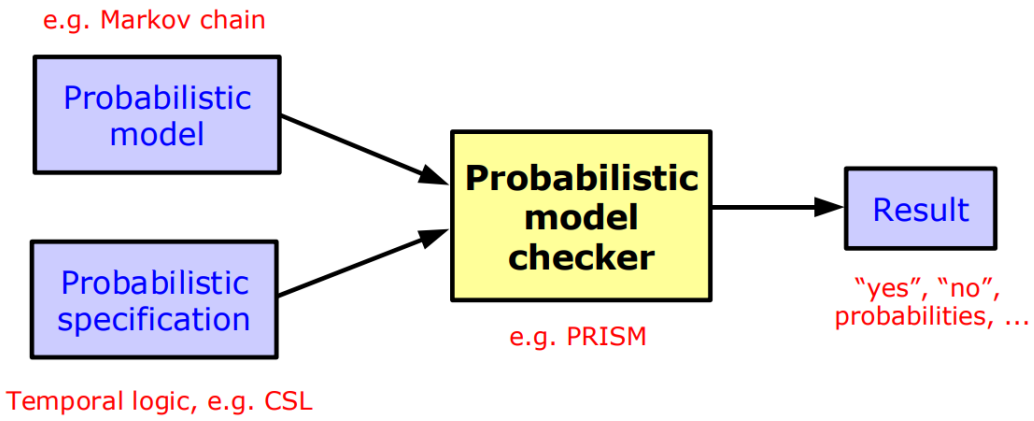


Figure 3: Prism Process **?**.

Simulink

Simulink is a Matlab extension that is widely used to *model* and *simulate* dynamical systems **?**. It is particularly suitable for specifying mathematical models of real time systems. Simulink has two different approaches to ensure that failures are identified as early as possible.

- **Model testing** attempts to identify failures in models by executing them for some test inputs sampled by a guided randomized algorithm **?**.
- **Model checking** attempts to exhaustively verify the correctness of models against some given formal properties **?**.

In the case of Model checking, the inputs (system and property) are translated and used as inputs in some model checker or Satisfiability Modulo Theories (SMT) solvers. Since Simulink models often represent continuous dynamic and hybrid systems, applying model checking to such models typically leads to undecidable **?**.

2.2 Uppaal in a Nutshell

As mentioned previously, Uppaal is a model checking tool for verification of real time systems jointly developed by Uppsala University and Aalborg University. In order to do that, Uppaal adopts the concept of Timed (Finite) Automata as model system (transistion system) **?**.

Timed Automata

A *Timed Automaton* (TA) extends the concept of a Finite Automaton with clocks that are used to handle time in the system. Time is continuous and the clocks measure time progress which will increase globally at the same pace for the whole system. It is formally defined as a 6-tuple **?**:

$$\langle L, L_0, Act, C, Tr, Inv \rangle$$

Where:

- L is the set of **locations** (or states).
- $L_0 \subseteq L$ is the subset of **initial locations**.
- Act is the set of **actions**, and C is the set of **clocks**.

Before defining transitions, it is crucial to introduce the concept of a reset. In a timed automaton, the value of a clock can be reset during transitions.

The power set $P(C)$ represents all possible subsets of the clocks C that could be reset during a transition **?**. For example, if $C = \{x, y\}$, then the power set of C is:

$$P(C) = \{\emptyset, \{x\}, \{y\}, \{x, y\}\}$$

This describes the four possible ways clocks can be reset:

- $\emptyset$: No clocks are reset.
- $\{x\}$: Only clock x is reset.
- $\{y\}$: Only clock y is reset.
- $\{x, y\}$: Both clocks x and y are reset.

The notation $\mathcal{C}(C)$ denotes the set of clock constraints over a set C of clock variables. Each constraint is formed according to **?**:

$$g ::= x \sqsquare n \mid x - y \sqsquare n \mid g \wedge g \mid \text{true}$$

where $x, y \in C$, $n \in \mathbb{N}$, and $\sqsquare \in \{<, \leq, >, \geq, =\}$. These will be used in **guards** (on the transitions) or in **invariants** (on the locations). It is worth noting that invariants are the only way to enforce a transition **?**.

We can also define the Invariant Function (Inv) as follows:

- $Inv : L \to \mathcal{C}(C)$ is the **invariant function** that assigns a clock constraint to each location $\ell \in L$.

With this concepts in place, we can now complete the formal notation for transitions:

- $Tr \subseteq L \times \mathcal{C} \times Act \times \mathcal{P}(C) \times L$ defines the **transition relation**, i.e., the rules that determine how the system evolves.

The transition notation:

$$\ell_1 \xrightarrow{g,a,U} \ell_2$$

means that:

- The system can transition from location ℓ_1 to ℓ_2 if the **guard** g is valid.
- The transition is labeled by an **action** a.
- When the transition occurs, the **clocks** in set U are reseted.

Timed Labelled Transition Systems (TLTS) models the execution of TA through labeled transition systems, incorporating so-called delay transitions to capture the passage of time.

A TLTS is a model that introduces two types of transitions **?**:

- **Ordinary transitions**: Associated with actions, denoted by:

$$s \xrightarrow{a} s' \quad \text{where} \quad a \in \text{Act}.$$

- **Delay transitions**: Represent the passage of time, denoted by:

$$s \xrightarrow{d} s' \quad \text{where} \quad d \in \mathbb{R}_0^+.$$

These transitions must obey certain constraints **?**:

- **Time additivity**: If $s \xrightarrow{d} s'$ and $0 \leq d' \leq d$, then there exist intermediate states s'' such that:

$$s \xrightarrow{d'} s'' \xrightarrow{d-d'} s'.$$

- **Determinism of delays**: If $s \xrightarrow{d} s'$ and $s \xrightarrow{d} s''$, then necessarily $s' = s''$.

In terms of semantics, each TA defines a TLTS $T(TA)$ whose states are pairs of the form **?**:

$$\langle \text{location, clock valuation} \rangle.$$

Transitions follow specific rules, and if a transition depends on a clock x, it only occurs if the associated time constraint is valid. Time can advance as long as the current state still satisfies its constraints.

Clocks are valuation functions $\eta : C \to \mathbb{R}_0^+$, mapping each clock x to its current value. The main operations on clocks are **?**:

- **Delay**: Increases the value of all clocks, defined by:

$$(\eta + d)(x) = \eta(x) + d.$$

- **Reset**: Resets certain clocks, defined by:

$$\eta[R](x) = \begin{cases} 0, & \text{if } x \in R, \\ \eta(x), & \text{otherwise.} \end{cases}$$

More specifically the conversion of a Timed Automaton to a TLTS $T(ta) = \langle S, S_0, N, T \rangle$ occurs as follows **?**:

- S is the set of states;
- S_0 is the set of initial states;
- N is the set of transition labels, including actions and delays;
- T is the set of transitions, defined by:

$$\langle l, \eta \rangle \xrightarrow{a} \langle l', \eta' \rangle \quad \text{if there is a valid transition in } Tr.$$

The delay transition occurs as long as the state invariants are maintained.

Modelling in Uppaal

A system in Uppaal is formally modeled as a network of Timed Automata (TA) that operate in parallel through a composition approach **?**. This network consists of multiple interacting TA processes, where each automaton contains locations (discrete states) and transitions that enable movement between these locations. The behavior is controlled through clock constraints expressed as guards on transitions and invariants on locations, which enforce timing requirements. Additionally, synchronization between different automata is achieved through complementary actions, allowing coordinated behavior across the parallel components **?**. During a Transistion we can also update the value of a variable or clock a **?**.

It is now possible to build a model using Uppaal. As an example, it will be built a Worker and Hammer system (4), as mentioned before.

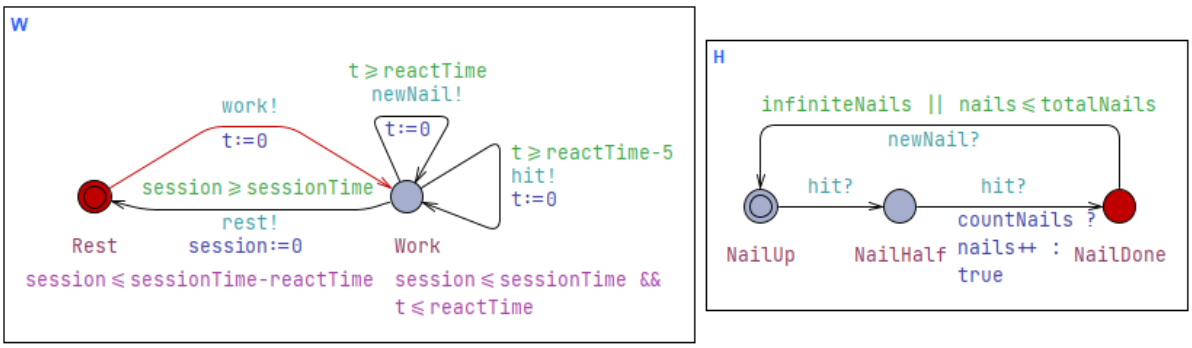


Figure 4: Worker/Hammer Automata Example. The image on the left automaton represents the Worker, while the on on the right represents the Hammer.

The Worker automaton consists of two states, as shown in Figure 4: Rest (the initial state) and Work. When the invariant condition is satisfied, a transition to the Work state is triggered, synchronized with the event press!, and the clock t is reset.

In the Work state, there are three possible transitions, although one remains excluded as long as the state invariant holds. If t > reactTime, the transition newNail is executed and the clock t is restarted. If t > reactTime – 5, the synchronization hit is performed, enabling the Hammer automaton to start its execution. When the invariant of the state is no longer satisfied, and the guard of the final transition is met, the automaton returns to the initial state, the session clock is reset, and the process begins again.

In Figure 4, the automaton on the right represents the Hammer. It is composed of three states, none of which contain invariants. The first transition occurs only when the hit synchronization is activated. The transition from NailHalf to NailDone takes place after a second activation of hit, during which the variable countNails is updated to a predefined value. The final transition, which returns the automaton to its initial state, is possible only if the logical conditions remain valid and the newNail synchronization is activated.

To ensure the complete functionality of the model, it is essential to declare the variables used in the Uppaal specification. The syntax for these declarations is very similar to that of C, and variables may be defined as either global or local.

It is important to note that sessionTime, reactionTime, totalNails, totalNail, and countNails are constants, not parameters. Uppaal does not support parameters in clocks, only in variables, when defining a generic process (Timed Automata). In this example, sessionTime is set to 100, reactionTime to 20, totalNails to 10, and both countNails and infiniteNails are set to True.

Verification in Uppaal

The next phase is to check that the model verifies the properties and to do so, Uppaal also has a window dedicated to that (verification window). With Uppaal query language it can be specified the following properties (mentioned above):

- **Reachability properties:** A specific condition that holds in some state of the model's potential behaviours.
- **Safety properties:** A specific condition that holds in all the states of an execution path.
- **Liveness properties:** A specific condition is guaranteed to hold eventually.

These properties can be described using a simple CTL logic variation (TCTL)**?**. It is a branching-time temporal logic used to formally specify and verify properties of systems, and it is based on the idea that all possible execution paths a system can take can be represented in the form of a "Computation Tree" **?**. The formulas should be one of the following **?**:

- A[] ϕ — Invariantly ϕ
- E<> ϕ - Possibly ϕ
- A<> ϕ - Always Eventually ϕ
- E[] ϕ - Potentially Always ϕ
- $\phi \longrightarrow \psi$ - ϕ leads to ψ

The symbols ϕ and ψ are formulae that describe state properties, such as what is the current state and clock constrains. Returning to the worker and hammer example, we can illustrate some properties using this notation. Notice that Deadlock is expressed using a special formula and IT consists of the keyword deadlock.

- **A[]!deadlock** Deadlock never occurs.
- **A[] W.Rest** The Worker always reaches the Rest state.
- **E<> H.NailDone** The Hammer will eventually finish Nailing.
- **A<> H.NailDone** The Hammer eventually reaches the NailDone state on all paths.

- **W.Work** $\longrightarrow$ **W.Rest** If the Worker is in the Work state, then it will eventually reach the Rest state.

These properties can be verified in the Verifier. The Verifier is a tool that checks if your model behaves as expected. If the property is marked in red, it means 'not satisfied' (the model does not guarantee the property). Otherwise, it is 'satisfied' (the model always meets the condition).

2.3 Imitator in a Nutshell

IMITATOR is command-line only tool to perform automated parameter synthesis for parameter timed systems. This model checker takes as input a network of Parametric Timed Automata. This type is a sub category of timed automata where clocks can be declared as parameters (without a constant pre-determined), making it particularly useful for modeling **Real-Time Systems** and **Software Product Lines (SPLs) ?**. In real-time systems, timing constraints often depend on environmental conditions or system configurations. By allowing certain timing constraints to remain as parameters rather than fixed values, Parametric Timed Automata enable the analysis of multiple scenarios, ensuring that the system behaves correctly under various conditions **?**. Similarly, in Software Product Lines (SPLs), where models need to be adapted and reused across different products or configurations, the use of parametric timing constraints avoids the need to create separate automata for each variation. A single model can be created where critical variables can be written as paramaters. This enhances flexibility and reduces redundancy in system modeling **?**.

Imitator is a model checker specialized in analyzing parametric systems. Its strength lies not only in verifying whether system properties hold but also in determining the exact conditions (parameter constraints) under which they are satisfied. By using Imitator, we can systematically evaluate a model and check if the properties defined in the property.imiprop file are valid, while also obtaining the precise parameter values required for them to hold. Moreover, Imitator goes beyond mere verification by offering insights into the constraints under which these properties are valid. This information is valuable for understanding the system's behavior and can guide further refinement or optimization to enhance the system's reliability and performance **?**.

Parametric Timed Automata

A *Parametric Timed Automaton* (PTA) is a generalization of TA that introduces parameters in the guards and resets of clocks, allowing the modeling of systems where certain temporal values are unknown or variable. Unlike TA, where timing bounds are fixed, PTA enable symbolic analysis and the synthesis of constraints on these parameters. It is formally defined as a 7-tuple **?**.

$$\langle L, L_0, Act, C, Tr, P, Inv \rangle$$

Where:

- L is the set of **locations** (or states).
- $L_0 \subseteq L$ is the subset of **initial locations**.
- Act is the set of **actions**, and C is the set of **clocks**.
- P is a finite set of parameters.
- $Tr \subseteq L \times \mathcal{C}(C) \times Act \times \mathcal{P}(C) \times L$ defines the **transition relation**, i.e., the rules that determine how the system evolves.

The transition notation:

$$\ell_1 \xrightarrow{g,a,U} \ell_2$$

Means that:

- The system can transition from location ℓ_1 to ℓ_2.
- This transition occurs if the **guard** g is valid.
- The transition is labeled by an **action** a.
- When the transition occurs, the set of **clocks** U is reset.

The power set $P(C)$ represents all possible subsets of the clocks C that could be reset during a transition **?**. For example, if $C = \{x, y\}$, then the power set of C is:

$$P(C) = \{\emptyset, \{x\}, \{y\}, \{x, y\}\}$$

This describes the four possible ways clocks can be reset:

- $\emptyset$: No clocks are reset.
- $\{x\}$: Only clock x is reset.
- $\{y\}$: Only clock y is reset.

- $\{x, y\}$: Both clocks x and y are reset.

The notation $\mathcal{C}(C)$ denotes the set of clock constraints over a set C of clock variables. Each constraint is formed according to:

$$g ::= x \square n \mid x - y \square n \mid g \wedge g \mid \text{true}$$

where $x, y \in C$, $n \in \mathbb{N}$, and $\square \in \{<, \leq, >, \geq, =\}$. These will be used in **guards** (on the transitions) or in **invariants** (on the locations). It is worth noting that invariants are the only way to enforce a transition **?**.

Regarding the semantics of the automaton, we define a *parameter valuation* v as a function that assigns concrete (numerical) values to the parameters $p_1, p_2, \ldots, p_n$ in the automaton. The result of applying v to a parametric timed automaton (PTA) A is denoted by $v(A)$, which corresponds to a non-parametric timed automaton — that is, all parameters are replaced with specific constant values.

Given a PTA $A = (\Sigma, L, l_0, X, P, I, E)$ and a parameter valuation v, the *concrete semantics* of $v(A)$ is defined by a *timed labelled transition system*, as in the case of TA.

A TLTS (Timed Labelled Transition System) introduces two types of transitions:

- **Ordinary transitions**: Associated with actions, denoted by:

$$s \xrightarrow{a} s' \quad \text{where} \quad a \in \text{Act}.$$

- **Delay transitions**: Represent the passage of time, denoted by:

$$s \xrightarrow{d} s' \quad \text{where} \quad d \in \mathbb{R}_0^+.$$

These transitions must obey certain constraints:

- **Time additivity**: If $s \xrightarrow{d} s'$ and $0 \leq d' \leq d$, then there exist intermediate states s'' such that:

$$s \xrightarrow{d'} s'' \xrightarrow{d-d'} s'.$$

- **Determinism of delays**: If $s \xrightarrow{d} s'$ and $s \xrightarrow{d} s''$, then necessarily $s' = s''$.

In terms of semantics, each Timed Automaton (TA) defines a TLTS $T(ta)$ whose states are pairs of the form:

$$\langle \text{location}, \text{clock valuation} \rangle.$$

Transitions follow specific rules, and if a transition depends on a clock x, it only occurs if the associated time constraint is valid. Time can advance as long as the current state still satisfies its constraints.

Clocks are valuation functions $\eta : C \to \mathbb{R}_0^+$, mapping each clock x to its current value. The main operations on clocks are:

- **Delay**: Increases the value of all clocks, defined by:

$$(\eta + d)(x) = \eta(x) + d.$$

- **Reset**: Resets certain clocks, defined by:

$$\eta[R](x) = \begin{cases} 0, & \text{if } x \in R, \\ \eta(x), & \text{otherwise.} \end{cases}$$

The conversion of a Timed Automaton (TA) to a TLTS $T(ta) = \langle S, S_0, N, T \rangle$ occurs as follows:

- S is the set of states;
- S_0 is the set of initial states;
- N is the set of transition labels, including actions and delays;
- T is the set of transitions, defined by:

$$\langle l, \eta \rangle \xrightarrow{a} \langle l', \eta' \rangle \quad \text{if there is a valid transition in } Tr.$$

The delay transition occurs as long as the state constraints are maintained.

Limitations in Parametric Timed Automata

Introducing parameters in real-time temporal logic can indeed lead to undecidability issues, notably when such parameters are unbounded. On top of this, the use of multi-rate automata together with linear constraints on clocks also leads to undecidability, as does the use of stopwatches **?**. The addition of parameters, increases the complexity of the model checking of these systems, this complexity often results in undecidability, meaning that there is no general algorithm that can determine the validity of formulas within the logic. In fact, most interesting problems that are decidable in timed automata become undecidable in Parametric Timed Automata, including the ones below **?**.

- **EF-emptiness**

  Is the set of parameter valuations for which a given location l is reachable empty?

- **EF-universitily**

  Do all parameters allow to reach a given location l?

- **AF-emptiness**

  Is the set of parameter valuations for which all runs eventually reach a given location l empty?

- **AF-universitily**

  Do all parameters allow to reach a given location l for all runs?

- **Language and Trace Preservation**

  For a given set of timing parameters, are there other parameter values that yield the same sequence of discrete events?

However, there are adaptations that can be applied to the automaton to ensure the decidability of certain problems. For example, reducing parameters, parametric clocks, and non-parametric clocks result in the decidability of the EF-emptiness problem(Parametric clocks are time variables in parametric timed automata whose behavior depends on parameters that are unknown or undefined a priori) **?**. Another approach is through the restriction of parameter usage, utilizing a subcategory of Parametric Timed Automata called Lower/Upper Bound Parametric Timed Automata (L/U PTA), where all parameters are compared with the clocks either as an upper bound or a lower bound. With this restriction, several verification problems that are generally undecidable for general PTA become decidable **?**. In particular, the EF-emptiness problem (determining whether there exists an instantiation of parameters such that a given state is reachable), the EF-universality problem (checking whether all possible parameter valuations allow the reachability of a given state), and the EF-finiteness problem (determining whether the number of parameter valuations leading to reachability is finite) become decidable. These problems, which are crucial for analyzing the behavior of parametric timed systems, fall within PSPACE complexity when restricted to L/U PTA, ensuring their computational feasibility **?**. However, not all verification problems benefit from this restriction. Certain problems remain undecidable even under general PTA constraints. For instance, the AF-emptiness problem (determining whether there exists a parameter valuation such that all runs eventually reach a given state) is undecidable. Similarly, language preservation, which checks whether the language of a PTA remains unchanged under all possible parameter valuations, is also undecidable. Additionally, the EG-emptiness problem (verifying whether there exists a parameter valuation such that some execution remains within a given set of states indefinitely) remains an open question **?**.

Verification in Imitator

To synthesize the system's parameters that validate the intended property based on the desired properties, Imitator offers two approaches to the parameter synthetise**?**:

- **Synthesis (Synth)**

  IMITATOR attempts to synthesize all parameter valuations satisfying the property.

- **Witness**

  IMITATOR attempts to exhibit at least one parameter valuation satisfying the property.It works like syntesis but stops the analysis as soon as one such set is found

IMITATOR follows the "Best Effort" paradigm, meaning that attempts to synthesize a set of parameter valuations without guaranteeing termination, due to the undecidable nature of most problems in parametric timed automata. To ensure that the computation eventually halts, the tool relies on approximations, which often results in outcomes that are not sound or complete, meaning the results may include some incorrect parameter values (not sound) or miss some valid ones (not complete). The tool classifies its results into three categories and informs the user about them **?**.

- **Exact**

  The synthesis found all and only the correct valuations.

- **Over-approximated**

  The set may contain invalid solutions.

- **Under-approximated**

  Some correct solutions may be missing.

- **Possibly invalid**

  There are no guarantees that the solution is correct.

In terms of properties, IMITATOR supports a vast number of properties, including reachability ("EF"), safety ("AG not"), liveness, deadlock-freeness and trace preservation (robustness). Additionally, other algorithms will be discussed in the following chapters, such as Minimum-Time Reachability, which aims to synthesize parameter valuations that minimize the time required to reach a given state, and Optimal

Parameter Reachability, which determines the minimum or maximum value of a parameter when reaching a specific state **?**. A summary of these properties and their corresponding syntax can be found in Table 1.

The result is displayed in the terminal as a printout, showing the constraint applied to the parameters if they exist in the model. Otherwise, the output will be "True" or "False." Additionally, the result can be complemented with a graphical visualization, which can be requested at the end of the command using the options parameter as mentioned above. We will see this applied to examples in the next chapter.

Table 1: Some Properties and their correspondence syntax in IMITATOR **?**

Property	**Syntax Examples**
Deadlock Free	property := synth DeadlockFree ;
Safety	property := synth AGnot(State Predicate);
Reachability	property := synth EF(State Predicate);
Optimal parameter reachability	property := synth EFpmin/pmax(State Predicate,Parameter);
Liveness	property := synth CycleThrough(State Predicate);
Robustness	property := synth TracePreservation(Parameters);

Other interesting properties are:

- **Inverse Method**
- **Behavioral cartography**
- **Minimal-time reachability**

It is important to note that the CycleThrough algorithm does not cover all Liveness properties. It only verifies whether there exists a cycle in the system (infinitely executable) that visits a given location infinitely often—that is, it ensures that a certain state will eventually be visited and will continue to be visited over time. More complex properties, such as verifying that whenever A occurs, eventually B occurs, or that all executions eventually reach a state X, must be verified using a different algorithm written under the observer formalism **?**.

An observer is a parametric timed automaton placed in parallel with the system (usually referred to as the Test Automaton) that neither alters nor constrains its behavior, acting instead as a trigger when a given property is violated. This approach allows more complex properties, such as Safety or Liveness, to be reduced to a Reachability property **?**.

The synthesis process generates a text file containing a detailed description of the system, along with computational information related to the synthesis, including the type of approximation performed. The key result of interest, the parameter constraint, is located in the section marked by `BEGIN CONSTRAINT` and `END CONSTRAINT`. An example of this file can be found in `Appendix B`, with the mentioned section clearly identified.

2.4 Uppex in a Nutshell

Uppex is a tool that facilitates the creation of a family of software product lines using a list of features defined in an Excel spreadsheet **?**. The process begins by defining configurations, where each configuration represents a unique combination of features. Features correspond to system characteristics, such as "slow" or "fast". Not all feature combinations are valid, some may be semantically inconsistent. To prevent such invalid combinations, Uppex uses a separate sheet where features are organized in a tree-like structure. In this structure, features located on the same leaf cannot be selected simultaneously, ensuring that only meaningful configurations are generated. Additionally, a Limits and Queries sheet is maintained, in which queries are defined through formulas and constants are assigned predefined values according to the configuration being executed. To execute, it is necessary to ensure that both the JVM and Uppaal are installed. Once verified, the latest .jar file can be used to run the command uppex runAll model.xml.

Overview of Software product lines

We can define a A software product line (SPL) as a technique to describe and develop a family of software products with many common artefacts, which are called *features*. Each valid combination of features describes a possible software product from this SPL. This techniques have been used by many companies, facilitating the development, analysis, and deployment of several large software artefacts with commonalities **?**. There are many ways to implement SPL, which can be grouped into two groups **?**:

- **Compositional Approaches**
- **Annotative Approaches**

In Compositional Approaches, features are treated as separate systems or code fragments that extend the base code. The key idea is to compose them in parallel. After defining certain features, they

are integrated into the system. This integration can be done in different ways, ranging from simple concatenation to complex code transformations. Among the various types of parallel composition, the most notable are: **feature-oriented programming**, **aspect-oriented programming**, and **delta-oriented programming**.

In the other hand we have Annotative Approaches where annotations or markings are used to control the presence or absence of the Features that we want to add to the system (in this case they are already present in the base code). For example, in C and C++ programming languages, preprocessor directives such as ifdef, ifndef, else, and endif are used to conditionally include or exclude blocks of code during compilation.

Overview of Uppex

Uppex is a tool developed in Scala that uses Apache POI libraries to read Microsoft documents. Its main goal is to simplify the construction and verification of multiple Uppaal models generated from different configurations of a single file, following the principles of Software Product Lines previously studied. The tool relies on an annotation mechanism to allow customization of various parts of the model, such as channels, shared variables, data types, time bounds, and requirements.

Uppex reads an Excel file containing the configurations and a Uppaal file with the source model with annotations, and generates a new Uppaal file for each configuration found. Each generated file replaces the original model and is verified by Uppaal. The output is an HTML report summarizing the generated configurations and the properties that were verified. The workflow follows the structure illustrated in Figure 5.

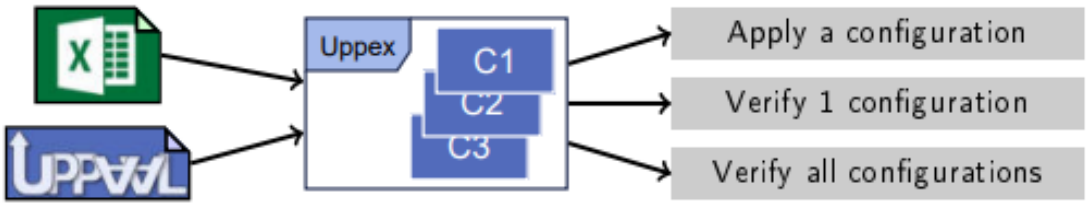


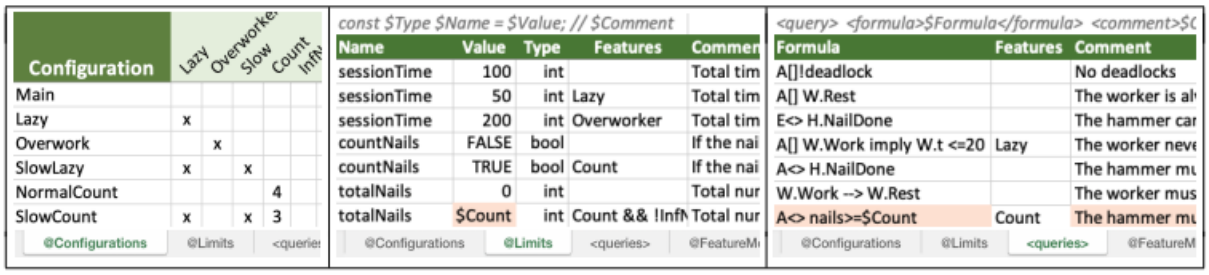
Figure 5: Uppex WorkFlow **?**

The declarations that indicate annotations in the Uppaal file follow the format *"// @Name"* and continue until a blank line is encountered. These serve as hooks that allow Uppex to inject and update the values used to configure the model, and are referred to simply as annotations. To inject and update the queries to be verified for each configuration, a different annotation is used, called a XML annotation. As the

name suggests, it encompasses blocks that start with *"<Name>"* and end with *"</Name>"*. Each of these annotations is defined in the Excel file on a sheet with the same name. The first cell of the sheet specifies the pattern to be followed in order to generate the code that will be injected into the annotation block with the corresponding name in the Uppaal file. This is followed by a table where one column contains the name of the variable to be inserted, and the next columns specify the type and value of the variable separately. Additionally, there is a column named Feature that indicates which variables are active in the current configuration, discarding the others. In cases where duplicate variable names exist, only the last occurrence is considered. The configuration declarations are defined in the "@Configurations" sheet, with each configuration representing a combination of the existing features.To control the valid combinations of features, there is another sheet named "@FeatureModel", where the features are organized in a tree structure. In this structure, features that share the same root node cannot be selected simultaneously.

Revisiting the Worker and Hammer example

We will use the Worker and Hammer system to illustrate our description.



Configuration	Lazy	Overworker	Slow	Count	InfN
Main					
Lazy	x				
Overwork		x			
SlowLazy	x		x		
NormalCount				4	
SlowCount	x		x	3	

@Configurations | @Limits | <queries>

const $Type $Name = $Value; // $Comment

Name	Value	Type	Features	Commen
sessionTime	100	int		Total tim
sessionTime	50	int	Lazy	Total tim
sessionTime	200	int	Overworker	Total tim
countNails	FALSE	bool		If the nai
countNails	TRUE	bool	Count	If the nai
totalNails	0	int		Total nur
totalNails	$Count	int	Count && !InfN	Total nur

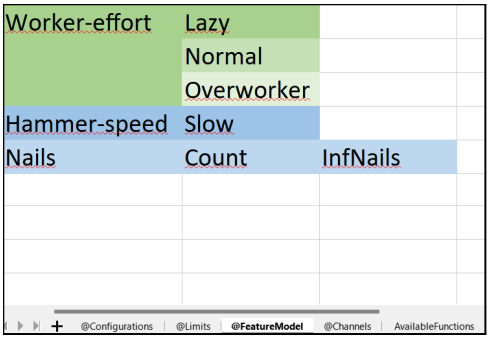
@Configurations | @Limits | <queries> | @FeatureM

<query> <formula>$Formula</formula> <comment>$C

Formula	Features	Comment
A[]!deadlock		No deadlocks
A[] W.Rest		The worker is al
E<> H.NailDone		The hammer car
A[] W.Work imply W.t <=20	Lazy	The worker neve
A<> H.NailDone		The hammer mu
W.Work --> W.Rest		The worker mus
A<> nails>=$Count	Count	The hammer mu

@Configurations | @Limits | <queries> | @FeatureM

Figure 6: Uppex Configuration, Limits and Queries Sheets



Worker-effort	Lazy	
	Normal	
	Overworker	
Hammer-speed	Slow	
Nails	Count	InfNails

@Configurations | @Limits | @FeatureModel | @Channels | AvailableFunctions

Figure 7: Uppex Feature Model Sheet

Figure 6 represents the Configurations, Limits, and Querie sheets, respectively. Assuming Lazy is the active configuration, the annotation block is rewritten using the following pattern:`const $Type $Name`

= $Value; //$Comment, where the $ notation identifies the column from which the value should be extracted and replaced. Furthermore, as previously explained, only the values whose Feature matches the selected configuration (in this case, Lazy) are left blank will be considered.

```
// @Limits
const int sessionTime = 50; // Total time that the worker can work
                            // before stopping
const bool countNails = false; // If the nails should be counted.
// If so, the system will have an infinite amount of states.
const int totalNails = 0; // Total number of new nails available
                          // (when counting nails)
const bool infiniteNails = false; // If there is no bound on the
                          // number of nails (can cause overflows)
const int reactTime = 20; // Maximum time that the worker can take to
                          // hit or place a new nail
```

Figure 8: Limits Block Rewritten with new values

The “<queries>” sheet follows the same logic, but it refers to a XML annotation, using the following pattern: `const $Type $Name = $Value; // $Comment`

```
<queries>
<query>
  <formula>A[]!deadlock</formula>
  <comment>No deadlocks</comment>
</query>
<query>
  <formula>A[] W.Rest</formula>
  <comment>The worker is always resting</comment>
</query>
<query>
  <formula>E&lt;&gt; H.NailDone</formula>
  <comment>The hammer can finish a nail</comment>
</query>
<query>
  <formula>A&lt;&gt; H.NailDone</formula>
  <comment>The hammer must complete a nail</comment>
</query>
<query>
  <formula>W.Work --&gt; W.Rest</formula>
  <comment>The worker must be able to rest after working</comment>
</query>
</queries>
```

Figure 9: Queries Block Rewritten with new values

It is also important to highlight the constraints on feature combinations illustrated in Figure 7. For instance, the user cannot select both Lazy and Overworker in the same configuration, thus preventing illogical or inconsistent combinations. Once the Excel configuration is complete, Uppex can be executed. The tool is distributed as a .jar file, which means it can be run directly from the command line. Uppex offers several execution options, allowing for flexible use depending on the user's needs. For example, it is possible to:

- Run a specific configuration only (e.g., run the configuration named Main);

  ```
  java -jar uppex.jar --run -p Main Model.xml
  ```

- Validate the model and input files before generating the Uppaal models;

  ```
  java -jar uppex.jar --validate Model.xml
  ```

- Run all configurations defined in the Excel file, generating one Uppaal model for each and verifying the corresponding properties;

  ```
  java -jar uppex.jar --runAll Model.xml
  ```

As an example, we will choose the last option and run all configurations. When executed, Uppex generates an HTML report as output, summarizing the models and properties analyzed for each configuration. For the previous example, the resulting report can be found in the attachment.

2.5 Proposed extensions to Uppex

With the Imitator Model Checker and the Uppex tool introduced, the following chapters will detail the integration of Imitator into Uppex's backend as an alternative Model Checker to Uppaal. This integration also brought improvements to the types of supported features, the range of accepted properties and models, as well as enhancements to the final report generated by Uppex. Additionally, the first steps were taken toward developing a graphical user interface for the tool, aiming to provide a more user-friendly experience and reduce reliance on the terminal.

Chapter 3

Modelling in Imitator

To explore the capabilities of the Imitator tool in greater detail, we begin by thoroughly defining its syntax, outlining the process of building a complete model in Imitator (Section 3.1). The rest of this chapter describes how to use Imitator using four real-time systems (Section 3.4): a *Traffic Light System*, a *Coffee Machine*, a *Worker Hammer System* and a simple *ATM Machine*. For each system, parameter synthesis will be demonstrated by verifying parameter constraints to prevent deadlocks. Additionally, one of the parameters in each system will be optimized.

refer to sections 3.2 and 3.3

3.1 Specifying automata

The Imitator tool can always be divided into three main parts. In each part, the worker automaton from the worker and hammer example presented in Chapter 2 will be used to illustrate the Imitator's syntax.

- **Variable declarations**

  In the variable declaration block, illustrated in Listing 3.1, we define the essential elements of the model, including `clock`s, discrete variables, `constant`s, and `parameter`s. These elements are used in expressions that represent temporal constraints in the rest of the model.

  The `clock`s are special variables that can evolve continuously, all at the same rate. Traditional variables are discrete and have an assigned type, which can be `Int` (integer), `Boolean`, `Array`, etc., written e.g. `x : Int` for an integer variable. Only discrete variables can be shared among

Listing 3.1: Variable Declaration Exemple using Worker Automata

```
var   session, t              : clock;
      sessionTime, reactTime  : parameter;
```

different automata. Variables marked with `constant` cannot be modified. Finally, a `parameter` is an undefined variable that is inferred automatically by Imitator, which will be explained later.

- **Automata**

  In the Automata block, we define the timed automata that represent the system's behavior. It includes states, which represent different phases of the system, and transitions, which define how the system moves from one state to another. Additionally, it contains invariants, which impose constraints on the time that can be spent in a state, guards, which set conditions for a transition to occur, and assignments, which update variables or clocks after a transition.

  Before constructing the automaton, it is necessary to explicitly declare all the synchronization actions present in the model using the keyword `synclabs`, followed by the corresponding transitions.

  Each location of the automaton is initialized with `loc`, followed by its name and, if present, an invariant. The definition continues with `when`, followed by the transition condition, `sync` for the synchronization action, `do` for variable or clock updates, and `goto` to indicate the next state. If there are multiple transitions from the same location, they are listed sequentially.

  Each location can be urgent if no time passage is allowed in that location, defined by the keyword `urgent`. This means the system must leave this location immediately without allowing time to elapse. A location can be accepting if it represents a significant final state for verifying liveness properties. Accepting locations are used to define states that must be reached repeatedly in infinite executions, making them useful for checking the recurrence of desired behaviors. It is defined by the keyword `acepting`. Additionally, a location can combine both properties, being both urgent and accepting, ensuring that the system reaches this location without delay and that it is considered for acceptance conditions in liveness verification.

  It is also worth noting that the keyword `Stop` allows certain clocks to be interrupted in specific locations, while `Flow` enables the modification of the clock's evolution rate.

Listing 3.2: Construction of the Automata using Worker Automata as Example

```
(*Worker Automata*)
automaton Worker

synclabs: rest, hit, newNail, Work;

loc Rest: invariant session <= sessionTime - reactTime
    when True sync Work do {t := 0} goto Work;

loc Work: invariant session <= sessionTime && t <= reactTime
    when t >= reactTime sync newNail do {t := 0} goto Work;
    when t >= reactTime - 5 sync hit do {t := 0} goto Work;
    when session >= sessionTime sync rest do {session := 0} goto
       Rest;

end (* Worker *)
```

- **Initial state definition**

  The initial state definition section is split between the discrete initialization and the continuous initialization. This section starts with the keyword **init**, with each condition separated by **&**. The discrete initialization (optional introduced by the discrete keyword) assigns an initial value to each discrete variable, and sets the initial location for each automaton. The continuous initialization (optional introduced by the continuous keyword) defines the initial constraints over clocks and parameters (possibly also using discrete variables).

Listing 3.3: Initial state definition using Worker Automata as Exemple

```
init := loc[Worker] = Rest
      & session = 0
      & t = 0
      & sessionTime >= 0
      & reactTime >= 0;
end (* of file *)
```

3.2 Specifying queries

Finally, in terms of the syntax of the queries, which, as explained earlier, are written in a separate file, the query consists of the synthesis mode (synth or witness) and the synthesis algorithm. This condition is known as a state predicate that defines a specific state of an automaton. It typically takes the form loc[AUTOMATON] = LOCATION, where AUTOMATON is the name of the automaton and LOCATION is the name of a location within it. State predicates support logical operations like conjunction (AND), disjunction (OR), and can also involve global discrete variables **?**. The query is stored in a separate file with the `.imiprop` extension.

To check if the system is syntactically correct and in the desired format, we save the code in a file with the .imi extension (e.g., system.imi). Then, we execute the following command in the terminal: `./imitator system.imi [hammer_proprities.imiprop]`. The system.imi file serves as a model, representing various systems. Alongside, the property.imiprop file contains all the properties that we aim to check. The -option flag in the command line allows to specify the desired output file format, whether it be PNG, Uppaal, or any other preferred format. It's important to note that the parameters enclosed in square brackets are optional **?**.

3.3 Installing Imitator

The easiest and fastest way to install Imitator is by using Docker. Other alternatives can be found online at https://www.imitator.fr/. You can download the Imitator docker image with the following command: `docker pull imitator/imitator`. The developers of Imitator recommended to run the container from a bash terminal using a shared directory with the host computer. This facilitates extracting the results from using Imitator within the Docker image. The container will be automatically removed once it is closed when using the option `--rm`. Summarising, we use the following command to run Imitator over a file called `docker run --rm -it --entrypoint /usr/bin/bash -v Directory Path/Directory Name imitator/imitator`.

In the next chapter, as will be explained, the command used in the Backend is quite similar, with the interactive option (-it) removed, resulting in: `docker run --rm --entrypoint /usr/bin/bash -v Directory Path/Directory Name imitator/imitator`.

3.4 Examples

We illustrate the syntax defined above with three examples: a *Coffee Machine*, a *Worker and Hammer System*, and a *ATM Machine*. For each system a set of properties will be tested to demonstrate the capabilities of the Imitator tool.

3.4.1 Coffee Machine

First, we have a Coffee Machine system. To simplify, it is assumed that this machine does not require coins or any other mechanism to start the process; it only has 1 button to initiate the process. This button is also used to add sugar to the coffee.

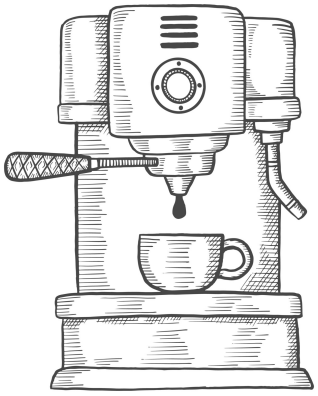


Figure 10: Simple Coffee Machine with only one button to control all operations. **?**

To translate this system into the IMITATOR syntax, it is necessary to represent events and transitions over time using concepts such as **clocks**, **variables**, **states**, and **transitions**. As explained in the previous chapter, to build our model, we need to construct three essential blocks.

In the first block, we declare the clocks and variables (discrete, rational, or parameters):

```
var  x, y       : clock;
     p1, p2, p3 : parameter;
```

The variable **p1** represents the time elapsed between two consecutive sugar requests after the process has started. In other words, it is the interval between each request made by the user to add sugar to the coffee. The variable **p2** defines the time window during which the user can press the button again to request more sugar. If the system allows this additional request, it will wait for user interaction within this

interval before proceeding with the process. Finally, the variable **p3** corresponds to the time required to produce and serve the coffee. This period includes all steps from the beginning of the preparation until the moment the beverage is served to the user.

The next step is the construction of the automaton. In this case, a single automaton will be developed to represent the operation of the coffee machine.

```
(*Coffe Machine*)
synclabs: press, cup, coffee, sleep;

loc idle: invariant True
    when True sync press do {x := 0, y := 0} goto add_sugar;

loc add_sugar: invariant y <= p2
    when x >= p1 sync press do {x := 0} goto add_sugar;
    when y = p2 sync cup do {} goto preparing_coffee;

loc preparing_coffee: invariant y <= p3
    when y = p3 sync coffee do {x := 0} goto cdone;

accepting loc cdone: invariant x <= 10
    when True sync press do {x := 0, y := 0} goto add_sugar;
    when x = 10 sync sleep goto idle;

end
```

The coffee machine's automaton consists of four states. The first state is **Standby(idle)**, where the system waits indefinitely for the user to take action to start making coffee. The next state is **add_sugar**, where the user can choose to add sugar. There is a time limit (p_2) for adding sugar, and a restriction (p_1) on how often sugar can be added in quick succession. These restrictions are enforced using invariants, in the state add_sugar, with the invariant $y \leq p_2$ and in the preparing_coffee state with the invariant $y \leq p_1$.The synchronization here is based on the *press* event, which triggers the action of adding sugar and starts the process. Next is the coffee preparation state**(preparing_coffee)**, which takes p_3 time units to complete. The synchronization with the coffee event triggers the completion of the coffee preparation. Finally, in the last state(**cdone**), if the user presses the button again within 10 time units, the system

goes back to add_sugar and repeats the process. This synchronization is triggered by the *press* event. If the user does not press the button within the allowed time, the system synchronizes with the *sleep* event, returning to the Standby state.

It is important to note that the last state, called the **accepting loc**, is different from the others because it represents a 'final' or 'accepting' state. In simple terms, once the system reaches this state, the coffee is prepared and ready to be served. This distinction is useful during the verification process, as it allows us to check whether the automaton can reach a successful end state instead of getting stuck in infinite loops within intermediate states **?**.

The model is summarized in Transition Table 2.

State	Invariant	Guard	Synchronized Event	Action	Next State
idle	Always (True)	Always (True)	press	x := 0, y := 0	add_sugar
add_sugar	$y \leq p_2$	$x \geq p_1$	press	x := 0	add_sugar
add_sugar	$y \leq p_2$	$y = p_2$	cup	None	preparing_coffee
preparing_coffee	$y \leq p_3$	$y = p_3$	coffee	x := 0	cdone
cdone	$x \leq 10$	Always (True)	press	x := 0, y := 0	add_sugar
cdone	$x \leq 10$	$x = 10$	sleep	None	idle

Table 2: Transition table of the coffee machine automaton.

To complete the model, it is necessary to declare the initial states as well as the possible constraints on the clocks and declared variables:

```
init :=
    (* Initial location *)
    & loc[machine] = idle
    (* Initial clock constraints *)
    & x = 0
    & y = 0
    (* Parameter constraints *)
    & p1 >= 0
    & p2 >= 0
    & p3 >= 0
;
```

The initial location, as explained earlier, is set to the *idle* state, meaning Standby mode. Additionally, the clocks are initialized to 0, and the parameters must be positive.

To verify whether our system complies with the syntax, we can, as explained in the previous chapter, run the model in the command line with the option to translate the system into a PNG image using the following command: `./imitator mod.imi -imi2PNG`

If this command fails, it means that our model has syntax errors that need to be fixed.

Considering that we saved our model in the **mod.imi** file, we obtain the following image as output in Figure 12, which visually represents our code.

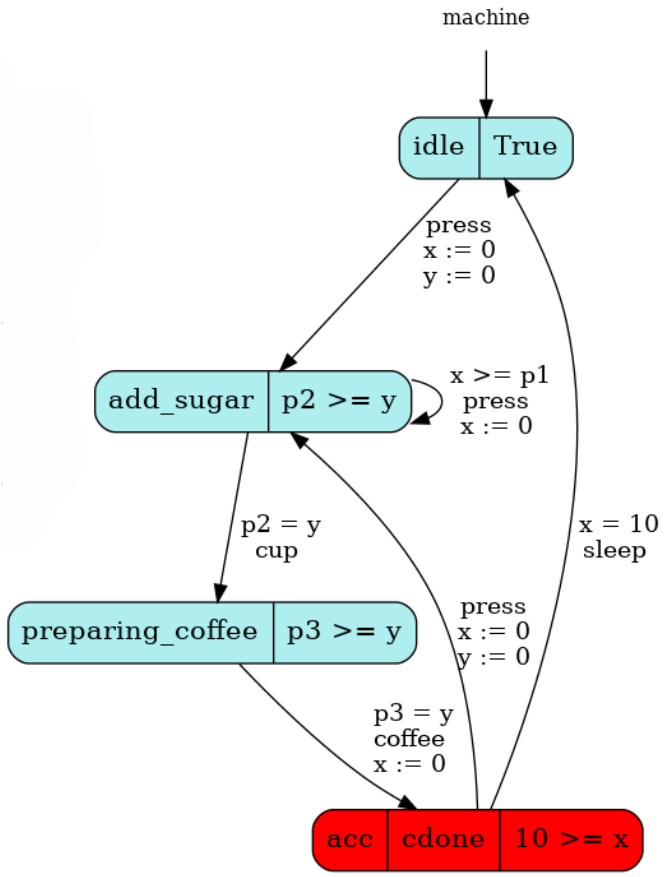


Figure 11: Coffee Machine Automata

System Analysis

Once the model is built, we can proceed to property verification. We start with parameter synthesis to ensure that deadlocks are avoided. To achieve this, we need to create a separate `.imiprop` file, distinct from the .imi file containing our model, as explained in the previous chapter.

Inside the file, we define the property to be verified. In this case, related to deadlock. The syntax is as follows : `property := #synth DeadlockFree;`

The result is stored in a `.res` file with the same name as the file where the model is saved. This file contains general information about the model as well as the verification of the property. However, the

most important part we extract is the following:

```
 p1 >= 0
& p2 >= 0
& p3 >= p2
OR
  p2 > p3
& p3 >= 0
& p1 = 0
```

Any values chosen for the parameters, as long as they satisfy the given constraint, ensure that the system does not reach a deadlock. We can also determine the maximum or minimum parameter value that ensures a given property is satisfied. For example, we may want to find the shortest possible time between two consecutive sugar requests($p1$) while still allowing the coffee to be served to the user (i.e., reaching the location `cdone`).

To achieve this, we can use the following property: `property := #synth EFpmin(loc[machine] = cdone, p1);` With this property, we obtain the following result:

```
p3 >= p2
& p2 >= 0
& p1 = 0
```

The parameter $p1$ can be 0, meaning that the request for sugar can be made instantly. However, this is only possible as long as the time required to prepare a coffee ($p1$) is greater than the time interval in which sugar can be requested ($p2$). Additionally, $p2$ must be strictly positive to ensure a valid sequence of events.

A synthesis of both verified properties can be found in the table 6

3.4.2 Worker and Hammer System

Now we have the "Worker and Hammer" system, as discussed in the State of the Art chapter and implemented in UPPAAL. This system represents the interaction between a worker and a hammer to perform a repetitive task, such as hammering nails.

As in the other examples, we start by defining the variables and clocks of the automata.

Property	Meaning	Result
`property := synth DeadlockFree;`	Ensures the system does not reach a deadlock state.	$p_1 \geq 0 \wedge p_2 \geq 0 \wedge p_3 \geq p_2$ **or** $p_2 > p_3 \wedge p_3 \geq 0 \wedge p_1 = 0$
`property := #synth EFpmin(loc[machine] = cdone, p1);`	Finds the minimum value of p_1 to reach location `cdone`.	$p_3 \geq p_2$ $p_2 \geq 0$ $p_1 = 0$

Table 3: Synthesis of properties and corresponding results.

```
var
session, t
:clock;

sessionTime,
reactTime,
totalNails
: parameter;
nails
: int;
```

The **t** and **session** timers are used to measure elapsed time for different purposes. The t timer tracks short-duration actions, such as hammering a nail or placing a new one, while the session timer records the total time a worker has spent working in a session, determining when the worker need to take a break. The variables **sessionTime**, **reactTime**, and **totalNails** are defined as parameters, unlike in the UPPAAL example, where they had fixed values. **sessionTime** represents the maximum time a worker can work before stopping, **reactTime** is the maximum time allowed for the worker to hit or place a new nail, and **totalNails** indicates the total number of new nails available. Finally, the variable **Nails** is defined as a discrete type, specifically an integer, and serves as an accumulator for the number of completed nails. Its value is initialized to 0 in the final section of the code. With all variables declared, the next step is the definition of the automata.

```
(*Worker Automata*)
automaton Worker

synclabs: rest, hit, newNail, Work;
```

```
    loc Rest: invariant session <= sessionTime - reactTime
        when True sync Work do {t := 0} goto Work;

    loc Work: invariant session <= sessionTime && t <= reactTime
        when t >= reactTime sync newNail do {t := 0} goto Work;
        when t >= reactTime - 5 sync hit do {t := 0} goto Work;
        when session >= sessionTime sync rest do {session := 0} goto
           Rest;

    end

    (*Hammer Automata*)
    automaton Hammer

    synclabs: hit, newNail;

    loc NailUp: invariant True
        when True sync hit goto NailHalf;

    loc NailHalf: invariant True
        when True sync hit do {nails := nails +1} goto NailDone;

    loc NailDone: invariant True
        when nails <= totalNails sync newNail goto NailUp;

    end
```

The system consists of two automata: **Worker** and **Hammer**. The **Worker** represents a worker who alternates between the states Rest and Work. In the Rest state, the worker waits for the start of work through the Work synchronization, resetting the clock t in the process. After this synchronization, in the Work state, the worker can perform two actions: hitting the nail (hit), if t ≥ reactTime - 5, or placing a new nail (newNail!), if t ≥ reactTime, always resetting t after each action. The worker can remain in the Work state as long as session < sessionTime. Once this limit is exceeded, the worker returns to the Rest state through the transition rest!, which also resets the session clock.

The **Hammer** automaton represents the interaction of the hammer with the nails, alternating between

the states NailUp (new nail), NailHalf (partially hammered nail), and NailDone (completely hammered nail). Initially, it is in the NailUp state, where upon receiving the synchronization Hit, it transitions to NailHalf. Upon receiving a second synchronization Hit, it transitions to NailDone, incrementing the variable nails if countNails is activated. When the nail is completely hammered, the system can insert a new nail through the synchronization newNail?, returning to the NailUp state, provided that infiniteNails is true or nails < totalNails.

The synchronization between the two automata occurs through the channels Hit and newNail presented in the Worker and Hammer automata, ensuring that the sequence of events is consistent and respects the temporal constraints imposed by the clocks t and session.

We can better visualize the behavior of the Worker automaton in Table 4 and of the Hammer automaton in Table 5.

State	Invariant	Guard	Synchronized Event	Action	Next State
`Rest`	Always (`True`)	session $\leq$ sessionTime - reactTime	`work!`	`session := 0`	`Work`
`Work`	session $\leq$ sessionTime	t $\leq$ reactTime	`newNail!`	`t := 0`	`Work`
`Work`	session $\leq$ sessionTime	t $\geq$ reactTime - 5	`hit!`	`t := 0`	`Work`
`Work`	session $\geq$ sessionTime	None	`rest!`	`session := 0`	`Rest`

Table 4: Worker Transistion Table

State	Invariant	Guard	Synchronized Event	Action	Next State
`NailUp`	Always (`True`)	nails $\leq$ totalNails or infiniteNails	`newNail?`	None	`NailHalf`
`NailHalf`	Always (`True`)	None	`hit?`	`nails++`	`NailDone`
`NailDone`	Always (`True`)	None	`newNail?`	None	`NailUp`

Table 5: Hammer Transistion Table

Finally, we just need to declare the initial states of the automata, which are Rest and NailUp for the Worker and Hammer, respectively. Additionally, we initialize the clocks and the nails variable to 0, and set the condition that the parameters must be greater than 0.

```
init :=

& loc[Worker] = Rest
& loc[Hammer] = NailUp
```

```
    & session = 0
    & t = 0

    & nails = 0
    & sessionTime >= 0
    & reactTime >= 0
    ;
    end
```

As in the previous examples, we convert our model code into a visual representation.

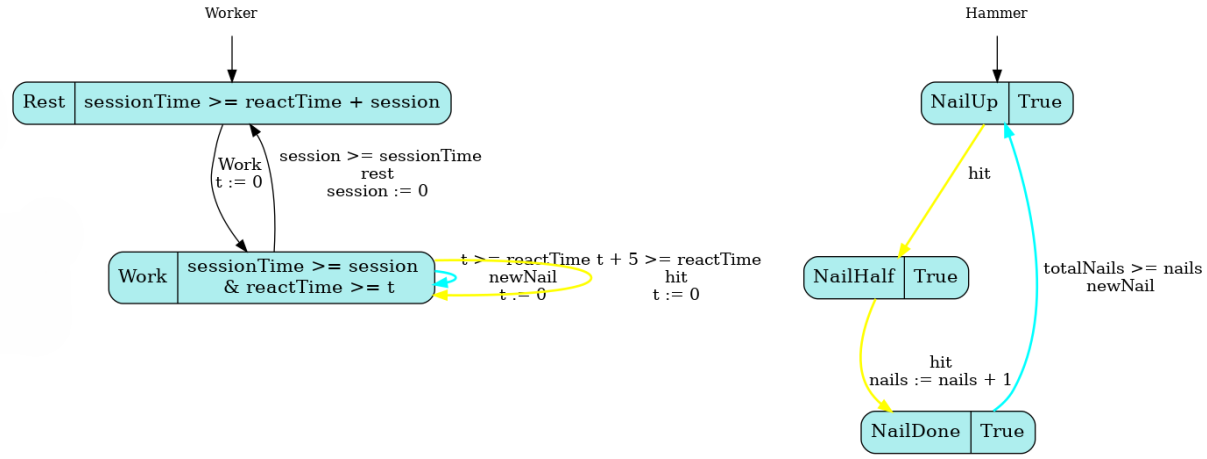


Figure 12: Worker and Hammer Automata

System Analysis

As before, we start by determining the parameter values needed to avoid a deadlock. In other words, we analyze how the worker's working time, the time required to hammer or place a new nail, and the number of nails must be set to prevent this situation. The property, as previously mentioned, is stored in a separate `.imiprop` file and has the following syntax: `property := synth DeadlockFree;` Analyzing the `.res` file, the parametric interval that guarantees the absence of deadlock is as follows:

```
 0 > reactTime
& sessionTime >= 0
OR
  reactTime >= sessionTime
& sessionTime >= 0
```

We can also determine the parametric interval that guarantees the completion of the worker's action—that is, ensuring that the location NailDone in the Hammer automaton is reachable. For this, we use a simple Reachability property with the following syntax: `property := #synth EF(loc[Hammer] = NailDone);`

The parametric interval is of the form:

```
 reactTime >= 0
& sessionTime >= reactTime
```

To ensure that the completion of the action is guaranteed, reactTime must be a positive value, and additionally, it must be ensured that sessionTime is always greater than or equal to reactTime. Table 6 summarizes the verified properties and their corresponding synthesis results. The first property ensures that the system avoids deadlock by constraining the parameters `reactTime` and `sessionTime`. The second property determines the conditions under which the system can reach the target location `cdone`, ensuring progress towards the completion of the task. Together, these results confirm that the model behaves correctly within the specified parameter ranges.

Property	**Meaning**	**Result**
`property := synth DeadlockFree;`	Ensures the system does not reach a deadlock state.	$reactTime > 0$ $\wedge\ sessionTime \geq 0$ **or** $sessionTime \geq reactTime$ $\wedge\ sessionTime \geq 0$
`property := #synth EF(loc[Hammer] = NailDone);`	Finds the minimum value of p_1 to reach location `cdone`.	$reactTime \geq 0$ $\wedge sessionTime \geq reactTime$

Table 6: Synthesis of properties and corresponding results.

3.4.3 ATM Machine

The final example is of a simple ATM with only two functions: balance inquiry and cash withdrawal, considering time constraints for each phase of the operation. The example was also taken from the official Imitator website **?**. We begin by declaring the system's variables and clocks:

```
var
x,y,z : clock;
```

```
p1 : parameter;
p2 : parameter;
p3 : parameter;
```

Clock x controls the duration of each operation and is used to limit the time allowed for the selected operation. For example, if x = 15 and the chosen operation is withdraw, the user has 15 time units to complete it. Clock y is responsible for tracking the login time, i.e., the time the user takes to provide access credentials. Finally, clock z monitors the total session time (login + operations). Regarding the variables, they are all defined as parameters: p1 is the maximum allowed session time, p2 defines the maximum time for login, and p3 sets the maximum time allowed per operation, whether it is a balance inquiry or a withdrawal.

```
automaton pta

synclabs: ;

loc Idle: invariant True
    when True do {x := 0 , y := 0 , z := 0} goto Start;

loc Start: invariant z <= p1
    when z = p1 goto Idle;
    when z <= p1 do { y := 0 } goto Login;

loc Login: invariant y <= p2 && z <= p1
    when z = p1 goto Idle;
    when True do {x := 0} goto Withdrawals;
    when y = p2 goto Start;
    when True do {x := 0} goto Check;

loc Withdrawals: invariant x <= p3
    when True goto Login;

loc Check: invariant x <= p3
    when True goto Login;
```

```
end
```

The automaton is composed of five locations: Idle, Start, Login, Withdrawals, and Check. The initial state is Idle, which represents the ATM being in a standby state, waiting for the start of an operation. When this occurs, all clocks are reset to 0 and the automaton transitions to the Start location. This location marks the beginning of the user interaction. The invariant z <= p1 limits the session duration, with p1 representing the maximum session time; if this limit is exceeded, the system returns to the Idle state (standby). Otherwise, it proceeds to the Login state, where clock y is reset. This clock tracks the time the user takes to log in. In this location, the machine waits for the user to enter their credentials, with the constraint that the login duration must not exceed either of the specified time bounds, as indicated by the invariant `y <= p2 && z <= p1`. If both conditions are satisfied, the system moves to the Withdrawals location if that was the selected function, or to Check otherwise. In both cases, clock x is reset. If the selected operation is Withdrawal, the system transitions to that location, giving the user a limited time to complete the withdrawal, which cannot exceed p3, as shown by the invariant `x <= p3`. If the option is Check, it moves to the corresponding location, where the same time restriction applies, also enforced by the invariant `x <= p3`. In both cases, after completing the action, the system returns to the Login location, allowing the user to repeat or change the selected operation. As with the previous examples, we can construct a transition table for the system, as shown in Figure 7.

State	Invariant	Guard	Synchronized Event	Action	Next State
Idle	Always (True)	Always (True)	start	x := 0, y := 0, z := 0	Start
Start	$z \leq p_1$	$z = p_1$	timeout	None	Idle
Start	$z \leq p_1$	$z \leq p_1$	proceed	y := 0	Login
Login	$y \leq p_2 \wedge z \leq p_1$	$z = p_1$	timeout	None	Idle
Login	$y \leq p_2 \wedge z \leq p_1$	$y = p_2$	restart	None	Start
Login	$y \leq p_2 \wedge z \leq p_1$	Always (True)	withdraw	x := 0	Withdrawals
Login	$y \leq p_2 \wedge z \leq p_1$	Always (True)	check	x := 0	Check
Withdrawals	$x \leq p_3$	Always (True)	done	None	Login
Check	$x \leq p_3$	Always (True)	done	None	Login

Table 7: Transition table of the ATM automaton.

Finally, we define the parameter constraints and the initial location of the automaton.

```
init :=

& loc[pta] = Idle,
```

```
& x = 0
& y = 0
& z = 0

& p1 >= 0
& p2 >= 0
& p3 >= 0
;

end
```

As mentioned, the system starts in the Idle location, which represents the standby state. Additionally, the only constraint imposed on the parameters is that they must be greater than 0. As with the previous examples, we can convert the model into a figure, as shown in Figure 13.

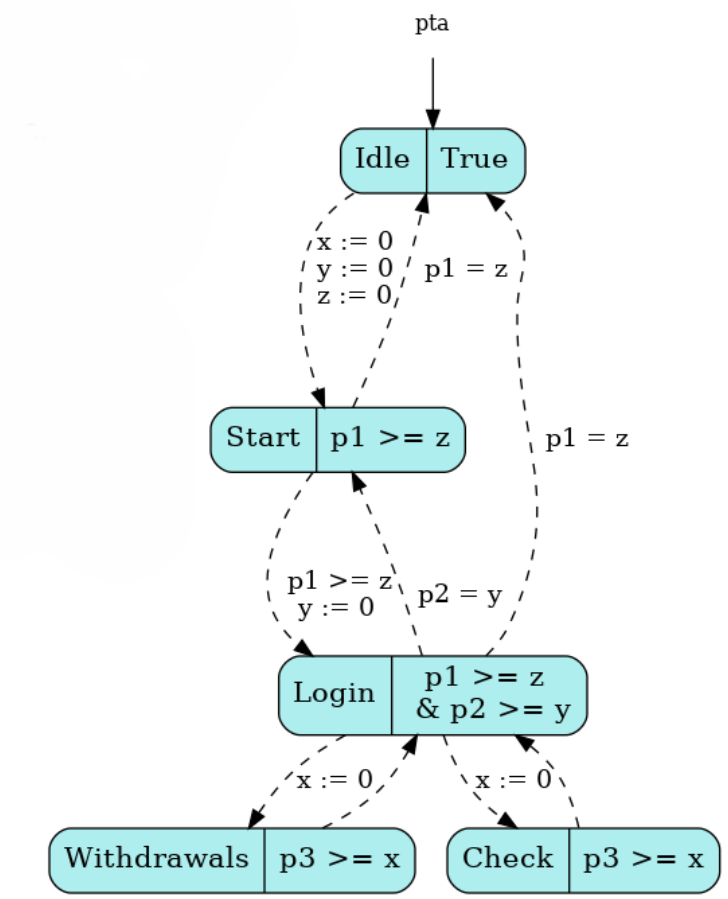


Figure 13: ATM Automata

System Analysis

As in the previous examples, the first property to be verified is deadlock-freedom, ensuring that the system does not reach any state from which execution cannot proceed. The syntax for this property, as previously mentioned, is: `property := synth DeadlockFree`. The parametric interval that guarantees the verification of the property is as follows:

```
p3 >= 0
& p2 >= 0
& p1 >= 0
```

It tell that whatever values the parameters may take, it must be ensured that they are always positive in order to guarantee that "regardless of the situation, the system always has an action to take."

The next property to be verified is a liveness property that searches for at least one cycle in the automaton that goes through the Withdrawals state and can be executed infinitely (i.e., without deadlock). In practical terms, this ensures that it is always possible to perform withdrawals. For this purpose, we use the CycleThrough algorithm, which was briefly mentioned in the previous chapter. The corresponding syntax is: `property := synth CycleThrough(loc[pta] = Withdrawals);`. The returned parametric interval guarantees that the cycle exists for any non-negative values of parameters p1, p2, and p3.

```
p3 >= 0
& p2 >= 0
& p1 >= 0
```

Additionally, Imitator provides extra information such as the depth at which the cycle was found (i.e., the number of transitions required to complete it), as well as the number of states generated and effectively explored during the analysis.

A summary of the analyzed properties and their corresponding results can be found in Table 8.

Property	**Meaning**	**Result**
`property := #synth DeadlockFree;`	Ensures the system does not reach a deadlock state.	$p_1 > 0$ $\wedge\, p_2 \geq 0$ $\wedge\, p_3 \geq 0$
`property := #synth CycleThrough(loc[pta] = Withdrawals);`	Find a cycle in which the Withdrawals location is visited. `cdone`.	$p_1 > 0$ $\wedge\, p_2 \geq 0$ $\wedge\, p_3 \geq 0$

Table 8: Synthesis of properties and corresponding results.

Chapter 4

IMITATOR Backend for Uppex

At this stage of the work, and taking into account the capabilities demonstrated by the tools in the previous chapters, the primary objective is the integration of the Imitator with the UPPEX system. As mentioned in Chapter 2, UPPEX allows extending an UPPAAL model through the use of annotated blocks and XML blocks. Previously, UPPEX only used UPPAAL as the model checker in the backend. However, with the integration of Imitator, it is now possible to have two backends: UPPAAL and Imitator. This integration opens up new possibilities for model checking, allowing users to choose between UPPAAL and Imitator based on their specific needs. Additionally, the flexibility to work with annotated blocks alone when using Imitator simplifies the modeling process while maintaining the robustness of the system. This means users can more easily extend their models and run checks with two powerful tools, optimizing performance and accuracy in different use cases. Additionally, this integration brought improvements to the Excel frontend, allowing it to handle the new Imitator syntax and support a new type of constraints and analyses, now enabling the use of parameters.

This chapter starts by presenting a simple example (the Coffee Machine) as a motivation for the new tool, before detailing the changes made to Uppex to accommodate the new backend and the resulting modifications. Finally, the previously referenced Worker-Hammer example will be used to demonstrate the functionality and results of the new version of Uppex.

4.1 Quick Start Uppex with Imitator

To download the new version of Uppex, the following steps must be followed:

- **Access the official repository and select the latest version of the JAR file.**

  Go to the official Uppex repository using the following link and select the recent version.

- **Ensure that Docker Desktop is running before executing the JAR file.**

- **Ensure that java is intalled**
- **Ensure that the excel file and xml/imi is in the same directory as the jar file**

These instructions are described in more detail in the official documentation of the tool, which has been upgraded to include the changes implemented in the new version and can be accessed through the following link: https://github.com/alexandre04032000/uppex-imitator/blob/Uppex-Imitator/README.md

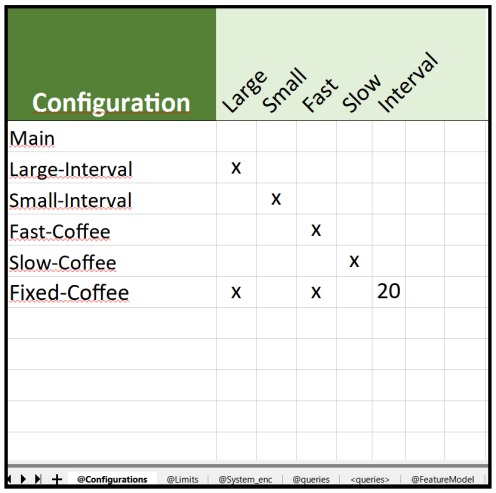
use just the landing github page

4.2 Using Uppex with Imitator by example

The model used to build the different configurations is the same as the one used in Chapter 3. It consists of three variables, all of which are parameters: p_1, the time between two consecutive sugar requests; p_2, the time interval during which the user can request sugar in the coffee; and finally p_3, the time required to complete the entire process of making and serving the coffee.

With this, we can construct four features: **Large, Small, Fast, Slow, Interval**. The first two refer to the time window during which the user can request sugar. Specifically, the Small feature represents a short interval both for making a sugar request and for the time between consecutive sugar requests. In contrast, the Large feature corresponds to longer intervals for both. The last two features are related to parameter p_3, representing the coffee preparation time: the Fast feature indicates a quicker preparation process, while the Slow feature indicates a longer one. It is important to note that all these features have predefined values. However, if different values are needed, they can be easily specified using the Interval feature.

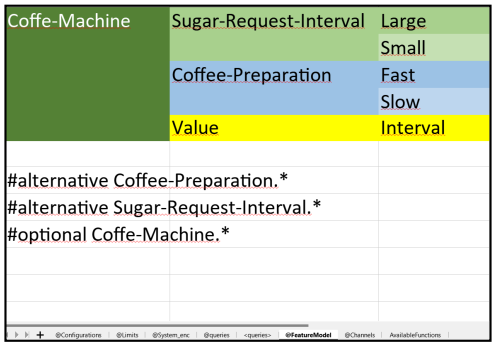
Using these features, we built six configurations, as shown in Figure 14: **Main, Large-Interval, Small-Interval, Fast-Coffee, Slow-Coffee, and Fixed-Coffee**. It is worth nothing that feature **Main** represents the original model without any modifications, with all three variables treated as parameters. In contrast, **Fixed-Coffee** represents a non-parametric model in which the interval for requesting sugar in the coffee is large, requiring the overall process to be fast—this is represented by the selection of the Slow-Coffee and Fixed-Coffee features.



Configuration	Large	Small	Fast	Slow	Interval
Main					
Large-Interval	x				
Small-Interval		x			
Fast-Coffee			x		
Slow-Coffee				x	
Fixed-Coffee	x		x		20

Figure 14: Configurations Sheet Uppex

It is necessary to construct the feature model in the corresponding sheet, as shown in Figure 15, in order to constrain the feature selections and prevent combinations that do not make sense for the system in question.



Coffe-Machine	Sugar-Request-Interval	Large
		Small
	Coffee-Preparation	Fast
		Slow
	Value	Interval
#alternative Coffee-Preparation.*		
#alternative Sugar-Request-Interval.*		
#optional Coffe-Machine.*		

Figure 15: Feature Model Sheet Uppex

We also built the Limits and System_enc sheets, which represent the constraints applied to each parameter, as shown in Figures 16 and 17, respectively.

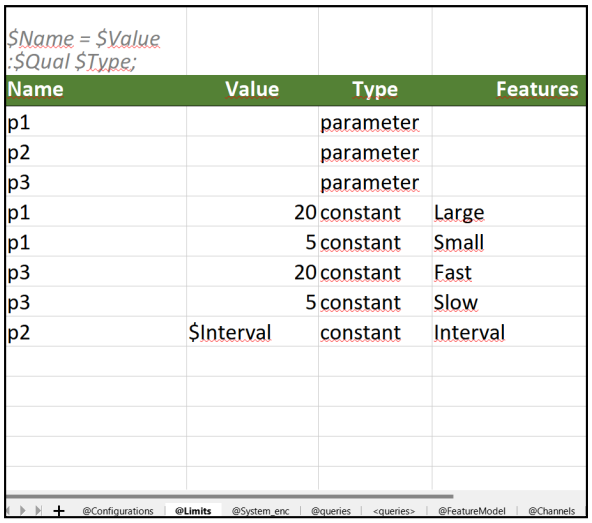


Figure 16: Limits Sheet

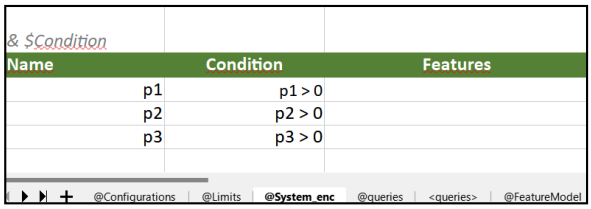


Figure 17: Parameters Constrain Sheet

Finally, we configured the Queries sheet (Figure 18). For the sake of simplicity, all configurations are set to verify the Deadlock property. Additionally, although not shown in the figure, the synthesis mode (as explained in the previous chapter) is set to synth.

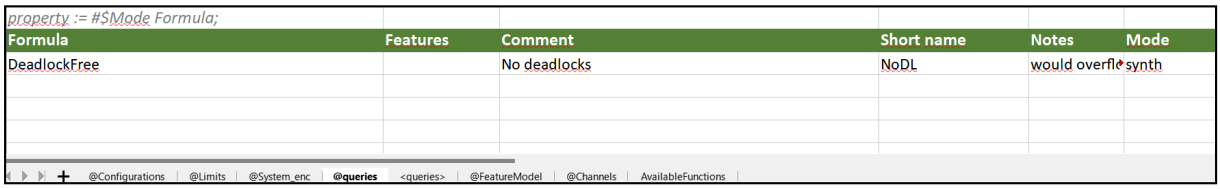


Figure 18: Queries Sheet Uppex

With the parameterized model, we can run the JAR file, and the operating mode is exactly the same as the old version. Within the same directory, we need to have the JAR file and we can name it uppex.jar, the Excel file, and the Imitator file, with the last two files needing to have the same name.

In the terminal, we use the command: `java -jar uppex.jar -runAll Imitator_Model.imi`, thus compiling all configurations. In each configuration, the deadlock property will be verified, and the results will be presented in an HTML file grouped by property and by configuration. For our example, the results can be found in Appendix C.

use ref

As expected, for each of the configurations, in order for the property to be satisfied, p_1, p_2, and p_3 must fall within a specific range determined by the algorithm used. It is worth noting that the fixed_coffee configuration does not have any associated interval since, as previously explained, all its variables are assigned fixed values. This turns it into a parametric model that, for the chosen values, verifies the condition. In this case, it reduces to the model checking of timed automata.

4.3 Implementation details

As previously mentioned, we will now provide a detailed explanation of the changes made to the old version of the tool to accommodate the new functionalities.

4.3.1 Imitator Parser

To properly process and store the data extracted from Uppaal (.xml), Imitator (.imi), and Excel files, two specific data structures need to be created. These structures are designed to organize, structure, and simplify access to the collected information, ensuring efficient storage and data handling.

For reading Uppaal models, the solution to the problem is shown in the figure above in red. There is a base type called **Block** (like a code block), declared as a trait. Inside each block, there can be a tag (**NameBl**) that identifies the relevant part of the model, while the rest is called **content**.For this, two classes are created that extend this type, meaning they inherit its basic properties and behaviors, allowing them to reuse and specialize its functionalities. It is important to note that NameBl is written as an abstract class, meaning it cannot be instantiated directly and serves as a model for other classes that extend it. This means that NamedBl can define common methods and attributes but leaves the implementation of certain details to its subclasses. These are called **XmlElm**, if the tag is of the XML type, or **AnnotationBl**, when it is an annotation. Each of these subclasses specializes the behavior defined in the abstract class NamedBl, adapting to the specific type of block they represent. This ensures a clear distinction between XML elements and annotations, allowing proper handling for each case within the model. Along with the data structure, we also have a variety of helper functions aimed at displaying the model.

In the constructors of the classes `NamedBl`, `XmElm`, and `AnnotationBl`, the following parameters are used:

- **`name`**:

  Represents the name of the annotation.

- **`oldLines`**:

  Contains the original content of the annotation, exactly as it appears in the source file.

- **`newLines`**:

  Stores the updated content of the annotation, rewritten with the new values of the selected product. These values are formatted according to the structure specified in the first cell of the Excel file that

matches both the name and type of the annotation.

To demonstrate how it works, we can consider the listing 4.1, as an example. In this case, there is only one block with an annotation, specifically an annotation block of type **(Name)**, where the name is "Limits", corresponding to the **name** parameter in the constructor. The content of this block will be stored exactly as it is in the `oldLines` parameter.

To populate `newLines`, we need to refer to the Uppex Excel file. The Excel file contains a sheet with the same name as the annotation block, as shown in Figure 19. Additionally, in the first cell of the sheet, we can find the structure that specifies how to construct each annotation block.

For instance, if we consider the product in question to be "lazy," then the annotation block is rewritten according to the structure defined in the Excel file. This rewritten block is displayed in listing 4.2. The Excel part will be detailed in the next chapter.

Listing 4.1: Example of Imitator syntax

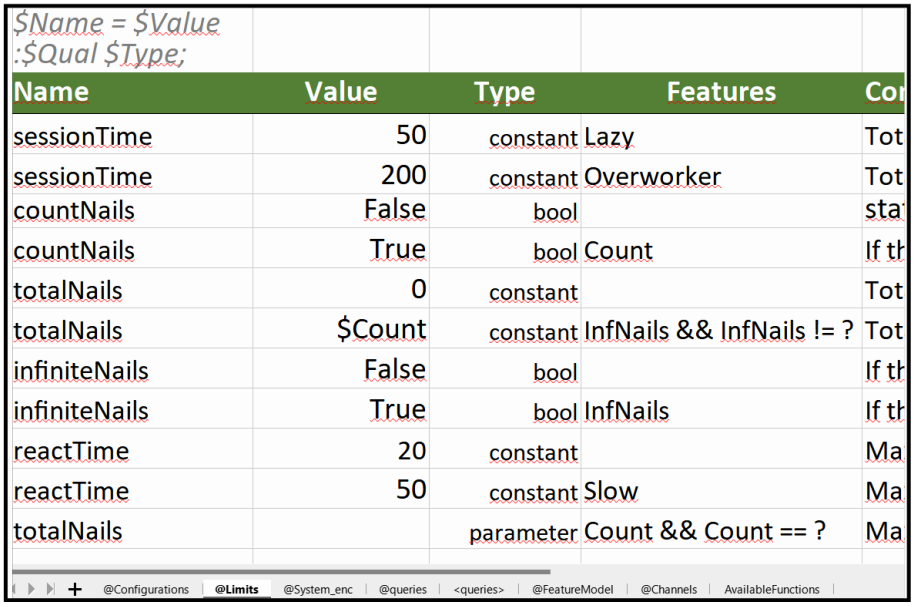
```
var
    session, t : clock;
    nails : discrete;
    b = True : bool;


(* @Limits *)
    sessionTime = 100 : constant;
    countNails = True : bool;
    infiniteNails = False : bool;
    reactTime = 20 : constant;
    totalNails : parameter;
```

Content Block

Stored as Content Block

Annotation Block

Stored as Annotation Block named Limits



$Name = $Value
:$Qual $Type;

Name	Value	Type	Features	Co
sessionTime	50	constant	Lazy	Tot
sessionTime	200	constant	Overworker	Tot
countNails	False	bool		sta
countNails	True	bool	Count	If th
totalNails	0	constant		Tot
totalNails	$Count	constant	InfNails && InfNails != ?	Tot
infiniteNails	False	bool		If th
infiniteNails	True	bool	InfNails	If th
reactTime	20	constant		Ma
reactTime	50	constant	Slow	Ma
totalNails		parameter	Count && Count == ?	Ma

Figure 19: Limits Sheet from Uppex

Listing 4.2: Annotation Block rewritten with new values

```
(* @Limits *)
    countNails = False : bool;
    sessionTime = 50 : constant;
    totalNails = 0 : constant;
    infiniteNails = False : bool;
    reactTime = 20 : constant;
```

To better understand the Data structure, we built a class diagram, as shown in Figure 20.

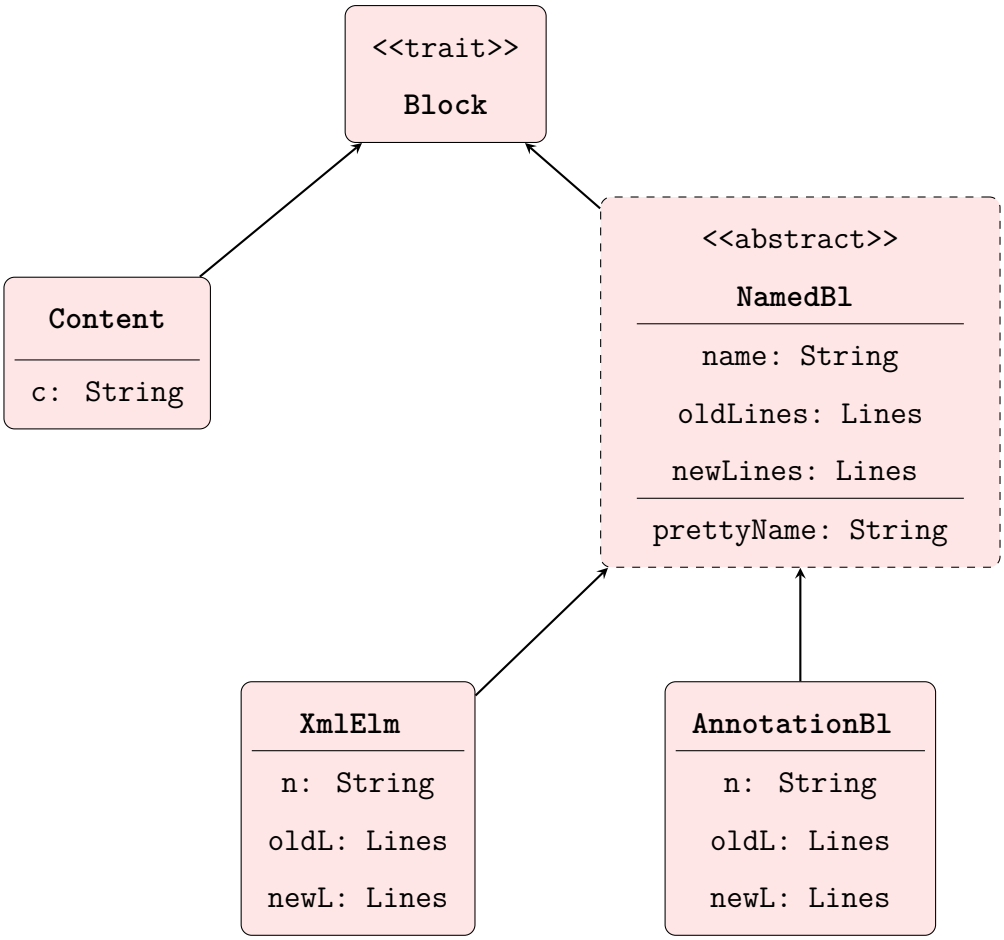


Figure 20: Uppaal Parser Data Structure

The data structure for Imitator will be very similar, allowing us to reuse the Uppaal structure. However, we can omit the XmlElm blocks, as they are no longer applicable, as shown in Figure 21.

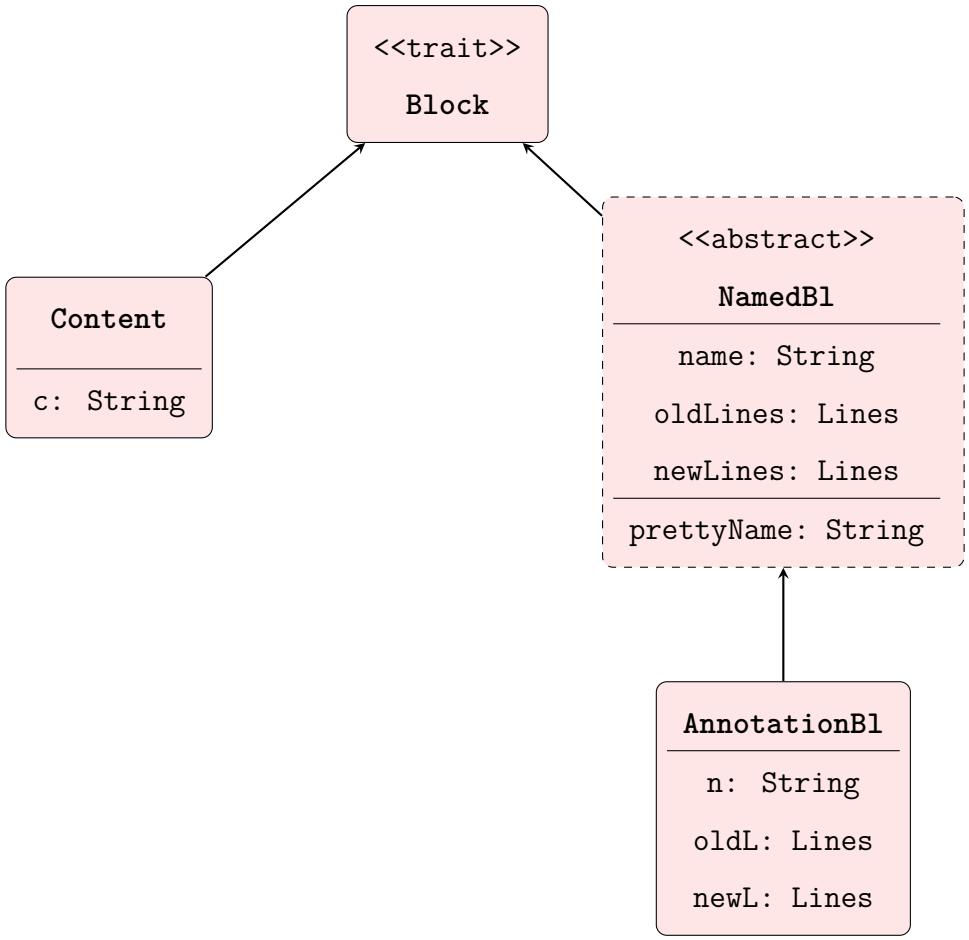


Figure 21: Imitator Parser Data Structure

To expand the tool to support Imitator models while reusing code without adding unnecessary complexity(such as code duplication and the need to assign different names to identical code blocks), a trait with several abstract methods has been created. The functions responsible for rewriting code blocks that were previously defined only for UPPAAL will be implemented in the Imitator or Uppaal classes, which extend the trait. This will allow code reuse while maintaining the flexibility and modularity of the system. This structure will also be reused, although, in the case of Imitator, the XmlElm block will not be used, as the tags inside < > will not be employed. This structure will now be part of a new singleton called GenModel and it is imported during the parsing of either the Uppaal or Imitator file, with the trait class also being declared in the same file and having the same name.

With the data structure set up, we need to build a parser to read the Imitator file, classifying each part according to the tags encountered. In parsing, code is taken from the preprocessor, broken into smaller pieces and analyzed so other software can understand it. The parser does this by building a data structure out of the pieces of input **?**.

Note that in the new version of Uppex, the notation that identifies the beginning of an annotation block

for Imitator is in the form "**(*Name of the Annotation Block*)**".

Table 9: Syntactic rules of the grammar

Rule	::=	Definition
annotationBlock	::=	"(*" "@" anName "*)" [newLine body]
lineBlock	::=	line
body	::=	repsep(neLine, newLine)

Table 10: Basic tokens of the grammar

Token	::=	Definition
newLine	::=	\n
line	::=	rep(word)
neLine	::=	rep1(word)
word	::=	[^\n]+
anName	::=	[^*]+

The functioning of this parser is quite simple. We start by reading the file as a string and passing the result to the parse function, which in turn calls imitator, the main parser that splits the code into blocks. The syntactic rules used for this parsing process are shown in Table 9, while the basic tokens of the grammar are presented in Table 10. The parser uses repsep(imitatorElem, newLine) to divide the content into lines or blocks. imitatorElem works as a block recognizer: if it finds (*@name*), it identifies it as an annotation(Annotation Block) and stores its name and content; otherwise, it considers the line as normal content (Content).

4.3.2 Better feature expressions in the Excel Reader

Regarding the Excel configuration, its functionality has remained largely the same, with changes limited to the Feature Model.

In the previous version, each annotation table mapped variables to patterns and included a special Features column containing boolean expressions over feature names. Entries were included only if the

expression evaluated to true under the selected configuration. If multiple entries shared the same key, the last one would override the previous ones. This behavior is preserved in the new model. However, with the enhanced capabilities of the new backend, the type of supported boolean expressions had to be extended. In addition to checking for active configurations, it is now possible to compare features with strings and doubles, allowing the definition of richer and more expressive configurations, as illustrated by some examples in Table 11.

Expression Type	**Old Version**	**New Version**
Simple Boolean Expressions	`Lazy && !Overworker`	`Lazy && !Overworker`
Comparison with Strings	Not Supported	`Count == ?`
Comparison with Numbers	Not Supported	`Slow > 0.5`
Combination of Types	Not Supported	`(Count == ?) && (Slow < 10)`

Table 11: Evolution of Boolean Expression Support in the `Features` Field

The first step was to update the syntactic structure of the logical expressions that the parser can analyze and evaluate, referred to as FeatExpr. This structure is represented by an enum, which is a data type that represents a fixed and limited set of possible values. In Scala, enums can be used similarly to sealed traits with case classes, allowing for the clear definition of immutable and extensible data hierarchies. In this new syntactic structure, the Comp value was introduced, as shown in Figure 22.

```
enum FeatExpr:
  case Feature(id: String)
  case And(f1: FeatExpr, f2: FeatExpr)
  case Or(f1: FeatExpr, f2: FeatExpr)
  case Imply(f1: FeatExpr, f2: FeatExpr)
  case Not(f: FeatExpr)
  case Comp(v1: FeatVal, v2: FeatVal, op: String)
  case True
```

Figure 22: Updated FeatExpr Enum

The Comp value takes three arguments: v1 and v2 of type FeatVal, and a third one called op, which indicates the type of operator that we want to use. FeatVal is a new data structure (as shown in Figure 23) created to represent the possible values that the elements on each side of the comparison can take, which are:

- `Feature(id: String)`: a reference to a feature, that is, a variable whose value will be looked up in the product.
- `Value(n: Double)`: a numeric literal value.
- `Opt(o: String)`: an option, such as a list of categories, states, or labels.

```
enum FeatVal:
  case Feature(id:String)
  case Value(n:Double)
  case Opt(o:String)
```

Figure 23: FeatVal Enum

It was also necessary to update the function responsible for evaluating a logical expression of type FeatExpr (as shown in Figure 24), based on a set of features provided in FProd, which represents a product configuration (e.g., a Map[String, Any] containing the names and values of the features).

```
def eval(fe: FeatExpr)(using prod: FProd): Boolean = fe match
  case Feature(id) => prod contains id
  case And(f1,f2) => eval(f1) && eval(f2)
  case Or(f1,f2) => eval(f1) || eval(f2)
  case Imply(f1, f2) => !eval(f1) || eval(f2)
  case Not(f2) => !eval(f2)
  case Comp(v1,v2,op) => getResult(Comp(v1,v2,op))
  case True => true
```

Figure 24: Updated Eval Function

As a result of the changes made to the definition of FeatExpr, a new case named Comp was added, representing a comparison between two elements. When an expression of this type is evaluated, a helper function, getResult, is called. This function applies the logical operation represented by op between the operands v1 and v2. It determines the type of the values and executes the corresponding operation—such as equality, inequality, greater than, or less than—returning a Boolean value as intended.

Finally, it is necessary to enhance the parser so that it can correctly identify and capture comparison expressions whenever they appear in the Feature column of the Annotations tables in the Excel file. As shown in Table 12, the featComp rule was introduced, allowing the parsing of comparisons between two

elements. This rule, on the left-hand side of the expression, captures a featureVal representing a feature, the operator (op) which must be one of the supported comparison operators, and finally, featValOpt, which can be a number, a list, or text, as shown in Table 13.

Rule	Definition
featExpr	::= featImpl | ε
featImpl	::= featDisj [("->" | "=>" | "<->" | "<=>" | "#") featImpl]
featDisj	::= featConj ["||" featDisj]
featConj	::= literal ["&&" featConj]
literal	::= "(" featImpl ")"
	| "!" literal
	| "true"
	| "false"
	| featComp
	| feature
featComp	::= featureVal ("<" | ">" | "<=" | ">=" | "==" | "!=") featValOpt
feature	::= *identifier*
featureVal	::= *identifier*
featValOpt	::= *number* | "[*optionList*]" | *optText*

Table 12: Syntactic rules of the feature expression grammar

Token	::= Definition
identifier	[a-zA-Z0-9_][a-zA-Z0-9_]*
number	[0-9]+(\.[0-9]+)?
optionList	[a-zA-Z_,-]+
optText	[\??,a-zA-Z_,-]+

Table 13: Basic tokens of the feature expression grammar

4.3.3 Report generation

As previously mentioned, Uppex returns an HTML report summarising the configurations and the corresponding verified properties, along with their results — True if the property holds, or False otherwise. The

report is implemented as a class with the same name, Report, which includes attributes and methods to:

- Add Products
- Add results to the last added product (OK, Fail, Timeout).

This data structure is defined as:

```
Map[String, Map[String, Set[String]]]
```

That is:

$$\text{Product} \rightarrow \{\text{"ok"} \rightarrow \text{Set[Property]},\ \text{"fail"} \rightarrow \text{Set[Property]},\ \text{"timeout"} \rightarrow \text{Set[Property]}\}\}$$

With the introduction of the new backend and the addition of parametric models, it was necessary to implement a function to add the parametric constraints, called addConstraint.

$$\text{Product} \rightarrow \{\text{"ok"} \rightarrow \text{Set[Property]},\ \text{"fail"} \rightarrow \text{Set[Property]},\ \text{"timeout"} \rightarrow \text{Set[Property]}\},\ \text{"constrain"} \rightarrow \text{Set[Property]}\}\}$$

In addition, with the possibility of generating an image from the model, the addImage function was included, which allows associating the model images with the configurations. The structure is quite simple, consisting of a map of the configuration and the location where the model image is stored, if it exists.

```
Map[String, Set[String]]
```

That is:

$$\text{Product} \rightarrow \text{Set[Images]}$$

With all the information stored, the class also has a method that compiles this data into the final report. This method iterates through the data structures and aggregates the following into the three sections in which the report is structured:

- By Products

  The properties are grouped by products, with different formatting depending on whether they hold, do not hold, or have a parametric interval.

- By Property

  The grouping is done by property, with each product having a property according to the result of the respective property.

- Image By Product

  In this section, each product is assigned the visual format of the system.

4.3.4 Imitator Backend

Finally, it was necessary to implement a new backend, responsible for processing each configuration along with its properties, handling the obtained results, and generating the corresponding report.

As mentioned in the previous chapter, in the Bash terminal, a volume (local folder) will be mounted into the Docker container to allow IMITATOR to run with direct access to the files stored there. The backend assumes that this folder already exists; however, if it does not, it will be automatically created on the desktop with the name examples. This functionality is designed to work correctly on Windows, Linux, and macOS operating systems.

Similarly, for embedding the model images into the report, a folder named images must exist in the same directory as the JAR file. The same logic applies: if the folder does not exist, it will be created automatically.

We can divide the **Core of the Backend** into two main parts:

- **Function to store the properties to be verified:** This function aims to generate, for each property defined in the configurations, a separate file with the extension `.imipop`. Each file contains the property syntax exactly as specified in the Excel file and will later be verified against the model. These files are stored in a specific directory named `examples`, which is then mounted into the Docker container.

- **Function to run IMITATOR:** This function is responsible for executing IMITATOR inside a Docker container for each of the properties stored in the `examples` directory. It also checks whether the main `.imi` model file—with parameters, variables, and constraints updated according to the configuration—exists in that same directory. The function handles the individual execution of each property, interprets the returned results (e.g., `True`, `False`, or parametric constraints), and integrates the outcomes into the final report. Additionally, the function generates images of the models to be included in the final report; these images are stored in the `images` directory, located alongside the `.jar` file.

If the result is `True` or `False`, it is added to the report by invoking the `addOk` or `addFail` function, respectively. Furthermore, if any approximation was made by IMITATOR, this is also communicated to the user in the report. In the case of a parametric constraint, the newly mentioned function `addConstrain` is called, receiving the constraint as its argument.

We can summarize the process as follows:

1. **Creation of working directories:** Initially, the backend checks whether the required folders (`examples`, `images`, and `volume`) exist in the file system. If they do not, they are created automatically, ensuring compatibility across different operating systems (Windows, Linux, macOS).

2. **Generation of the updated model:** Based on the configurations provided by the user, the main model (`.imi`) is updated with the relevant parameters, variables, and constraints. This model is temporarily saved and will be used in all subsequent verifications.

3. **Creation of property files (`.imirop`):** For each property specified in the configurations, a separate file with the `.imirop` extension is created. These files contain the logical specification of the property to be verified. Each file is stored in the volume folder that will be mounted into the Docker container.

4. **Execution of IMITATOR:** The IMITATOR verifier is executed inside a Docker container. For each `.imirop` file, the backend invokes IMITATOR with the `.imi` model and the corresponding property, using the mounted volume containing the generated files. The system waits for the response and interprets the obtained result.

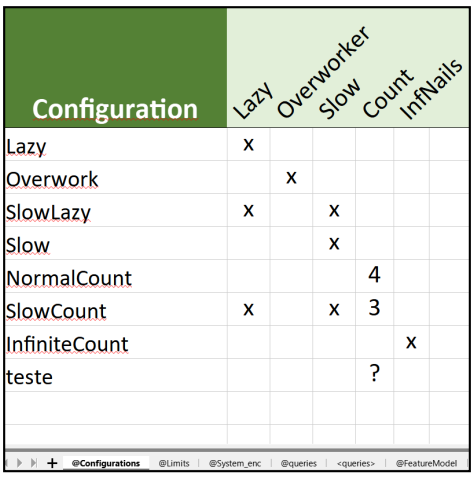
5. **Result handling:** Depending on IMITATOR's response (`True`, `False`, or a parametric formula), the result is classified as verified, not verified, or conditional. Additional information such as the type of approximation and execution time may also be recorded.

6. **Generation of the final report:** The results of each verification are compiled into a formatted report, with the option to include images of the analyzed models. These images are stored in the `images` folder and referenced in the report to enhance visualization and interpretation of the results.

4.4 Revisiting the Worker and Hammer example

For the implementation of the new Uppex, the Coffe Machine and Worker and Hammer system was selected.

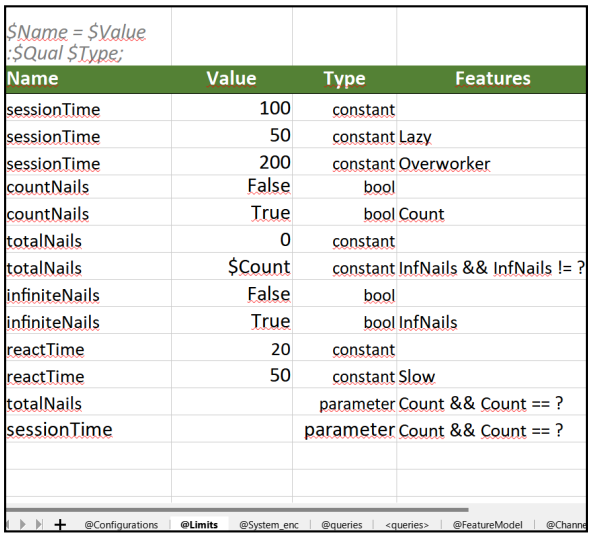
Starting with the Configuration sheet, the settings from the previous model were kept, and two new configurations were added. This allows for two groups—one non-parametric and the other parametric—in order to demonstrate the versatility of the tool.



Configuration	Lazy	Overworker	Slow	Count	InfNails
Lazy	x				
Overwork		x			
SlowLazy	x		x		
Slow			x		
NormalCount				4	
SlowCount	x		x	3	
InfiniteCount					x
teste				?	

Figure 25: Configurations Sheet Uppex

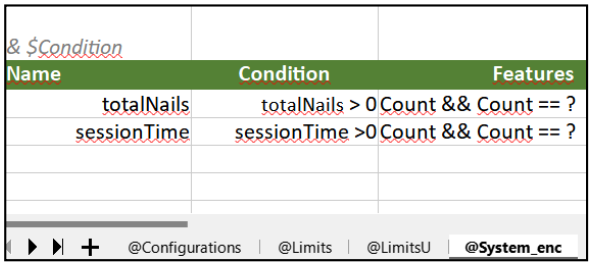
As we can see in Figure 25, the test configuration is a parametric configuration, with a ? in the product Count indicating its parametric nature. This can be interpreted as an uncertain or variable number selected for the Count product, unlike the previous configurations which have exact values.



$Name = $Value :$Qual $Type;

Name	Value	Type	Features
sessionTime	100	constant	
sessionTime	50	constant	Lazy
sessionTime	200	constant	Overworker
countNails	False	bool	
countNails	True	bool	Count
totalNails	0	constant	
totalNails	$Count	constant	InfNails && InfNails != ?
infiniteNails	False	bool	
infiniteNails	True	bool	InfNails
reactTime	20	constant	
reactTime	50	constant	Slow
totalNails		parameter	Count && Count == ?
sessionTime		parameter	Count && Count == ?

Figure 26: Limits Sheet



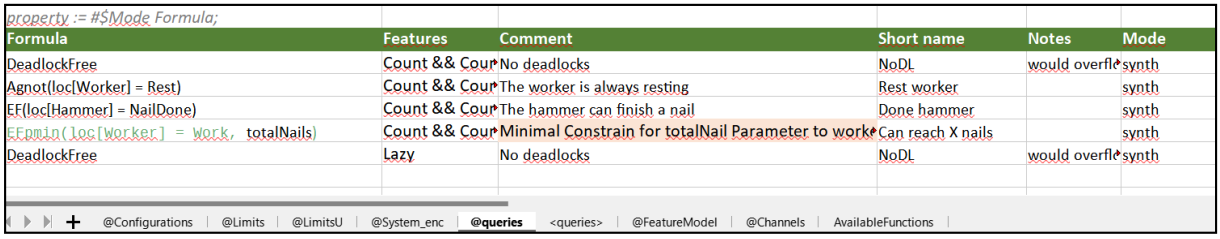
& $Condition

Name	Condition	Features
totalNails	totalNails > 0	Count && Count == ?
sessionTime	sessionTime >0	Count && Count == ?

Figure 27: Parameters Constrain Sheet

In the test configuration, sessionTime and totalNails are introduced as parameters, as shown in Figure 26, and the only condition imposed on these parameters is that they must be greater than 0, as seen in Figure 27. Additionally, the variable declaration pattern has changed compared to the previous version to accommodate the Imitator syntax.

Regarding the Queries Sheet Figure 28, in the non-parametric configurations that include the Lazy product, we check for the occurrence of deadlock. In the test configuration, in addition to this, we also verify the minimum value of the parameter totalNails required for the Work location in Worker to be reachable. Finally, the parameters constraint is checked to ensure that the NailDone location in Worker is reachable.



property := #$Mode Formula;

Formula	Features	Comment	Short name	Notes	Mode
DeadlockFree	Count && Cour	No deadlocks	NoDL	would overfl	synth
Agnot(loc[Worker] = Rest)	Count && Cour	The worker is always resting	Rest worker		synth
EF(loc[Hammer] = NailDone)	Count && Cour	The hammer can finish a nail	Done hammer		synth
EFpmin(loc[Worker] = Work, totalNails)	Count && Cour	Minimal Constrain for totalNail Parameter to work	Can reach X nails		synth
DeadlockFree	Lazy	No deadlocks	NoDL	would overfl	synth

@Configurations | @Limits | @LimitsU | @System_enc | @queries | <queries> | @FeatureModel | @Channels | AvailableFunctions

Figure 28: Queries Sheet Uppex

With the updated Excel file, we move on to the parameterization of the .imi file that encapsulates our model. As previously explained, the annotation blocks are identified by the format `(*Name*)`. In this case, we will have two blocks: one for the variables, named `(*Limits*)`, and another for the parameter constraints, named `(*System_enc*)`. For the non-parametric configurations, the Limits block will have the following structure:

```
(*@Limits*)
sessionTime = 100
: constant;
totalNails = 0
: constant;
countNails = True
: bool;
reactTime = 20
: constant;
infiniteNails = True
: bool;
```

Whereas in the test configuration, its content will differ due to the transformation of sessionTime and

totalNails into parameters:

```
(*@Limits*)
countNails = True
: bool;
infiniteNails = False
: bool;
reactTime = 20
: constant;
sessionTime
: parameter;
totalNails
: parameter;
```

In the terminal, we use the command: `java -jar uppex.jar -runAll Imitator_Model.imi`, thus compiling all configurations. The results can be seen in Appendix D. As previously explained, the groupby product and groupby requirement sections maintain the same format as the previous version; however, there is now a new type of result—specifically, the parametric interval, with an indication if an approximation has occurred. Finally, the new section includes the model image for each configuration, except for those not referenced in the queries sheet—that is, in the example, InfiniteCount, Main, NormalCount, Slow, and OverWork will remain empty.

Chapter 5

Towards a Graphical User Interface for Uppex

In order to make Uppex a more user-friendly tool for all types of users, initial steps were taken to develop an interface using Python and JavaScript.

Requirements. To be able to use the beta version of the tool, it is necessary to ensure that the following conditions are met: Python, the Java Virtual Machine (JVM), and Bash are installed, and Docker is available.

Installation. When the conditions above are met, the user can install this GUI by downloading and running the following Bash script: https://github.com/alexandre04032000/uppex-imitator/blob/Uppex-Imitator/interface.sh. If the process runs successfully, the application will be available on port 8080 of the localhost.

Running. This interface allows users to choose between the desired model checker, specifically between Uppaal and IMITATOR. Once one of these options is selected, a menu is displayed (as shown in Figure 29), allowing the user to choose from the following options:

- Validate the Model
- Run a Single Product
- Run All Products
- Change the Excel File
- Change .imi or .xml file

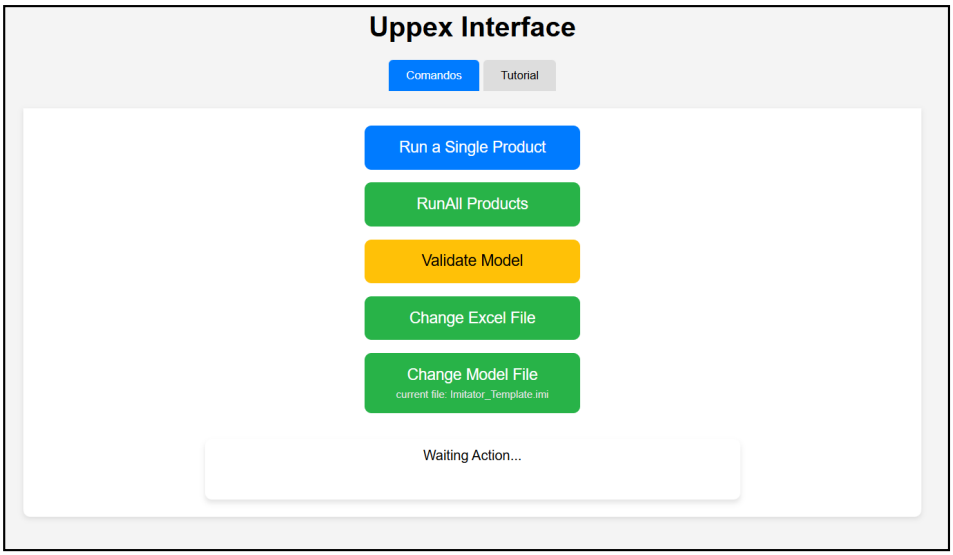


Figure 29: Uppex Interface Menu

It should be emphasized that, depending on the choice between Uppaal or Imitator in the menu shown in Figure 29, the extension of the file to be selected is already known in the last option, which also helps to identify which file is being read. There is also a separate tab called Tutorial which, as the name suggests, guides the user in interacting with the tool.

Chapter 6

Conclusions and future work

Conclusions

Model checking is a complex task that consists of the automatic verification of properties in finite-state systems, such as elevators or vending machines. When applied to real-time systems, this task often becomes intractable due to the state explosion problem, leading to high computational costs. To address this issue, variations of the model are created, simplifying certain aspects depending on the system's objectives, based on the theory of Software Product Lines. For this purpose, the Uppex tool is used, which, through a Microsoft Excel interface, reads the configurations selected by the user and builds an Uppaal model with small adjustments according to each configuration, also including the respective properties to be verified. As a result, it returns an HTML document summarizing the configurations, properties, and whether or not they were successfully verified.

The objective is to incorporate IMITATOR—a tool for the analysis and verification of real-time systems that also enables parameter synthesis—providing greater flexibility, optimization, and versatility to the system, as an additional backend and Model Checker within the Uppex tool. This objective was achieved through the changes made and demonstrated in Chapter 4. To test these modifications, the Worker-Hammer example was used, producing the expected results, notably the inclusion of parametric intervals in the output as well as the visual representation of the variations of the example model.

Prospect for future work

As future work, there are several improvements that can be made:

- The development of an user interface that encapsulates the current method of running the tool, which is currently done via command line. This would make it more appealing to professionals who

are less familiar with the area, while also simplifying the selection of the model checker type and the choice of available options.

- Improvement of the report generated from the system analysis, particularly the visual representation of the models. Many of the model images are often repeated, and to make the report more compact and less extensive, one possible enhancement would be to group the images that are similar together.

Appendix A
Full Imitator code of the hammer example

```
var
      session, t
           :clock;


      sessionTime,
      reactTime
           : parameter;


      nails
           : discrete;


      totalNails = 20
        : constant;




automaton Worker


synclabs: rest, hit, newNail, Work;

loc Rest: invariant session <= sessionTime - reactTime
      when True sync Work do {t := 0} goto Work;

loc Work: invariant session <= sessionTime && t <= reactTime
      when t >= reactTime sync newNail do {t := 0} goto Work;
      when t >= reactTime - 5 sync hit do {t := 0} goto Work;
      when session >= sessionTime sync rest do {session := 0} goto Rest;

end



automaton Hammer

synclabs: hit, newNail;
```

```
loc NailUp: invariant True
      when True sync hit goto NailHalf;

loc NailHalf: invariant True
      when countNails = True sync hit do {nails := nails +1} goto NailDone;

loc NailDone: invariant True
      when infiniteNails = True || nails <= totalNails sync newNail sync newNail goto Nailup

end


init :=


        & loc[Worker] = Rest
        & loc[Hammer] = NailUp


        & session = 0
        & t = 0

    & nails = 0

    & sessionTime >= 0
    & reactTime >= 0


;


end
```

Appendix B
Running Imitator with the hammer example

```
------------------------------------------------------------
Number of IPTAs                          : 2
Number of clocks                         : 2
Has invariants?                          : true
Has clocks with rate <>1?                : false
L/U subclass                             : U-PTA
Bounded parameters?                      : false
Has silent actions?                      : false
Is strongly deterministic?               : true
Number of parameters                     : 1
Number of discrete variables             : 1
Number of actions                        : 4
Total number of locations                : 5
Average locations per IPTA               : 2.5
Total number of transitions              : 7
Average transitions per IPTA             : 3.5
------------------------------------------------------------
```

```
BEGIN CONSTRAINT
 Parameter Constrain
END CONSTRAINT
```

```
------------------------------------------------------------
Constraint soundness                     : exact
Termination                              : regular termination
Constraint nature                        : good
------------------------------------------------------------
Number of states                         : 1
Number of transitions                    : 0
Number of computed states                : 2
Total computation time                   : 0.007 second
States/second in state space             : 128.6 (1/0.007 second)
Computed states/second                   : 257.2 (2/0.007 second)
Estimated memory                         : 2.078 MiB (i.e., 272498 words of size 8)
------------------------------------------------------------

------------------------------------------------------------
```

```
 Statistics: Algorithm counters
------------------------------------------------------------
main algorithm + parsing                : 0.043 second
main algorithm                          : 0.014 second
------------------------------------------------------------
 Statistics: Parsing counters
------------------------------------------------------------
model parsing and converting            : 0.025 second
------------------------------------------------------------
 Statistics: State computation counters
------------------------------------------------------------
number of state comparisons             : 0
number of constraints comparisons       : 0
number of new states <= old             : 0
number of new states >= old             : 0
StateSpace.merging attempts             : 0
StateSpace.merges                       : 0
------------------------------------------------------------
 Statistics: Graphics-related counters
------------------------------------------------------------
state space drawing                     : 0.000 second
------------------------------------------------------------
 Statistics: Global counter
------------------------------------------------------------
total                                   : 0.050 second
```

Appendix C
Uppex HTML Coffee Result

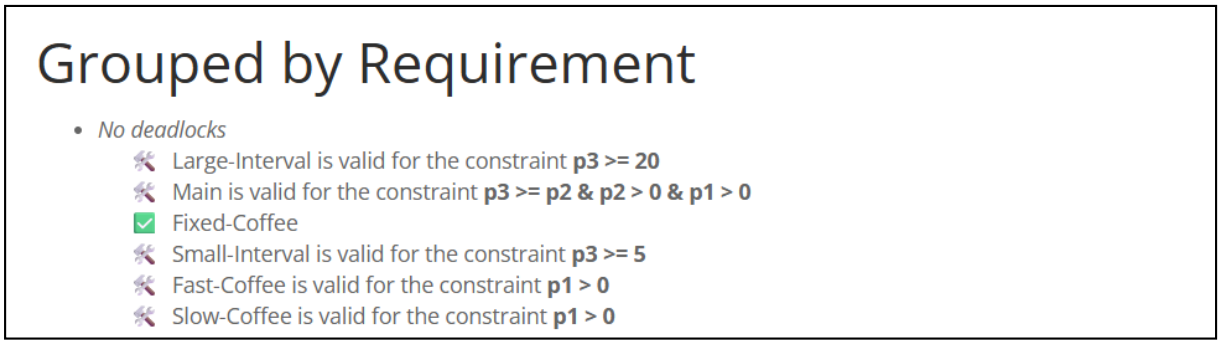


Figure 30: Grouped by Requerimets Report Result

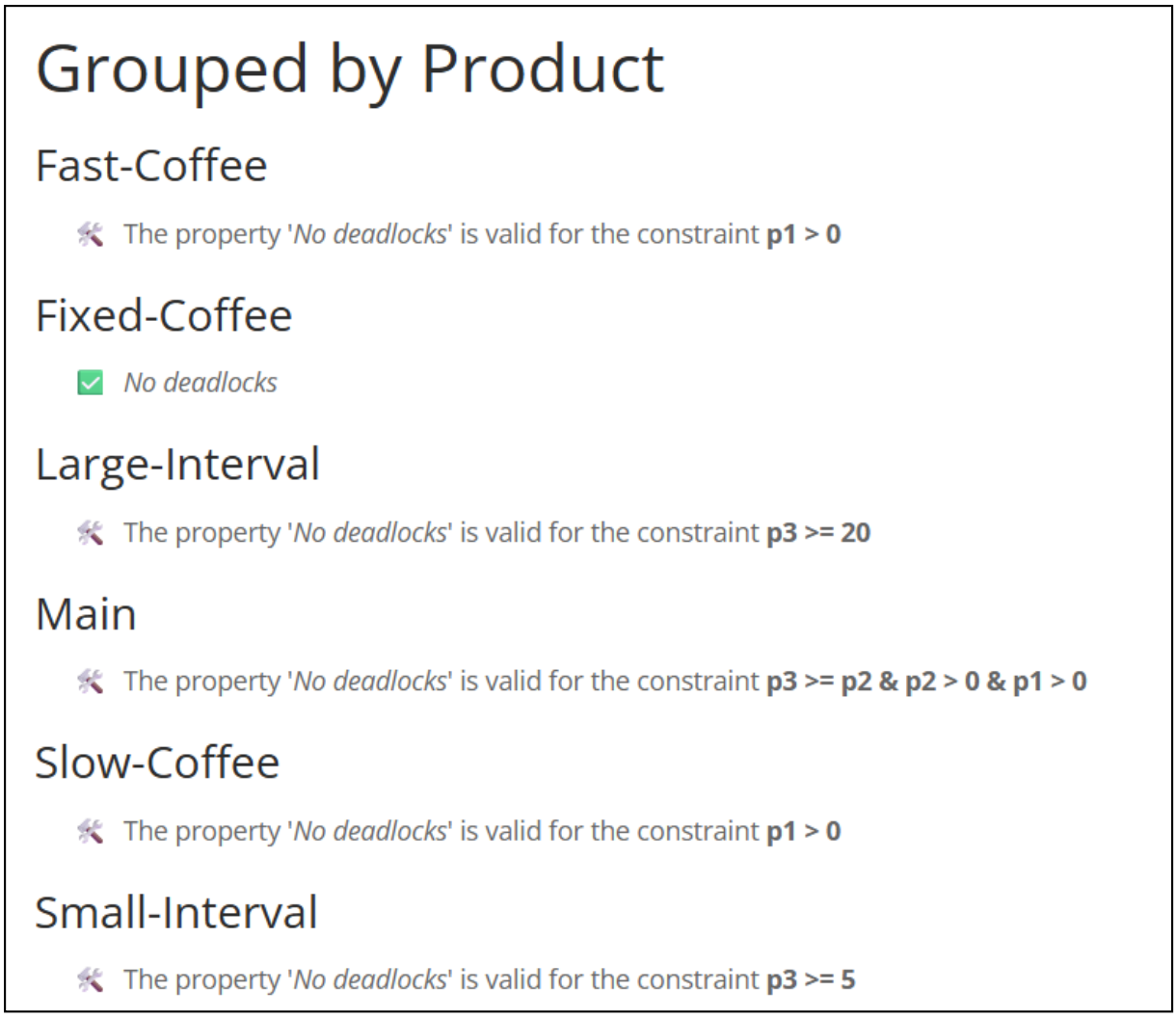


Figure 31: Grouped by Requerimets Report Result

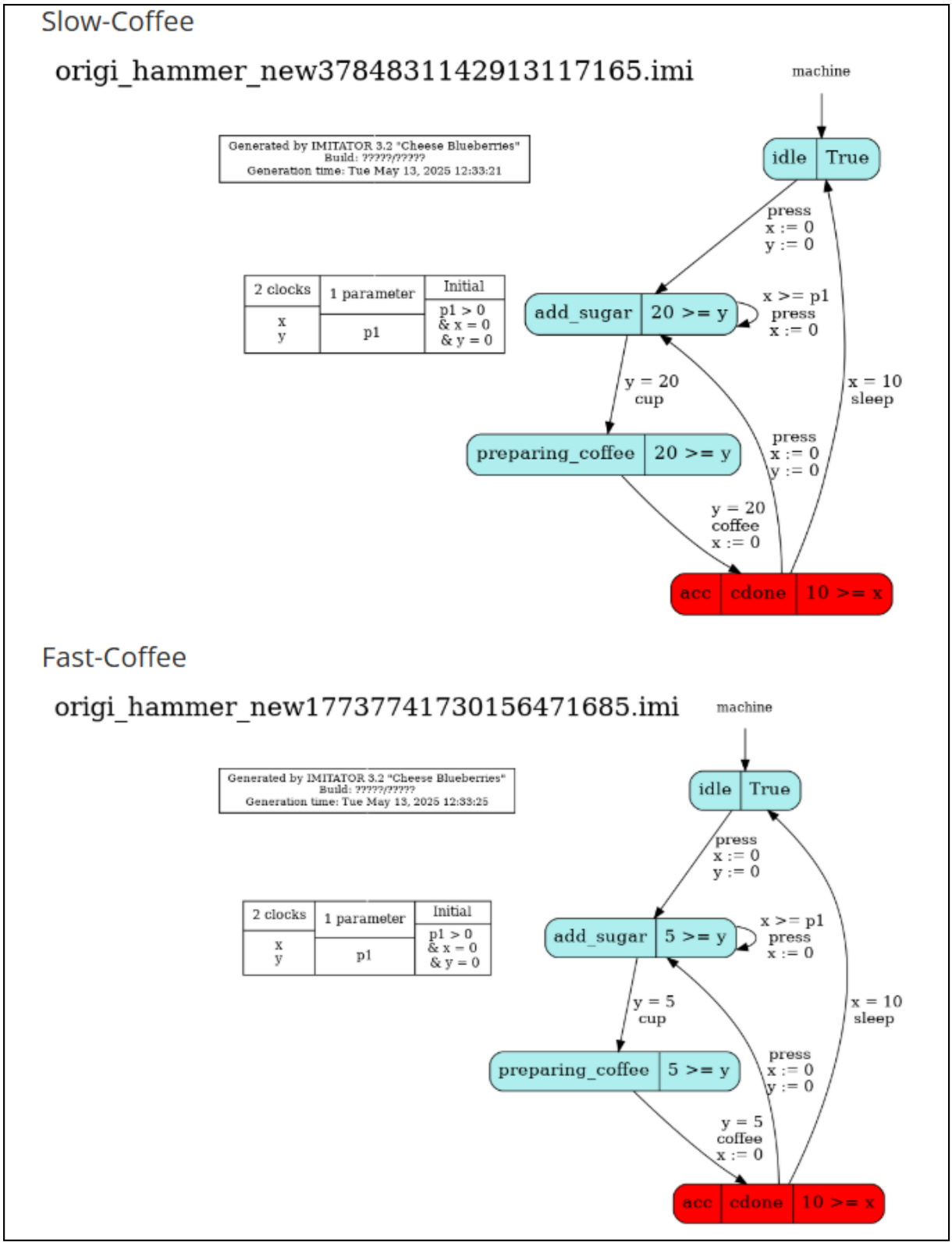


Figure 32: Coffee Automata Products Part 1 Report Result

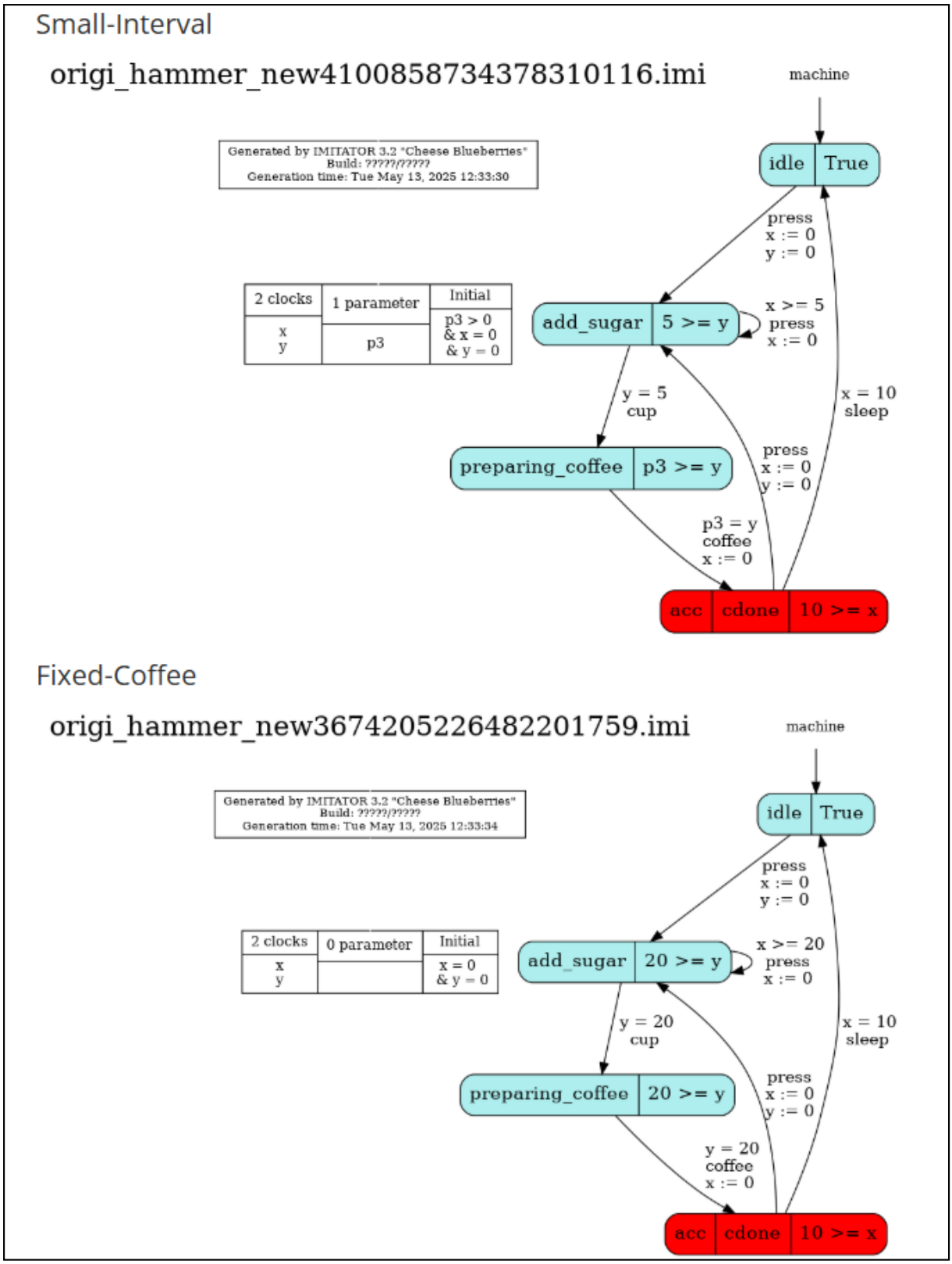


Figure 33: Coffee Automata Products Part 2 Report Result

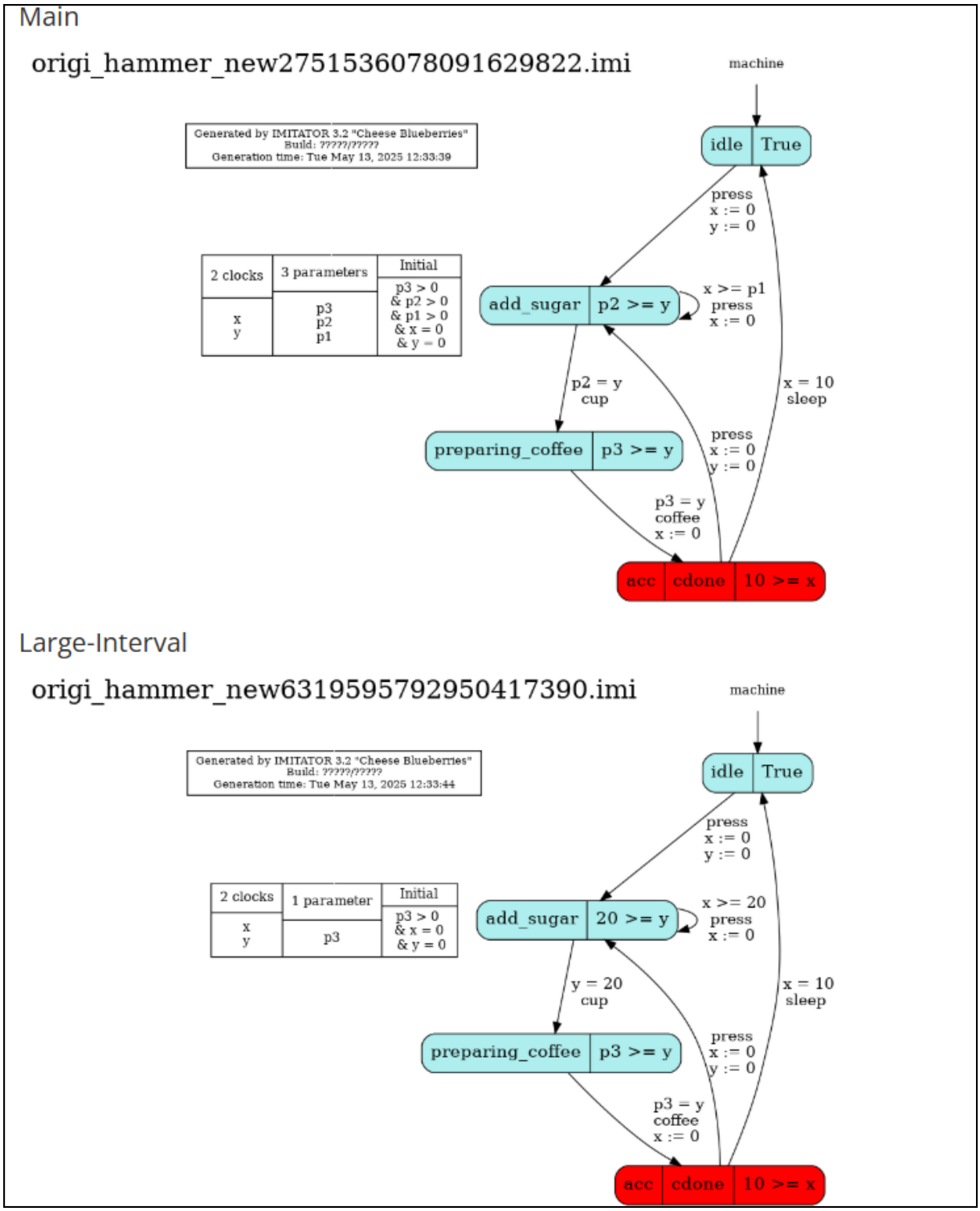


Figure 34: Coffee Automata Products Part 3 Report Result

Appendix D
Uppex HTML Hammer Result

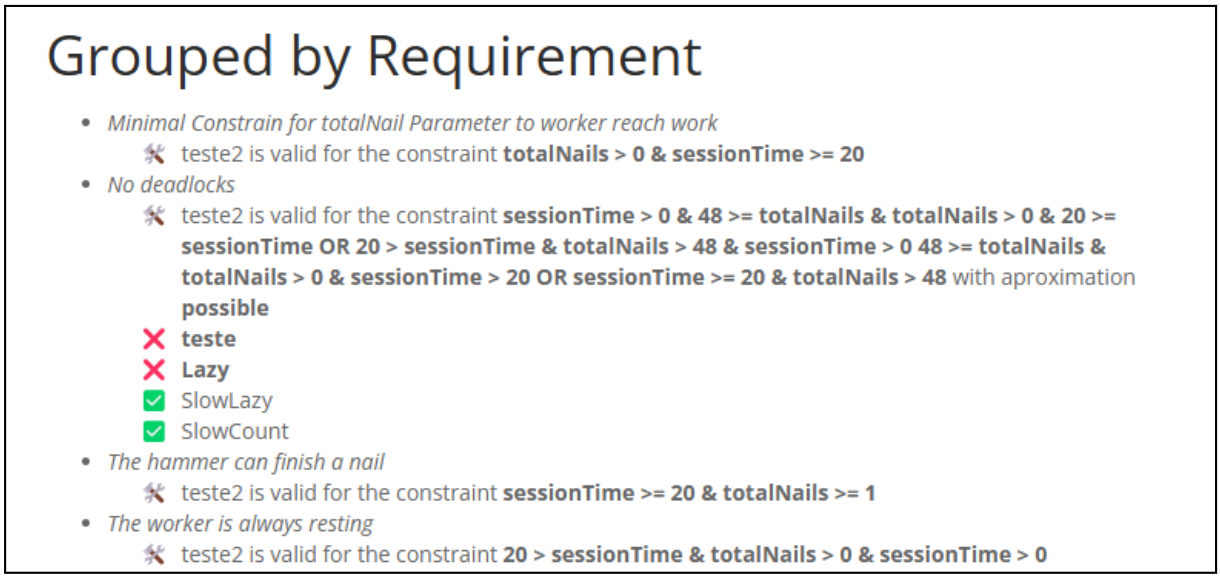


Figure 35: Grouped by Product Coffee Report Result

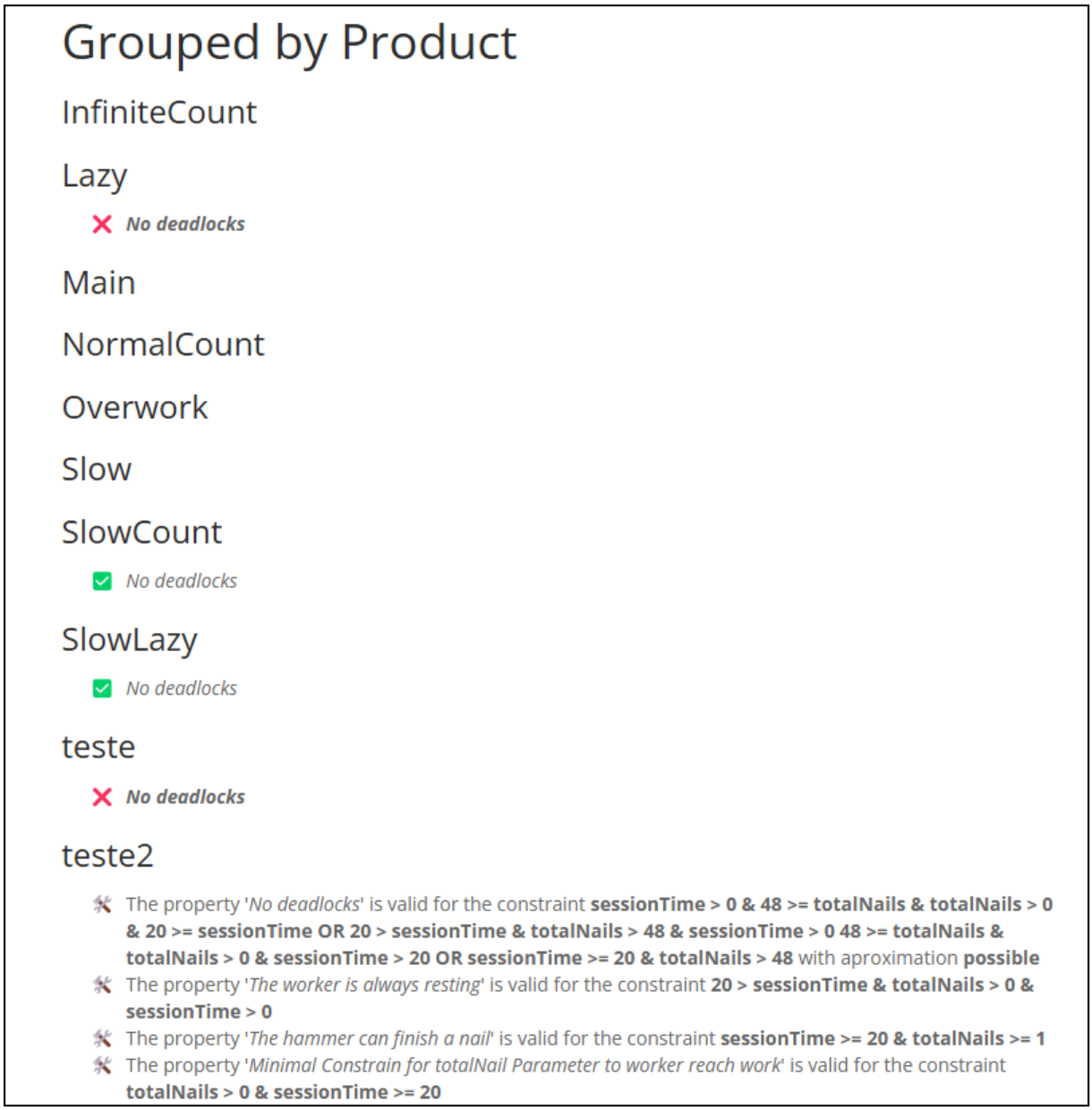


Figure 36: Grouped by Product Report Result

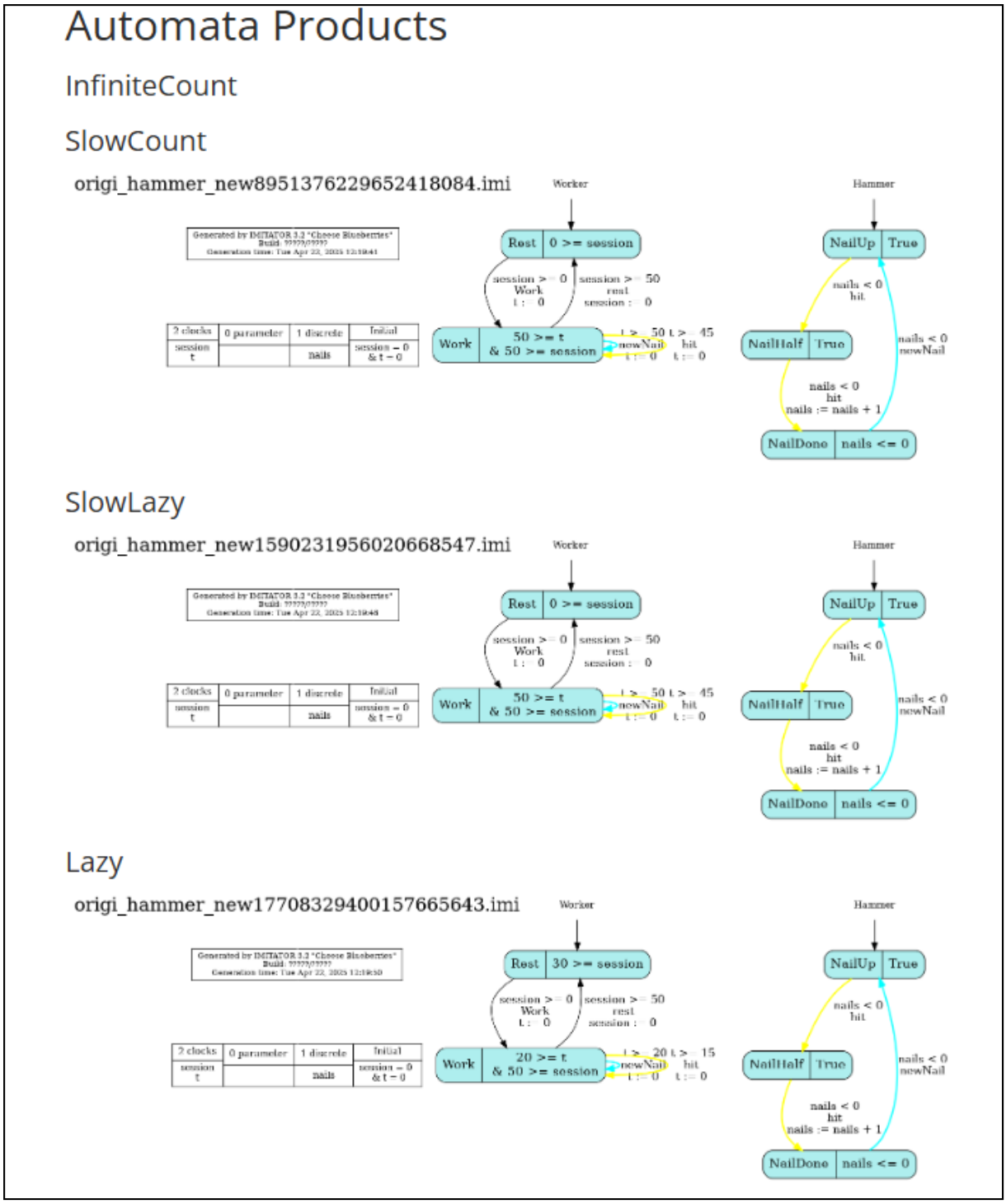


Figure 37: Automata Products Part 1 Report Result

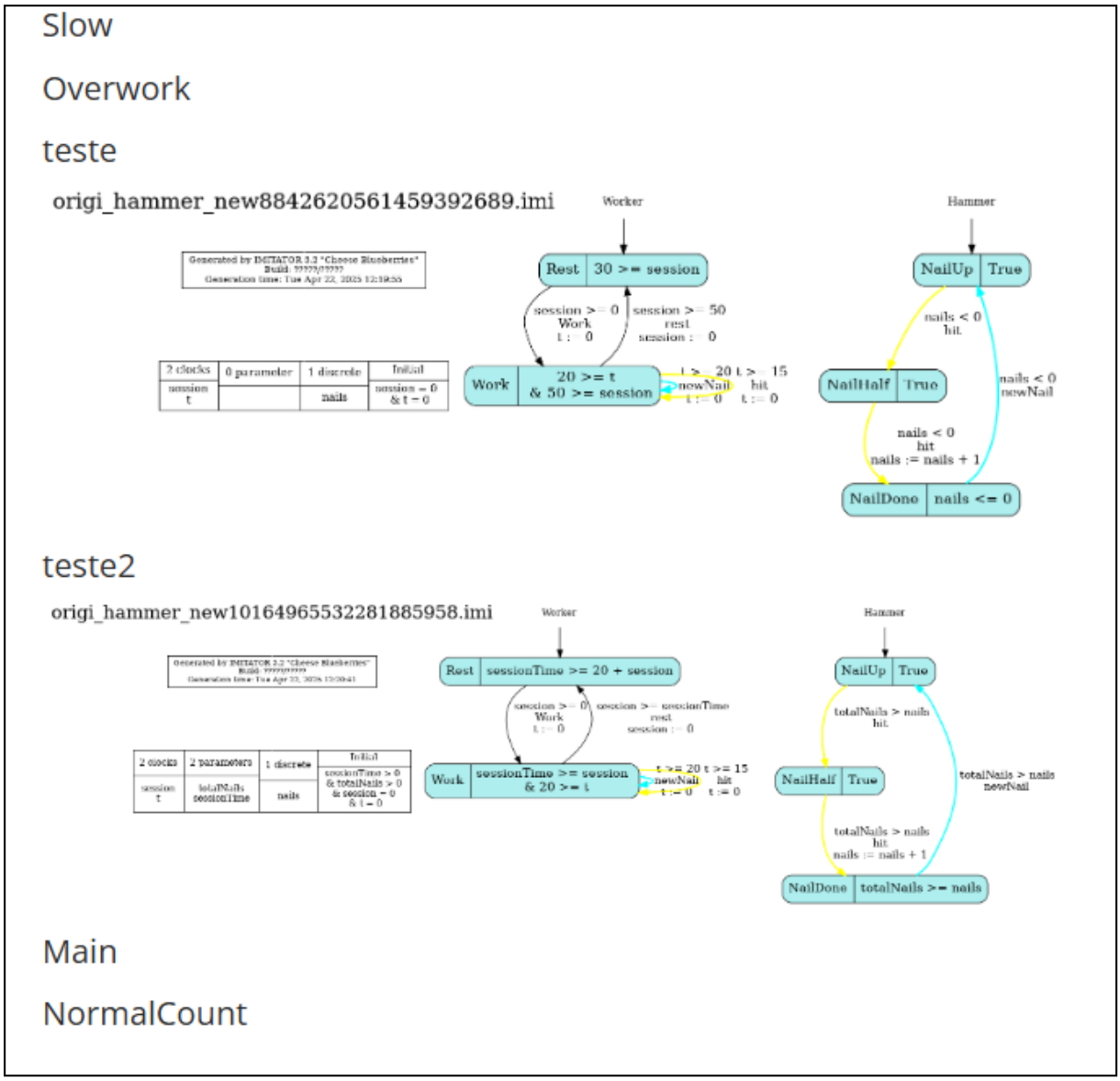


Figure 38: Automata Products Part 2 Report Result