E.T.S. INGENIERÍA INFORMÁTICA
MÁSTER EN INGENIERÍA DEL SOFTWARE E INTELIGENCIA ARTIFICIAL

Generación automática de autómatas temporizados mediante aprendizaje de trazas

Automatic generation of timed automata using trace-based Learning

Realizado por
Rafael López Gómez
Tutorizado por
María del Mar Gallardo Melgarejo
Cotutorizado por
Laura Panizo Jaime
Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, 26 de septiembre de 2023

Resumen:

En la actualidad, gracias al éxito indudable de distintas técnicas del área de la Inteligencia Artificial, se están desarrollando aplicaciones software cada vez más complejas, que se integran en la vida diaria de los ciudadanos. Sin embargo, la complejidad inherente a tales sistemas, que funcionan normalmente como *cajas negras* (la implementación es desconocida), hace casi imposible analizar su corrección, lo que es imprescindible cuando estas aplicaciones realizan tareas críticas. En el presente trabajo, se utilizan técnicas de *aprendizaje pasivo* (passive learning) para extraer de forma automática *modelos formales* de estas aplicaciones que pueden, posteriormente, ser analizados utilizando técnicas de verificación avanzadas como el model checking.

Las técnicas de aprendizaje pasivo utilizan el comportamiento *observado* del sistema bajo aprendizaje (system under learning/SUL) para construir un modelo formal del sistema. En el presente proyecto, los sistemas aprendidos son sistemas reactivos que interaccionan con su entorno y cuyo comportamiento depende del tiempo. La consideración del aspecto temporal de estos sistemas hace especialmente complicada su aprendizaje puesto que es una magnitud continua. Sin embargo, en contraposición, los modelos construidos son más precisos que los que no tienen en cuenta el tiempo.

Por otro lado, las observaciones del SUL son secuencias de estados observados (trazas observadas), en las que cada estado es una instantánea de cómo se encuentra el SUL en un momento dado. El modelo formal construido a partir de las observaciones es un *autómata temporizado* (timed automaton), que es un autómata en el que cada estado está enriquecido con variables de tipo reloj que evolucionan con el tiempo.

Como resultado del proyecto desarrollado se ha construido la herramienta *LearnTA* que contiene una componente principal que implementa el aprendizaje pasivo descrito. Sin embargo, además de esta componente se han estudiado e implementado otras herramientas auxiliares que permiten que se adapte a distintos tipos de SUL. Por ejemplo, se han construido aplicaciones para la extracción de las trazas observadas con un módulo de instrumentalización que les da un formato común. Así mismo, se ha construido una componente de verificación que comprueba que todas las trazas observadas del SUL son producidas por el autómata temporizado construido, lo que, de alguna forma, garantiza la relación de corrección entre el autómata y el SUL.

Con respecto a otros trabajos de la literatura, las principales contribuciones originales de este proyecto son la incorporación del parámetro

tiempo como actor principal de la herramienta *LearnTA*, y su utilización explícita para construir autómatas temporizados más fieles a los SUL de los que proceden. Así mismo, en el trabajo se han considerado una noción más enriquecida de los estados observados que permite la construcción de autómatas más precisos.

En el proyecto, se presenta también la evaluación de la herramienta *LearnTA* con distintos casos de estudio y su comparación en cuanto a corrección y eficiencia con otras de la literatura.

Palabras claves: Aprendizaje de Autómatas, Autómata temporizado, Aprendizaje pasivo.

Abstract:

Currently, thanks to the undoubted success of different techniques in the area of Artificial Intelligence, increasingly more complex software applications are being developed, which are integrated into the daily lives of citizens. However, the inherent complexity of such systems, which normally function as *black boxes* (the implementation is unknown), makes it almost impossible to analyze their correctness, which is essential when these applications perform critical tasks. In this project, *passive learning* techniques are used to automatically extract *formal models* of these applications, that can subsequently be analyzed using advanced verification techniques such as model checking.

Passive learning techniques use the *observed* behaviour of the system under learing (SUL) to build a formal model of the system. In this work, the learned systems are reactive systems, that interact with their environment and whose behaviour depends on time. The consideration of the temporary aspect of these systems makes their learning especially complicated, especially complicated since it is a continuous magnitude. However, the built models are more accurate than those that do not take time into account.

On the other hand, SUL observations are sequences of observed states (observed traces), in which each state is a snapshot of how the SUL is at a given moment. The formal model built from the observations is a *timed automaton*, which is an automaton in which each state is enriched with clock-like variables that evolve over time.

As a result of the developed project, the *LearnTA* tool has been built, which contains a main component thar implements the passive learning described. However, in addition to this component, other auxiliary tools have been studied and implemented that allow it to adapt to different types of SUL. For example, applications have been built to extract the traces observed with an instrumentation module that gives them a common format. Likewise, a verification component has been built that verifies that all the observed traces of the SUL are produced by the constructed timed automaton, which, in some way, guarantees the correctness relationship between the automaton and the SUL.

With respect to other works in the literature, the main original contributions of this project are the incorporation of the time parameter as the main actor of the *LearnTA* tool, and its explicit use to build timed automata that are more faithful to the SULs from which they come. In addition, in the work we have considered a more enriched notion of the observed states that allows the construction of more precise automata.

The project also presents the evaluation of the *LearnTA* tool with different case studies and its comparison in terms of correctness and efficiency with others in the literature.

Keywords: Automata Learning, Timed Automata, Passive Learning

Índice de contenidos

1. Introducción . . . 1
  1.1. Motivación . . . 1
  1.2. Objetivos . . . 2
  1.3. Estructura de la memoria . . . 4
2. Preliminares . . . 5
  2.1. Autómata temporizado . . . 5
    Valuación de relojes . . . 6
    Semántica de un Autómata Temporizado . . . 6
    Simplificaciones y subclases de autómata temporizado . . . 7
  2.2. K-Tail . . . 9
3. Estado del arte . . . 9
  3.1. Técnicas de aprendizaje pasivo . . . 10
    Timed k-Tail (TkT) . . . 10
    Real-Time Identification (RTI) . . . 11
    Real-Time Identification from positive data (RTI+) . . . 12
    Timed Automata Generator (TAG) . . . 12
  3.2. Métodos alternativos . . . 13
    Real Time Automata Learning (RTALearning) . . . 13
    Multi-mode hybrid Automaton Model (MOHA) . . . 13
    Genetic Programming for Timed Automata (GenProgTA) . . . 14
    Learning Timed Automata from Interaction Traces . . . 14
4. Diseño e implementación . . . 15
  4.1. Formato de las trazas . . . 17
  4.2. Autómatas del algoritmo . . . 17
  4.3. Algoritmo de aprendizaje . . . 19
    Algoritmo `CompressTraces` . . . 20
    Fase 1 . . . 22
    Fase 2 . . . 27
5. Pruebas y evaluación del algoritmo . . . 31
  5.1. Pruebas con trazas sintéticas generadas por modelos Uppaal . . . 32
  5.2. Pruebas con autómatas aleatorios . . . 35
  5.3. Comparación con el algoritmo TAG . . . 38
  5.4. Evaluación del algoritmo con un sistema real . . . 40
6. Conclusiones y líneas futuras . . . 40
  6.1. Trabajos Futuros . . . 41

Añadir fase de aprendizaje activo 41
Construcción de métodos híbridos 41
Diseño e implementación de lógica y verificador 42
Construcción de herramientas de visualización 42
A. Tecnologías 44
Java 44
Maven 44
Eclipse 45
JSON 45
Jackson 46
Uppaal 46
Graphviz Java 47
B. Resultados de las pruebas 48
B.1. Pruebas con trazas sintéticas generadas por modelos Uppaal 48
B.2. Modelo aprendido de las trazas del algoritmo DASH 50

Índice de figuras

1. Contexto general del proyecto 3
2. Ejemplo de autómata temporizado 8
3. Arquitectura de *LearnTA* 16
4. Ejemplo de formato de las trazas 18
5. Ejemplo de autómata temporizado resultante 20
6. Compresión de trazas caso 1 21
7. Compresión de trazas caso 2 22
8. Caso observación inicial con evento 23
9. Ejemplo `FastMerge` parte 1 26
10. Ejemplo `FastMerge` parte 2 27
11. Ejemplo de estados equivalentes 30
12. Operación Merge 32
13. Esquema de la portabilidad de Java 45
14. Ejemplo del formato JSON 46

Índice de tablas

1. Características de las trazas de los sistemas de UPPAAL 33
2. Configuración de parámetros para las pruebas con UPPAAL 33
3. Impacto del número de trazas en el aprendizaje 35
4. Impacto de las fases del algoritmo 36
5. Complejidades de los autómatas aleatorios 38
6. Resultados de las pruebas con autómatas aleatorios 38
7. Complejidades de los autómatas aleatorios en la comparación . . 39
8. Comparación de los algoritmos 39
9. Impacto de la longitud de las trazas en el aprendizaje 48
10. Impacto del valor del parámetro K en el aprendizaje 49

1. Introducción

1.1. Motivación

El modelado de sistemas complejos, como es el caso de sistemas reactivos que interaccionan frecuentemente con el entorno, se ha estudiado desde muchos enfoques en la literatura. El objetivo principal de la tarea de modelado es la obtención de una versión simplificada del sistema, normalmente una sobre-aproximación, en el sentido de que el modelo permite más comportamientos que el sistema original. La relación de sobre-aproximación entre el modelo y el sistema original debe establecerse, idealmente, de manera formal. Disponer de un modelo permite analizar y verificar las propiedades críticas principales del sistema original utilizando muchos menos recursos.

Las técnicas de modelado pueden agruparse en tres grandes categorías. En primer lugar, está la creación de modelos en la fase del diseño del sistema. Por otro lado, están las técnicas conocidas como "extracción de modelos" que parten de un sistema real cuya implementación es conocida y accesible, y extraen un esqueleto del mismo que puede ser analizado. Finalmente, cuando la implementación del sistema real es desconocida (caja negra) y sólo es observable su interacción con el entorno, las técnicas de modelado se concentran en extraer paulatinamente versiones intermedias del modelo. En este caso el aprendizaje se realiza a partir de las interacciones observadas, construyendo un modelo que responda ante los eventos externos del mismo modo que el sistema original. Es importante resaltar que cuando los sistemas a ser modelados tienen alguna componente continua, que evoluciona con el tiempo, su modelado, utilizando cualquiera de las técnicas mencionadas, se hace mucho más complicado.

Este trabajo se enmarca en la tercera categoría antes mencionada. En concreto, se tratará el problema de modelar el comportamiento de sistemas reactivos en los que su ejecución depende del tiempo (*temporizados*) y cuya implementación es desconocida. Como se ha mencionado anteriormente, en esta línea de técnicas de modelado solo son accesibles las llamadas "trazas de ejecución" que son secuencias de observaciones, ordenadas en el tiempo, del sistema en diferentes instantes durante su ejecución.

Aunque existen distintas técnicas que pueden utilizarse para esta tarea, en el proyecto exploramos el uso del de las técnicas de *Automata Learning*, del área de las Técnicas Formales iniciadas con el trabajo fundacional de Dana Angluin [3].

El objetivo de la técnica Automata Learning es la construcción de máquinas de estados (o autómatas de diversos tipos, dependiendo del método escogido) que tienen un comportamiento equivalente al sistema que se quiere aprender. El aprendizaje se produce mediante la interacción u observación del funcionamiento del sistema. Existen dos aproximaciones de esta técnica:

1. **Los métodos de aprendizaje activos** que siguen el modelo de Angluin y, en particular, el algoritmo L* [3], y que se caracterizan por hacer preguntas (queries) al sistema que se está aprendiendo (system under learning, SUL) para guiar la construcción del modelo formal. En este tipo de aprendizaje es necesario un *oráculo* que sirve de intermediario entre el sistema y el algoritmo de aprendizaje. Este oráculo es semejante a una API para extraer conocimiento del sistema.
2. **Los métodos de aprendizaje pasivos** que utilizan observaciones, llamadas trazas de ejecución, sin interactuar explícitamente con el sistema. El aprendizaje se realiza paulatinamente a medida que se van analizando las observaciones, generando autómatas intermedios que se corresponden con la información obtenida hasta ese momento. La ventaja que presentan con respecto al aprendizaje activo es que no dependen de la interacción con el SUL y, por tanto, los costes de desarrollo del entorno son menores. Como desventajas tenemos que la generación de trazas de ejecución es más costosa y los modelos resultantes pueden no ser completos ya que solo reflejan el comportamiento observado.

En este trabajo se utiliza la técnica de Automata Learning con aprendizaje pasivo para aprender el comportamiento de sistemas reactivos que evolucionan con el tiempo. Los modelos resultantes del aprendizaje serán *autómatas temporizados*. Los autómatas temporizados son el formalismo más adecuado para representar estos sistemas ya que integran de forma natural tanto su componente discreta, como su evolución con el tiempo. La construcción automática de autómatas temporizados a partir de trazas de ejecución conlleva una serie de retos como, por ejemplo, la necesidad de simplificar y restringir las capacidades del modelo resultante ya que no es posible obtener toda la información del sistema que se quiere aprender a partir de las trazas.

1.2. Objetivos

El objetivo de este proyecto es la realización de la herramienta *Learn-TA*, que incorpora un algoritmo basado en las técnicas de Automata Lear-

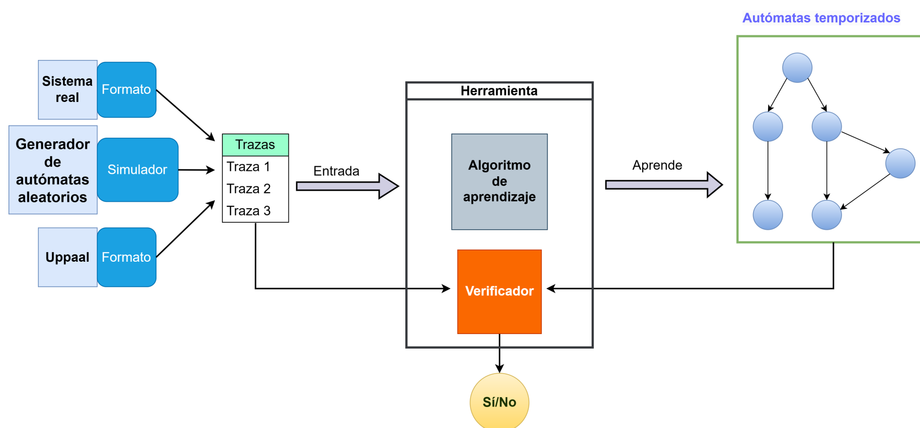


Figura 1: Contexto general del proyecto

ning con aprendizaje pasivo para la generación automática de autómatas temporizados que modelen el comportamiento de sistemas en los que el tiempo tiene un papel importante. En la Figura 1 se muestra una descripción general de todos los componentes y tareas que conforman el proyecto. El componente más importante de *LearnTA* es el algoritmo de aprendizaje, el cual es capaz de generar modelos de comportamiento, que son autómatas temporizados, de los sistemas originales a partir de sus trazas de ejecución. Para que el algoritmo sea capaz de procesar las trazas de ejecución de sistemas, es necesario definir un formato de entrada adecuado para transformarlas.

Como parte esencial de la herramienta, se ha construido un módulo de verificación (verificador) que es capaz de comprobar si el autómata generado reconoce las trazas de entrada, que podrían no ser las usadas para aprender el autómata. El verificador constituye un primer paso para la prueba de la *corrección formal* del autómata generado y ha sido utilizado para realizar pruebas que comprueben la eficacia del algoritmo de aprendizaje.

Por último, otra tarea realizada en el proyecto es la de encontrar y/o emular sistemas temporizados reales que sirvan de ejemplo para mostrar la utilidad de *LearnTA*. En este trabajo, se han explorado tres líneas. Por un lado, se ha utilizado un sistema real del que se han extraído diferentes trazas para la construcción del autómata. Por otro lado, se ha construido un generador de autómatas aleatorios que emulan el comportamiento de sistemas reales. En el generador se pueden configurar diferentes parámetros para controlar la complejidad de los autómatas resultantes. Junto al

generador se ha construido un simulador que permite construir trazas de ejecución a partir de los autómatas aleatorios. El simulador puede generar tanto trazas de ejecución del autómata como trazas "erróneas" (que no son reconocidas por el autómata) lo que será muy útil para la evaluación de *LearnTA*. Finalmente, como tercera fuente de ejemplos, se ha utilizado Uppaal [22], que es una herramienta para el modelado, validación y verificación de sistemas en tiempo real que utiliza redes de autómatas temporizados. Se han modificado sistemas ejemplo de la herramienta Uppaal y se ha construido un programa para poder extraer trazas con el formato adecuado.

A modo de recapitulación, este proyecto ha consistido en las siguientes tareas:

- Definición del formato de las trazas formadas por secuencias de observaciones que serán la entrada de la herramienta *LearnTA*.
- Diseño y construcción del tipo de estructura que de los autómatas temporizados producidos por *LearnTA*..
- Diseño e implementación de la herramienta *LearnTA*, la cual contiene un algoritmo de Automata Learning con aprendizaje pasivo, el verificador y diversas funciones para utilizarla.
- Diseño e implementación del generador de autómatas aleatorios y del simulador de ejecución para la extracción de trazas con el formato de entrada.
- Modificación de modelos de Uppaal y construcción de programas para la generación de trazas de ejecución con el formato adecuado.
- Selección de casos de estudio relevantes que puedan resaltar el potencial del software producido.
- Realización y evaluación de pruebas.

1.3. Estructura de la memoria

Sección 2. Preliminares

En esta sección se introducen los términos y conceptos utilizados a lo largo de la memoria.

Sección 3. Estado del arte

En esta sección se realiza un análisis del estado del arte en el que se recogen las aportaciones existentes más importantes en la línea de la generación automática de autómatas temporizados para modelar el comportamiento de un sistema. El estudio se realiza de forma general, incluyendo

artículos que utilizan tanto algoritmos dentro del Automata Learning, ya sean pasivos o activos, como métodos alternativos.

Sección 4. Diseño e implementación

En esta es la sección se describe con detalle el proceso de diseño e implementación de los componentes de la herramienta *LearnTA*. En particular, se describe algoritmo de aprendizaje, el formato de las trazas de ejecución y la estructura de los autómatas temporizados resultantes del aprendizaje.

Sección 5. Pruebas y casos de estudio

En esta sección se presentan las pruebas realizadas con la *LearnTA* y se hace comparación detallada con otra propuesta del estado del arte actual.

Sección 6. Conclusiones y líneas futuras

La última sección resume las aportaciones del trabajo y explora posibles líneas futuras de trabajo.

2. Preliminares

Esta sección está dedicada a introducir formalmente los conceptos que van a estar presentes a lo largo de la memoria, como es la definición de autómata temporizado.

2.1. Autómata temporizado

El concepto de *autómata temporizado* fue descrito por primera vez por Alur [1] en 1992.

Un autómata temporizado (TA) es una tupla (L, l_0, C, A, E, I) donde:

- L es el conjunto de locations del sistema.
- l_0 es la location inicial.
- C es el conjunto de variables de tipo reloj (clocks). Denotamos con $B(C)$ el conjunto de restricciones sobre los relojes de C.

- A es el conjunto de acciones, tanto las acciones internas del sistema como aquellas que sirven de sincronización con el entorno.
- $E \subseteq L \times A \times B(C) \times 2^C \times L$ son las transiciones etiquetadas del autómata. Una transición presenta la forma (q, a, g, r, q') donde:
  - q y q' son la location fuente y la location destino, respectivamente.
  - a es una acción de A.
  - $g \in B(C)$ son las restricciones de tiempo que constituyen la guarda de la transición.
  - r es una función que reseta el valor de algunos relojes a 0, y mantiene el mismo valor en el resto.
- La función $I : L \to B(C)$ asigna una condición invariante (restricción de tiempo) a cada location.

Valuación de relojes
Una valuación de relojes es una función $u : C \to \mathbb{R}^{\geq 0}$ que tiene como entrada un conjunto de relojes, y como salida el valor de cada reloj como un número real no negativo:

- Si $x \in C$ es un reloj, y u es una valuación, entonces $u(x)$ es el valor del reloj dado por u.
- Denotamos la valuación inicial en la que todos los relojes están inicializados a 0 con u_0 definida como $\forall x \in C.\ u_0(x) = 0$

El conjunto de todas las valuaciones de los relojes de C se denota como $\mathbb{R}^C$.

Semántica de un Autómata Temporizado
Dado un TA (L, l_0, C, A, E, I), su semántica viene dada por un sistema de transiciones etiquetadas $\langle S, s_0, \xrightarrow{-} \rangle$ donde:

- $S \subseteq L \times \mathbb{R}^C$ es el conjunto de estados.
- $s_0 = (l_0, u_0) \in S$ es el estado inicial.
- $\xrightarrow{-} \subseteq S \times (\mathbb{R}^{\geq 0} \cup A) \times S$ es la relación de transición.

Los estados son pares constituidos por la location actual del autómata y la valuación de todos los relojes.

La relación de transición $\xrightarrow{-}$ modela la evolución del sistema de transiciones de un estado fuente a otro destino. La etiqueta de la transición puede ser una acción (transición discreta) o bien un número real (transición continua) que representa el paso del tiempo. La relación de transición $\xrightarrow{-} \subseteq S \times (\mathbb{R}^{>0} \cup A) \times S$ se define formalmente como:

- Transición continua: $(l, u) \xrightarrow{d} (l, u + d)$ si y solo si $\forall d', 0 \leq d' \leq d$: $u + d' \in I(l)$
- Transición discreta: $(l, u) \xrightarrow{a} (l', u')$ si y solo si $\exists (l, a, g, r, l') \in E$ tal que $u \in g,\ u' = [r \rightarrow 0]u$, y $u' \in I(l')$

  donde:

1. $a \in A$
2. $d \in \mathbb{R}^{>0}$, es el tiempo transcurrido.
3. $u + d$ es la valuación que asigna a cada reloj $x \in C$ el valor $u(x) + d$.
4. $[r \rightarrow 0]u$ es la valuación que asigna a todos los relojes $x \in C - r$ el mismo valor que u, es decir, $u(x)$, mientras que a los relojes de r les asigna el valor 0.

Tomando como ejemplo el autómata temporizado de la Figura 2, que presenta un modelo que simula el comportamiento de una lámpara. Dicho modelo presenta:

- Tres locations: "off", "tenue" y "brillante".
- La location inicial es "off".
- Un único reloj del sistema "y".
- Los invariantes de las locations son todos "true" por tanto no aparecen como condición.

Dicha lámpara empieza en la location "off", que representa que está apagada. Posteriormente, un usuario puede pulsarla para encenderla con una luz tenue y que pase a "tenue"; esto queda reflejado en la figura con la transición con la acción "press?". Además, cuando es pulsada, su reloj interno ("y") se reinicia a 0. Cuando está encendida con luz tenue (location "tenue") si es pulsada otra vez en un intervalo de 5 segundos, pasará a aumentar la intensidad de luz, es decir, transitará a la location "brillante". En caso contrario, si es pulsada después de cinco segundos, simplemente se apagará (volverá a la location inicial "off"). Finalmente, cuando la lámpara esta iluminando con una luz brillante, puede ser apagada si alguien la vuelve a presionar, en otras palabras, permanecerá en la location "brillante" hasta que ocurra el evento "press?", y una vez que ocurra transitará a la location "off".

Simplificaciones y subclases de autómata temporizado

El análisis de los sistemas incluyendo el factor temporal así como la construcción de autómatas temporizados que los modelen es muy costoso y en algunos casos es un problema indecidible. Por ello, en la literatura

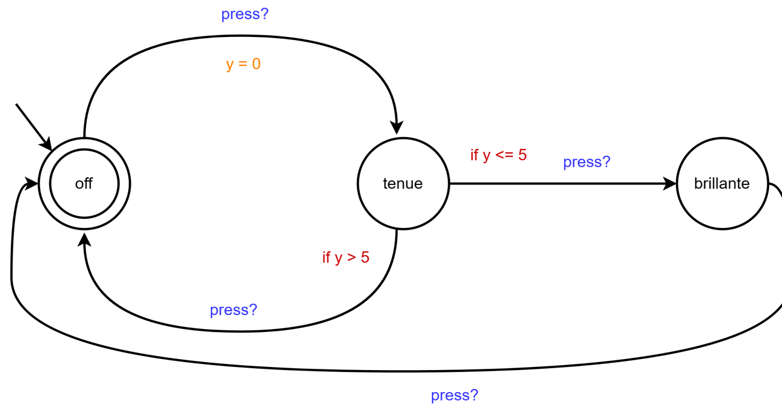


Figura 2: Ejemplo de autómata temporizado

se han aplicado una serie de simplificaciones, las más relevantes serán comentadas a continuación.

En primer lugar, no se consideran los **invariante**s incluidos por definición formal ya que en la práctica es complicado realizar la distinción entre guarda e invariante utilizando exclusivamente la información que aportan las trazas.

Por simplicidad, a partir de ahora se ha decidido llamar a las **location**s como **estado**s.

Finalmente, en la literatura se definen diferentes subclases de autómatas temporizados más acotados con el fin de posibilitar el análisis automático de dichos sistemas. Algunos de ellos son:

1. Autómata temporizado determinista (DTA): es un autómata temporizado que no permite que desde una location existan dos transiciones que tengan la misma etiqueta y guardas que se solapen y que vayan hacia diferentes locations.
2. 1DTA: es un DTA con un único reloj.
3. Autómata de tiempo real determinista (DRTA): variante de 1DTA donde el único reloj se resetea en cada transición para representar el intervalo de tiempo entre eventos. Las guardas de tiempo se definen como intervalos cerrados. Este es el tipo de autómata que genera el algoritmo de aprendizaje de *LearnTA*

4. Autómata de tiempo real probabilístico y determinista (PDRTA): DRTA que añade probabilidades a la ocurrencia de los eventos o acciones en las transiciones.

2.2. K-Tail

Los algoritmos que se han desarrollado en este proyecto están basados en K-Tail [4], un conocido algoritmo de inferencia que es capaz de generar un autómata finito determinista a partir de un conjunto de trazas de ejecución. Cabe destacar que K-Tail no tiene en cuenta el factor temporal y los autómatas generados no son temporizados. Su funcionamiento consiste en dos pasos. En el primer paso, genera un "Prefix Tree Automaton" (PTA), es decir, un autómata en forma de árbol donde cada rama acepta una de las trazas de ejecución de entrada. En el segundo paso, K-Tail modifica iterativamente el autómata fusionando los estados del modelo que probablemente representen el mismo estado en el sistema original. Estos estados equivalentes se identifican heurísticamente en función de su K-futuro, que consiste en el conjunto de secuencias de una traza que se aceptan a partir de ese estado. Se supone que dos estados con el mismo K-futuro son equivalentes y, por tanto, se fusionan. El proceso de fusión de estados produce un modelo más compacto y más general que el modelo inicial; de hecho, el autómata resultante tiene menos estados y acepta más trazas de ejecución que el inicial (es una sobre-aproximación). El algoritmo K-Tail ha servido de base para muchas otras técnicas que lo han mejorado o modificado con el fin, entre otros propósitos, de generar modelos de mayor calidad.

3. Estado del arte

En la actualidad, hay una gran cantidad de trabajos relativos a las técnicas de Automata Learning para el aprendizaje de sistemas en los que el tiempo no influye. Sin embargo, cuando se trata de aprender sistemas en los que se imponen restricciones de tiempo, hay numerosos retos que han dificultado la proliferación de aportaciones. Algunos de estos problemas son la gran cantidad de parámetros que hay que ajustar para producir un autómata temporizado o la falta de precisión en el resultado.

Tal y como se mencionó anteriormente, el enfoque de este proyecto es utilizar aprendizaje pasivo, sin interactuar directamente con el sistema. Sin embargo, en esta sección, se realizará un estudio de las aportaciones más relevantes en relación a la generación de autómatas temporizados

que sirvan como modelo de un sistema cuyo comportamiento se quiera aprender. El estado del arte se ha dividido en dos secciones: la primera presenta técnicas de aprendizaje pasivo, la segunda se centra en otros métodos existentes como el aprendizaje activo, o que quedan fuera del marco del Automata Learning pero tienen el mismo objetivo de aprender un autómata temporizado a partir del comportamiento de un sistema.

3.1. Técnicas de aprendizaje pasivo

Timed k-Tail (TkT)
El artículo [19] presenta Timed k-tail (TkT), un algoritmo que es capaz de recoger los aspectos temporales del sistema para así crear autómatas temporizados a partir de trazas de ejecución. TkT se basa en el algoritmo de inferencia de modelos **K-Tail** [4], expandiéndolo para poder generar elementos de los autómatas temporizados como relojes, operaciones de reset y guardas condicionales.

TkT parte de un conjunto de trazas con información acerca del tiempo y del orden de las operaciones que han sido ejecutadas por un sistema. Las trazas han de tener un formato específico en el que cada proceso ha de indicar tanto su inicio como su final. Entre el comienzo y el final de un proceso pueden ocurrir otros que serán considerados como sub-procesos del mismo. A partir de de las tarzas, TkT es capaz de construir automáticamente un autómata temporizado (TA) que capture el comportamiento general del software estudiado así como secuencias de operaciones anidadas si las hubiera. El algoritmo tiene cinco pasos:

- Paso 1. Normalización de trazas: En este paso se normaliza el tiempo de las trazas para que el primer evento de cada traza comience con tiempo 0 y se ajustan los demás eventos consecuentemente.
- Paso 2. Inicialización del autómata: TkT produce un autómata inicial donde cada traza es una "rama" distinta. En esta fase el autómata presenta relojes relativos al tiempo que tarda en ejecutarse un proceso, es decir, el evento de su inicio y el de su final. Si hay varias apariciones del mismo proceso, cada una se asocia con relojes distintos. Además, hay un reloj absoluto que lleva la cuenta del tiempo total transcurrido.
- Paso 3. Mezcla de estados: En esta fase se mezclan los estados que se consideran equivalentes así como transiciones que sean redundantes cuando se unen dichos estados. TkT define como estados equivalentes a aquellos que aceptan las mismas secuencias de eventos futuros con una longitud k, en otras palabras, que presenten los mismos k-futuros

(véase el algoritmo K-Tail de la sección 2.2). TkT presenta una longitud k fijada en 2. Ha de destacarse que en la definición de equivalencia el tiempo no se tiene en cuenta.

- Paso 4. Refinamiento de relojes: En la operación anterior, solo se unen los estados pero los relojes asociados a ellos necesitan ser tratados. En esta fase se reduce el número de relojes eliminando los redundantes que miden la duración de la misma operación.
- Paso 5. Generación de guardas: En el último paso, se procesan las restricciones de tiempo de cada reloj para producir intervalos temporales.

Real-Time Identification (RTI)

En el artículo [24] se presenta el algoritmo RTI, un algoritmo de aprendizaje que genera un DRTA a partir de datos de muestra, que contienen tanto ejemplos de trazas que pertenecen al sistema (trazas positivas) y que el autómata resultante debe de aceptar, como trazas que son erróneas y no son propias del sistema que se quiere aprender (trazas negativas) y que, por tanto, el autómata no debe de admitir. La idea es empezar con la construcción de un autómata con forma de árbol denominado como "Prefix Tree Automaton" (PTA) a partir de las trazas, y a partir de él ir mezclando sus estados. En este autómata cada traza de entrada se corresponde con una rama del árbol. Los nodos en los que acaba una cadena de entrada son marcados tanto para indicar que era un ejemplo positivo como negativo. Cada transición tiene una guarda temporal cuyos valores se inicializan de acuerdo a los valores obtenidos de los datos de entrada. Posteriormente, se realiza un proceso iterativo de operaciones que colorean el árbol en dos colores. El color rojo representa que un nodo es una parte fija del autómata definitivo y que por tanto no puede modificarse; el color azul representa que es un candidato a ser modificado, si un nodo no está coloreado es una pieza del árbol original que aún no está siendo considerada. Este proceso iterativo comienza coloreando la raíz del autómata de color rojo y sus hijos de color azul. En cada iteración puede mezclarse un nodo rojo con otro azul, se puede separar una transición que acabe en nodos de color azul (no procesados) o se puede marcar de color rojo un nodo azul si no hay una mezcla posible. Una vez que acaba un paso, se colorea de azul a todos los hijos de los nuevos nodos rojos. Para determinar la operación a realizar, el algoritmo posee una heurística que tiene en cuenta el tiempo y que decide que operación aplicar computando cual ofrece unos mejores resultados si se realiza.

Real-Time Identification from positive data (RTI+)
En el artículo [25], los autores presentan una mejora del algoritmo RTI+ descrito en [24]. El algoritmo soluciona alguno de los problemas de la primera versión, como es la dificultad de obtener trazas negativas de un sistema, es decir, trazas que incluyen comportamientos erróneos. En esta nueva versión solo se utilizan trazas positivas producidas por un sistema, que son fácilmente obtenibles a través de su ejecución. Además, el autómata resultante es ahora un PDRTA, pues le añaden probabilidades a las transiciones. Al igual que en su versión anterior, el algoritmo comienza generando un PTA donde cada rama acepta una traza de ejecución de entrada. Cada estado es accesible mediante un único camino ya que no aún no se ha realizado ninguna operación de mezcla o separación, por lo tanto, es consistente con los datos de entrada. Las guardas de tiempo se inicializan con respecto a los valores máximo y mínimo del tiempo de las observaciones procesadas. Una vez que se ha construido el PTA, el algoritmo trata de mezclar estados y separar transiciones usando un coloreado rojo-azul similar al usado en RTI. La diferencia más destacable radica en la heurística para decidir la operación realizar, que ahora se basa en "likelihood-ratio tests" (pruebas de razón de verosimilitud) que son pruebas estadísticas que evalúan la calidad de modelos que compiten y que hacen uso de las probabilidades añadidas a las transiciones. El inconveniente principal de este algoritmo es su tratamiento de las restricciones temporales que derivan en guardas de tiempo incorrectas o incluso la ausencia de las mismas.

Timed Automata Generator (TAG)
La aportación del artículo [5] es TAG, un nuevo algoritmo de Automata Learning con aprendizaje pasivo que está basado en **K-Tail** [4]. TAG aprende un PDRTA para describir y entender un sistema dependiente del tiempo a partir de sus trazas de ejecución.

El algoritmo comienza generando un autómata con forma de árbol a partir de todas las trazas de entrada. Posteriormente, reduce el tamaño del autómata realizando continuamente mezcla de estados equivalentes hasta que no queda ninguna unión pendiente. Al igual muchos algoritmos basados en K-Tail, dos estados son equivalentes si poseen las mismas secuencias futuras con una longitud K (en este algoritmo el parámetro K es modificable). Una vez ha terminado la fase de mezcla de estados, se procede a la fase de separación o split. La fase de split consiste en separar un único estado en dos cuando se considera que por el aspecto temporal de las trazas realmente son estados distintos. Durante esta fase pueden

realizarse mezcla de estados solo si no hay ninguna separación pendiente y si la mezcla no cancela una separación anterior. El algoritmo finaliza cuando no es posible realizar más operaciones de mezcla o separación.

3.2. Métodos alternativos

Real Time Automata Learning (RTALearning)
El trabajo [2] consiste en un nuevo algoritmo de Automata Learning de aprendizaje activo para generar DRTAs. El proyecto se inspira en el algoritmo L* de Angluin [3], que fue la primera aportación de la técnica de aprendizaje activo. La diferencia radica en que L* y sus derivados no están pensados para sistemas temporizados.

La idea del artículo es hacer consultas de pertenencia a un oráculo acerca de secuencias de estados temporizadas, y guardar las respuestas en una tabla de observaciones de tiempo real. Con esta tabla se construye un DRTA que, posteriormente, es transformado en lo que denominan un RTA "canónico" que sirve de hipótesis "H". A continuación, se le pregunta al oráculo si el lenguaje reconocido por "H" es equivalente al lenguaje del sistema que se está aprendiendo. Si los lenguajes son equivalentes, el oráculo devolverá una respuesta afirmativa, en caso contrario, devolverá un contraejemplo que será utilizado por el algoritmo para corregir el autómata. Este proceso se repite hasta que el oráculo devuelve una respuesta positiva acerca de la equivalencia.

Multi-mode hybrid Automaton Model (MOHA)
En el artículo [15] se propone un framework híbrido para aprender dinámicas de comportamiento discretas y continuas a partir de datos de conducción reales. La idea consiste en discretizar las variables a un nivel de grano grueso con el fin de distinguir patrones de conducción. Dichos patrones se consiguen particionando un modelo (autómata temporizado) en grupos de estados.

El primer paso de la técnica propuesta consiste en dar formato a los datos de entrada, realizando una discretización de las series temporales del dataset mediante agrupación con el conocido algoritmo de clasificación K-means. Una vez que se ha particionado el espacio de las series temporales, se transforman en cadenas de tiempo con el formato "(evento, tiempo)". De esta manera, las series temporales se clasifican según los eventos más significativos que marcan la diferencia entre un grupo (clúster) y otro. Posteriormente, se aplica el algoritmo RTI+ [25] ya antes mencionado, para construir un modelo que describa comportamientos discretos. Utilizando como base este modelo de eventos discretos, se

realiza una extracción de patrones de secuencias de estados comunes y se agrupan mediante técnicas de clasificación para identificar modos de comportamiento. Finalmente, se realiza una asociación entre las variables observables (datos temporales) y las "variables latentes" (patrones que se corresponden a secuencias de estados en el modelo).

Genetic Programming for Timed Automata (GenProgTA)
En la propuesta [21] se presenta GenProgTA, un algoritmo basado en programación genética que aprende un autómata temporizado determinista (DTA). En cada generación, puede realizarse una operación aleatoria de entre tres posibles:

- Mutación: en esta operación hay varias posibilidades, puede ocurrir una mezcla o separación de estados, un reset de un reloj o se puede añadir un nuevo estado.
- Cruce: se intercambian fragmentos entre dos autómatas de la misma población.
- Cruce de poblaciones: se produce la operación de cruce pero entre autómatas de distintas poblaciones.

Las operaciones y la generación de poblaciones de autómatas no están basadas en la observación ya que simplemente se introducen parámetros como los eventos (acciones), el número de relojes y los límites temporales de los mismos. El hecho de que el autómata sea aleatorio puede introducir inconsistencias. Para evaluar cada individuo se tienen en cuenta criterios como el tamaño del autómata, el determinismo o la precisión del modelo con respecto a unas pruebas en las que se comprueba la exactitud del autómata con respecto a trazas válidas e inválidas.

Este algoritmo presenta numerosos parámetros de control como la probabilidad de cada operación, el tamaño de la población y la ponderación de cada criterio de evaluación. Además, requiere información a priori acerca de las propiedades temporales del sistema, es decir, el número de relojes; y unas restricciones temporales para establecer el intervalo en el que puede ejecutarse el sistema. Por tanto, es necesaria la presencia de usuarios expertos para ajustarlo correctamente y elegir el modelo final más adecuado. Finalmente, por su naturaleza estocástica, pueden encontrarse diferentes óptimos locales.

Learning Timed Automata from Interaction Traces
La aportación del artículo [23] consiste en un método para generar autómatas temporizados a través del aprendizaje no supervisado utilizando trazas

de ejecución, y cuyo objetivo es facilitar el modelado de sistemas hombre-máquina (Human-machine system-HMS) de carga crítica . Los sistemas de carga crítica son aquellos que tienen muy en cuenta el tiempo, y deben soportar grandes fluctuaciones de carga en situaciones de crisis. Por ejemplo, sistemas de alerta temprana de desastres naturales y planificadores de rescate en zonas donde ha sucedido un desastre. Un problema común en la implementación de estos sistemas es la falta de disponibilidad y escalabilidad. Esto se debe a que en el desarrollo se suelen aplicar estilos ágiles en lugar de los basados en modelos. La razón por la que no se aplican métodos basados en modelos es por el esfuerzo que supone su construcción, así como la pobre integración de los mismos con las prácticas de desarrollo de software actuales.

El algoritmo propuesto construye autómatas temporizados que siguen las especificaciones de los utilizados en la herramienta Uppaal, es decir, los procesos se comunican a través de variables compartidas y se sincronizan mediante restricciones de reloj y canales. Las trazas son eventos de entrada-salida extraídos al escuchar los puertos de un sistema.

4. Diseño e implementación

Esta sección describe el proceso de diseño e implementación de todos los componentes que conforman la herramienta *LearnTA*.

La Figura 3 muestra una visión general del ecosistema de *LearnTA*. En primer lugar, a la izquierda en la figura, tenemos el sistema que se va a aprender y que consideramos como una caja negra. El sistema genera periódicamente información sobre su estado actual, esta información es lo que denominamos observaciones. La secuencia de observaciones generadas durante una ejecución se conoce como traza, por tanto, en cada ejecución del sistema se produce una traza que contiene diversas observaciones. En general, las trazas generadas por sistemas reales pueden presentar formatos muy dispares por lo que es necesario definir un formato de entrada para *LearnTA* y transformar las trazas que producen los sistemas a éste. Dicha transformación supondrá en muchos casos la construcción de un programa "ad hoc". El formato de entrada se describirá en detalle en la Sección 4.1 .

Una vez que las trazas tienen el formato adecuado, pueden ser leídas por *LearnTA*, que tiene dos módulos claramente diferenciados. Por un lado está el algoritmo de aprendizaje junto con todos sus componentes y, por otro, está el verificador. Con respecto al algoritmo de aprendizaje, cuando se introducen las trazas que se quieren aprender, lo primero

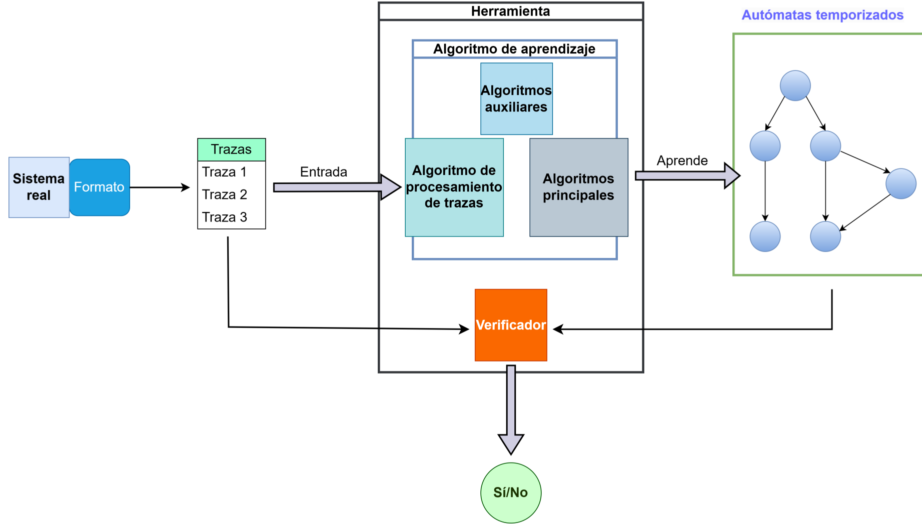


Figura 3: Arquitectura de *LearnTA*

que ocurre es una fase de preprocesado en el que las trazas son compactadas eliminando la información redundante. Concretamente se eliminan las observaciones que no muestren ningún cambio en el sistema. A continuación, comienza el aprendizaje de las trazas que generará un autómata temporizado que refleja el comportamiento de sistema original. La explicación del formato de los autómatas que genera *LearnTA* se encuentra en la Sección 4.2

Finalmente, se encuentra el verificador, que será el responsable de comprobar si el comportamiento observado del sistema real es aceptado por el modelo (autómata aprendido). El verificador acepta como entrada trazas en el mismo formato que acepta el algoritmo de aprendizaje de *LearnTA*, y se encarga de preprocesarlas y comprobar si son aceptadas (válidas) por el modelo. Gracias al verificador se pueden realizar dos tipos de comprobaciones. Por un lado, se pueden introducir trazas del sistema real y comprobar si el autómata es capaz de reconocerlas. Por otro lado, se pueden introducir trazas inválidas para comprobar si el modelo es capaz de identificarlas como erróneas.

Los siguientes secciones describen en profundidad cada uno de los componentes descritos en la arquitectura general. Para simplificar la explicación, se ha decidido llamar a las **location**s **estado**s y se seguirán las simplificaciones explicadas en la Sección 2.1.

La descripción de las diferentes tecnologías empleadas en el desarrollo de este proyecto pueden encontrarse en el Anexo A de este documento.

4.1. Formato de las trazas

Una traza se compone de una serie de observaciones que han ocurrido secuencialmente en el tiempo. Cada observación es un registro del estado del sistema en un momento concreto. La información contenida en cada observación es la siguiente:

- **Time:** Recoge el instante de tiempo en el que se produce la observación. Para normalizar el formato de las trazas se supone que las trazas se leen desde el inicio de la ejecución con tiempo 0.
- **Event:** Campo que recoge si se ha producido algún evento en el sistema. Si no se ha producido ninguno, su valor por defecto es la cadena vacía.
- **Variables:** Campo en el que se guardan el estado de las variables discretas del sistema que se está observando.

El algoritmo acepta trazas en formato JSON que cumplan con la estructura descrita. En cada archivo o cadena en formato JSON las trazas deben de estar contenidas en una lista. Cada traza viene dada a su vez como una lista llamada “observations”, y todos los objetos anidados en ella se corresponden con las observaciones (registros) recogidas. Finalmente, cada una de las observaciones presenta el formato antes explicado, capturando la información mínimamente necesaria para poder construir un autómata temporizado que simule el comportamiento del sistema que se quiere replicar.

La Figura 4 muestra un conjunto de trazas con el formato descrito que captura el comportamiento de una lámpara como el que modela el autómata de la Figura 2.

4.2. Autómatas del algoritmo

El algoritmo de aprendizaje implementado por *LearnTA* produce autómatas del tipo DRTA (autómata de tiempo real determinista - deterministic real time automata). La estructura de estos automátas es la siguiente:

Estados

Cada estado del autómata es la representación de cómo se encontraba el sistema en el momento de la observación. Cada estado puede contener diversas variables que reflejan el sistema real en ese momento. Nos

```json
[ {
  "observations" : [ {
    "time" : 0,
    "event" : "",
    "variables" : [ "off" ]
  }, {
    "time" : 1,
    "event" : "press?",
    "variables" : [ "tenue" ]
  }, {
    "time" : 2,
    "event" : "",
    "variables" : [ "tenue" ]
  }, {
    "time" : 8,
    "event" : "press?",
    "variables" : [ "off" ]
  } ]
}, {
  "observations" : [ {
    "time" : 0,
    "event" : "",
    "variables" : [ "off" ]
  }, {
    "time" : 1,
    "event" : "press?",
    "variables" : [ "tenue" ]
  }, {
    "time" : 2,
    "event" : "",
    "variables" : [ "tenue" ]
  }, {
    "time" : 3,
    "event" : "press?",
    "variables" : [ "brillante" ]
  }, {
    "time" : 4,
    "event" : "",
    "variables" : [ "brillante" ]
  } ]
} ]
```

Figura 4: Ejemplo de formato de las trazas

referiremos a estas variables como variables de estado. Dichas variables proceden del campo variables de las observaciones de las trazas.

Transiciones
Una transición (el paso de un estado a otro) ocurre solo si se dan una serie de condiciones específicas, y queda reflejada como una flecha que parte de un estado hacia el otro. Cada transición contiene una guarda (dos números entre corchetes y separados por una coma) que indica el rango de instantes en que es posible la transición. También puede contener un evento que ha de suceder para poder transitar. De esta forma, las condiciones ligadas a cada transición pueden obligar a que ocurra un evento dentro del intervalo de tiempo de la guarda. Alternativamente, si no contiene ningún evento, la transición puede ocurrir cuando cambian las variables del sistema y caen dentro del intervalo temporal indicado por la guarda.

El algoritmo de aprendizaje asume que el sistema real del que se está aprendiendo es determinista en el siguiente sentido:

- No puede haber dos transiciones por paso de tiempo (sin evento) partiendo del mismo estado.
- No pueden existir dos transiciones que compartan evento, tengan el mismo estado de origen y cuyas guardas de tiempo se solapen.

La Figura 5 muestra el autómata aprendido a partir de las trazas de la lámpara de la Figura 4. El estado inicial 0 con variable de estado "off" (cuyo contorno es rojo) corresponde a la primera observación de cada una de las trazas. De este estado inicial se puede transitar al estado 1 con variable "tenue" si ocurre el evento "press?" tras pasar entre 2 a 6 segundos desde que se inició el sistema.

4.3. Algoritmo de aprendizaje

En el Algoritmo 1 se puede observar una visión general del algoritmo de aprendizaje principal. Dicho algoritmo acepta un conjunto de trazas $\mathcal{T}$ y devuelve un autómata $\mathcal{A}$, fruto del aprendizaje de la información contenida en las trazas de entrada.

El algoritmo de aprendizaje comienza con el preprocesado de las trazas utilizando el algoritmo `CompressTraces`, donde se analizan para eliminar información redundante con el fin de compactarlas. Posteriormente, se realizan dos fases de aprendizaje en las que se construye el autómata temporizado a partir de las trazas tratadas previamente. Las siguientes secciones están dedicadas a describir cada una de las partes del algoritmo de aprendizaje principal.

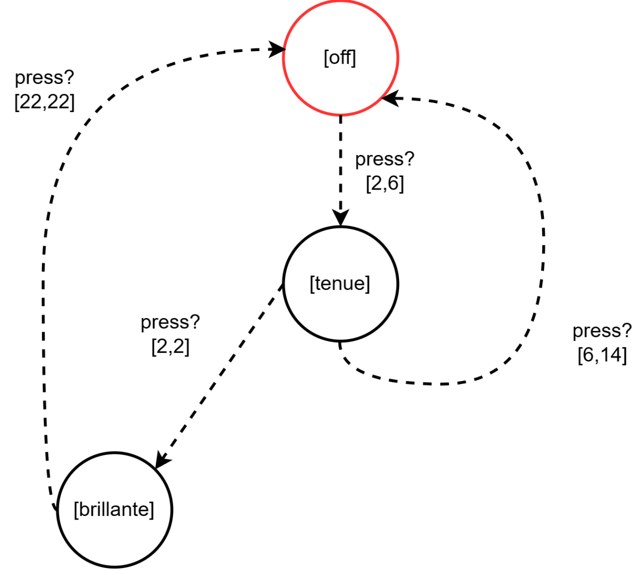


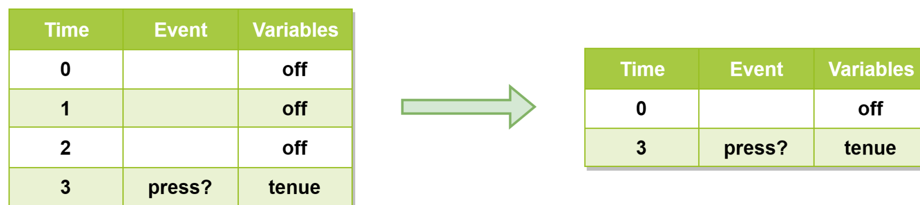
Figura 5: Ejemplo de autómata temporizado resultante

Algoritmo 1: Learning

1 **Algorithm** `Learning`($\downarrow \mathcal{T} = \{t_1, t_2 ... t_n\}, \uparrow \mathcal{A}$):
2 $\mathcal{T}' \leftarrow CompressTraces(\mathcal{T})$;
3 $\mathcal{A}' \leftarrow Phase1(\mathcal{T}')$;
4 $\mathcal{A} \leftarrow Phase2(\mathcal{A}')$;
5 **return** $\mathcal{A}$

Algoritmo `CompressTraces`
Antes de realizar el proceso de construcción del autómata, se realiza un preprocesado de las trazas para reducir su tamaño. Esta proceso se corresponde con la función `CompressTraces`.

La reducción se produce eliminando aquellas observaciones que se consideran redundantes porque solo pasa el tiempo y no hay cambios en el sistema. Por ejemplo, en una traza que contenga observaciones que ocurren cada 5 segundos, puede haber intervalos de tiempo en las que no se produzca ningún evento ni tampoco un cambio en las variables de estado



Time	Event	Variables
0		off
1		off
2		off
3	press?	tenue

Time	Event	Variables
0		off
3	press?	tenue

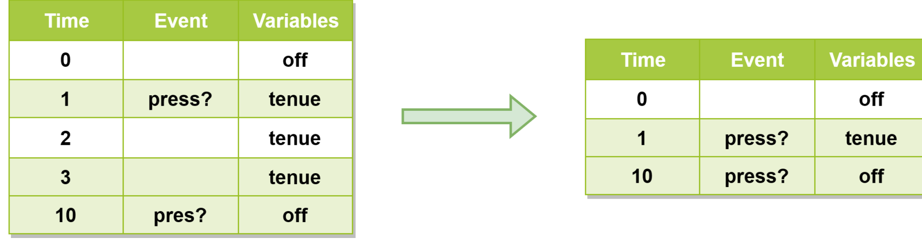
Figura 6: Compresión de trazas caso 1

del sistema. Por tanto, estos conjuntos de observaciones redundantes pueden compactarse en una sola que abarque todo el tiempo de este período.

Hay dos casos en los que se producen esta fusión de observaciones:

1. Si hay observaciones consecutivas en las que no se ha producido ningún evento y sus variables del sistema son idénticas, entonces el sistema se encuentran en el "mismo estado". Por tanto, se fusionan en una sola observación representada por la primera observación de ese conjunto. La Figura 6 ilustra cómo se produce la compresión en este caso. Las observaciones 0, 1 y 2 solo se diferencian en el paso del tiempo, por tanto pueden considerarse que el sistema está en el mismo estado y por tanto pueden unirse en una sola. Dejando la primera observación y eliminando las otras dos equivalentes queda como resultado una traza que contiene la misma información pero que es más compacta.
2. Si se produce un evento y posteriormente todas las observaciones no recogen ningún evento y poseen las mismas variables que la observación con el evento, se consideran dentro del mismo estado. En este caso, solo se guarda la que contiene el evento. En la Figura 7 se muestra un ejemplo de cómo se produce la compresión en este caso. En la observación 1 se produce el evento "press?" y la lámpara en ese momento pasa a estar brillando con una luz tenue (variable tenue). Tras eso, las observaciones 2 y 3 no muestran ninguna diferencia más allá del transcurso del tiempo con respecto a la observación 1. Por este motivo, se consideran redundantes y en la compresión solo permanece la observación 1 (de las tres que eran equivalentes). Así, queda una traza con una información idéntica a la anterior pero con menos redundancia.

Se podría decir que tras este proceso cada observación de cada traza se corresponde con una transición entre estados en el futuro autómata.



Time	Event	Variables
0		off
1	press?	tenue
2		tenue
3		tenue
10	pres?	off

Time	Event	Variables
0		off
1	press?	tenue
10	press?	off

Figura 7: Compresión de trazas caso 2

Algoritmo 2: Phase1

1 **Algorithm** `Phase1`($\downarrow \mathcal{T}, \uparrow \mathcal{A}$):
2 $\quad \mathcal{A} \leftarrow InitAutomata()$;
3 $\quad$ **for** $t_i \leftarrow \mathcal{T}$ **do**
4 $\quad\quad \mathcal{A} \leftarrow ProcessTrace(t_i, \mathcal{A})$;
5 $\quad$ **return** $\mathcal{A}$

Fase 1

Esta primera fase, mostrada en el Algoritmo 2, comienza con la construcción de un autómata vacío para posteriormente procesar secuencialmente cada una de las trazas compactadas de $\mathcal{T}$ con el fin de construir una primera aproximación del autómata.

El método `ProcessTrace`, cuyo funcionamiento está ilustrado en el Algoritmo 3, toma como entrada una traza $t \in \mathcal{T}$ y el autómata en construcción ($\mathcal{A}$) para expandirlo a partir de la información que se encuentra en las observaciones de la traza.

Este método analiza las observaciones, *obs*, de una traza en orden secuencial. Se asume que la primera observación coincide con el estado inicial del sistema. Por tanto, todas las trazas tienen que tener una primera observación "idéntica". La única excepción se produce cuando en la primera observación ocurre un evento del sistema. En este caso, cuando no hay trazas con una observación inicial sin evento, el algoritmo añade un estado inicial "desconocido". Esto es, se considera que se ha producido una transición de un estado inicial no conocido a un nuevo estado que posee las variables de la observación. La Figura 8 clarifica la situación descrita. En este caso, al no conocerse las variables del sistema en el estado inicial, se usan las mismas que el siguiente estado.

En el algoritmo `ProcessTrace`, la variable q_o siempre representa el último estado visitado en el autómata. Cuando se comienza a procesar

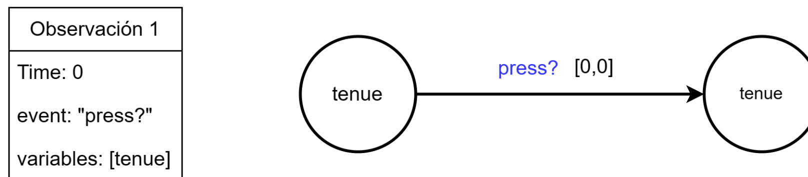


Figura 8: Caso observación inicial con evento

una traza, hay dos motivos por los que el valor de q_o puede no estar inicializado (su valor es ϵ):

1. El autómata no tiene estados, en otras palabras, ha comenzado a procesarse la primera traza. Por tanto, se crea el primer estado del autómata a partir de los datos de la primera observación utilizando la función Expand (descrita más adelante), y se actualiza q_o.
2. El autómata tiene estados creados al procesar trazas anteriores, pero se está procesando la primera observación de la traza actual. En este caso, q_o representará el estado inicial del autómata y se actualizará el autómata con la información de la primera observación *obs* utilizando la función Update.

Si el valor de q_o no es ϵ, entonces es que ya se han procesado algunas observaciones de la traza y se está recorriendo el autómata. En primer lugar se busca, con el método `searchTarget`, si existe un estado destino q_t al que se pueda llegar directamente mediante una transición desde q_o a partir del evento contenido en la observación actual. Si existe esta transición, se actualiza la información que contienen tanto la transición como q_t con la nueva observación usando la Función Update. En otro caso, si no existe la transición, se procede a realizar un algoritmo *novedoso* denominado `FastMerge`, que explicaremos a continuación.

Algoritmo FastMerge

`FastMerge` es una operación cuya función es comparar un segmento de la traza que se está analizando con estados y transiciones existentes del autómata en construcción. Esta comparación se realiza para encontrar secuencias de la traza que pueden ser equivalentes a secciones del autómata ya existentes y que, por tanto, pueden ser reutilizadas para no crear nuevos estados y transiciones. Se ha decidido no incluir el pseudocódigo del

Algoritmo 3: ProcessTrace

```
1  Algorithm ProcessTrace(↓ t, ↓ 𝒜, ↑ 𝒜):
2      obs ← t[0] ;                          /* Observación actual de la traza */
3      q_o ← ϵ ;                                   /* Último estado visitado */
4      q_t ← ϵ ;   /* Estado al que se puede transitar a partir de q_o */
5      if 𝒜.IsEmpty() then
6          q_o ← 𝒜.Expand(obs, q_o) ;
7      else
8          q_o ← 𝒜.InitialState() ;
9          q_o ← 𝒜.Update(obs, q_o, q_t) ;
10     i ← 1 ;
11     while i < |t| do
12         obs ← t[i] ;
13         q_t ← 𝒜.SearchTarget(obs, q_o) ;
14         if q_t ≠ ϵ then
15             q_o ← 𝒜.Update(obs, q_o, q_t) ;
16         else
17             q_t ← 𝒜.FastMerge(t[i, i + k], q_o) ;
18             if q_t ≠ ϵ then
19                 q_o ← q_t ;
20                 i ← i + (k − 1) ;
21             else
22                 q_o ← 𝒜.Expand(obs, q_o) ;
23         i ← i + 1 ;
24     return 𝒜
```

Función Update

```
1  Fun Update(↓ obs, ↓ q_o, ↓ q_t, ↑ q):
2      q ← 𝒜.UpdateState(q_t, obs) ;
3      𝒜.UpdateTransition(q_o, q, obs) ;
4      return q
```

Función Expand

```
1  Fun Expand(↓ obs, ↓ q_o, ↑ q):
2      q ← 𝒜.NewState(obs) ;
3      if q_o ≠ ϵ then
4          𝒜.NewTransition(q_o, q, obs) ;
5      return q
```

algoritmo porque es muy engorroso y no tendría una aportación relevante en la explicación.

El proceso de comparación se basa en el algoritmo K-Tail 2.2. De esta manera, se extrae el segmento de la traza compuesto por la observación que se está analizando y las K de observaciones que suceden a la observación actual que se compara con los diferentes fragmentos del autómata formado por estados y transiciones consecutivos hasta encontrar, si existe, uno equivalente. Para comprobar la equivalencia, el segmento de las observaciones extraído se transforma en estados y transiciones que llamamos auxiliares. El evento y el tiempo de una observación se convierten en una transición, mientras que las variables se transforman en un estado en el autómata al que se llega a través de la transición. De esta manera, los estados y transiciones de ambas secuencias, la auxiliar y la del autómata, se comparan uno a uno secuencialmente. El criterio de comparación es el siguiente:

- **Estados:** Dos estados son equivalentes si las variables de estado que contienen son idénticas.
- **Transiciones:** Dos transiciones son equivalentes si el evento que poseen es el mismo.

Si las dos secuencias resultan ser equivalentes, se crea una transición desde el último estado visitado del autómata (q_o) hasta el estado que es equivalente a la observación actual. Además, se actualiza el segmento del autómata con la información del fragmento de la traza que es equivalente. Finalmente, el estado actual del autómata (q_o) se actualiza al último estado construido (es el que devuelve la operación al final de su ejecución).

Cabe destacar que, en esta operación, se ha decidido no tomar en cuenta las guardas de las transiciones (factor temporal) porque se ha considerado que establecer criterios muy restrictivos en la fase inicial de la construcción del autómata podría dar lugar a autómatas demasiado grandes.

La Figura 9 muestra el autómata resultante del procesamiento de las cinco primeras observaciones de la traza de la izquierda (cuadro azul). La observación que está siendo tratada actualmente es la primera incluida en el rectángulo rojo apuntada por una flecha. El último estado visitado en el autómata (q_o en el Algoritmo 3) es el que tiene una flecha etiquetada con un 1. En este caso, el autómata no contiene ninguna transición desde q_o a un estado equivalente al de la observación actual. Por ello, en la fase 1 del Algoritmo 1 se procede a ejecutar el método `FastMerge` para buscar segmentos equivalentes, en este ejemplo, de longitud $K = 2$, entre el autómata y la traza partiendo de la observación actual. De acuerdo con el criterio de equivalencia, `FastMerge` determina que son equivalentes el

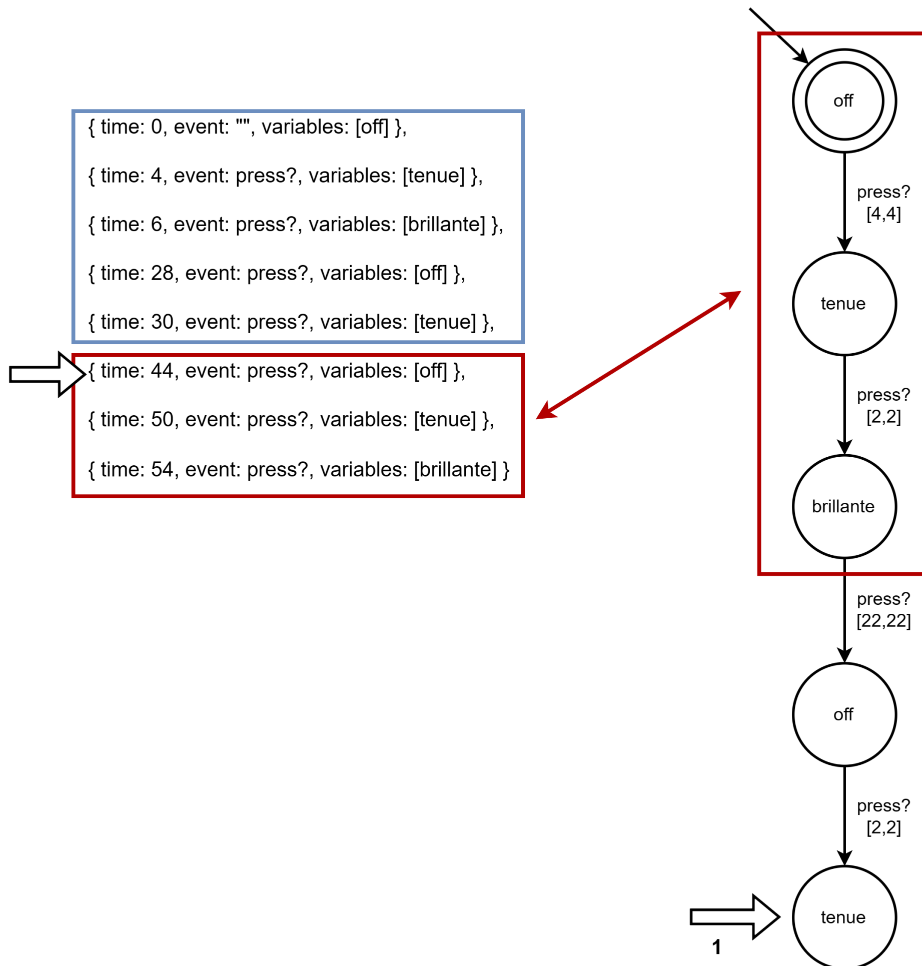


Figura 9: Ejemplo `FastMerge` parte 1

fragmento de observaciones de la traza que están dentro del rectángulo rojo y los estados y transiciones del autómata dentro del rectángulo rojo. La Figura 10 muestra los cambios en el autómata después de la aplicación de `FastMerge`. Por un lado, se genera una transición desde el último estado visitado (q_o) hacia el primer estado equivalente a la observación actual. Por otro lado, se actualizan, si es necesario, las guardas de las transiciones con el mínimo intervalo que contenga a los intervalos que se están mezclando (del autómata y de las observaciones). Por último, se actualiza q_o al k-ésimo estado de la secuencia equivalente en el autómata. En la Figura 10 este estado es el apuntado por la flecha etiquetada con 2.

Figura 10: Ejemplo `FastMerge` parte 2

Volviendo a `ProcessTrace`, si resulta que `FastMerge` no ha podido encontrar ningún fragmento del autómata que pueda reutilizarse. El autómata se expande con la función `Expand` creando un nuevo estado, una transición que lleve a éste a partir de q_o y se actualiza q_o con el nuevo estado creado.

Fase 2

El Algoritmo 4 muestra los pasos que se llevan a cabo en la fase 2, que consisten principalmente en buscar de forma iterativa estados equivalentes para fusionar hasta que no quedan más candidatos. Este proceso de unión de estados se realiza con el método `LookForMerge` 5 explicado en detalle posteriormente en la sección 17. Después de mezclar los estados equivalentes con `LookForMerge`, puede haber varias transiciones que tienen el mismo origen o destino, mismo evento y con guardas que se solapan. Normalmente este indeterminismo se subsana de manera natural en uniones posteriores de otros estados, ya que suelen ser transiciones a estados que

Algoritmo 4: Phase2

```
1  Algorithm Phase2(↕ 𝒜):
2  |  merged ← True ;
3  |  indet ← False ;
4  |  fixed ← True ;
5  |  do
6  |  |  while merged do
7  |  |  |  merged ← 𝒜.LookForMerge() ;
8  |  |  indet ← 𝒜.CheckIndet() ;
9  |  |  if indet then
10 |  |  |  fixed ← 𝒜.FixIndet() ;
11 |  while  fixed && indet;
12 |  if !fixed || indet then
13 |  |  Error : Invalid(𝒜) ;
14 |  merged ← True ;
15 |  while merged do
16 |  |  merged ← 𝒜.MergeFinalStates() ;
17 |  return 𝒜
```

son también equivalentes. Sin embargo, en algunas ocasiones no desaparece. Por ello, después del proceso de mezcla de estados se hace uso de la función `CheckIndet` (Algoritmo 4, línea 8) para comprobar si el autómata resultante es determinista (sección 4.2). Si no lo es, `FixIndet` (Algoritmo 4, línea 10) busca corregir el no determinismo intentando unir, si es posible, las transiciones en conflicto y los correspondientes estados origen y destino. Los estados solo pueden unirse si poseen las mismos valores de las variables.

Si se consigue solucionar el indeterminismo, se vuelve al principio de la fase 2 para buscar si es posible unir más estados tras la transformación. Si no se ha podido solucionar el indeterminismo, quiere decir que el sistema que se está aprendiendo no cumple las propiedades de determinismo definidas 4.2 y se lanza un error.

Finalmente, el autómata resultante puede tener múltiples estados finales que son equivalentes, pero que no se fusionan durante la fase 2 al no tener secuencias futuras de longitud K. El método `MergeFinalStates` (Algoritmo 4, línea 16) busca fusionar estos estados siempre que sean equivalentes. En este caso, se considera que dos estados finales son equivalentes si las variables de estado tienen los mismos valores y se ha llegado a ellos mediante una transición disparada por el mismo evento. Cabe des-

tacar que en esta operación no se comprueba el tiempo en las transiciones que se comparan.

Operación LookForMerge
El método `LookForMerge`, que se muestra en el Algoritmo 5, es el encargado de buscar estados equivalentes para posteriormente unirlos con la operación $Merge$ que se detallará posteriormente. En primer lugar, se guardan todos los estados del autómata en una lista qs para después realizar comprobaciones sobre cada uno de los estados q_i de qs. Para cada estado q_i se calculan todos sus K-futuros (secuencias de longitud K de estados y transiciones accesibles a partir del estado actual, a la hora de contar la longitud solo se tienen en cuenta los estados), que llamamos f_i. A continuación, el método `FindEq` (Algoritmo 5, línea 5) busca un estado q_{eq} en el autómata que sea equivalente a q_i. En este método, la definición de equivalencia es distinta a la del algoritmo `FastMerge`. Concretamente, para que dos estados q_i y q_{eq} sean equivalentes han de cumplirse dos requisitos: 1) en ambos estados las variables deben tener los mismos valores, y 2) las secuencias K-futuras de ambos estados tienen que ser equivalentes. Cuando se comparan dos secuencias K-futuras, se hace secuencialmente por pares de estados o transiciones respetando el orden de cada par de elementos. Para la comparación se han establecido una serie de criterios en el que el tiempo toma un papel fundamental. Para evitar una sobreaproximación excesiva del autómata a partir de este momento se tienen en cuenta las restricciones temporales de las guardas del modo siguiente:

1. Si ambos estados tienen el mismo número de posibles secuencias K-futuras, cada una de las secuencias de un estado tiene que tener una equivalente en las secuencias del otro estado, esto es:
   a) Los pares de estados han de poseer los mismos valores de variables.
   b) Las transiciones comparadas deben compartir evento y sus guardas de tiempo *han de solaparse al menos parcialmente*.
2. Si un estado tiene más secuencias K-futuras que otro, todas las secuencias del estado con el menor número de secuencias K-futuras tienen que tener un equivalente en las secuencias del estado con el mayor número. En este caso los criterios de equivalencia son:
   a) Los estados han de poseer las mismos valores de variables.
   b) Las transiciones que se comparen deben compartir evento y sus guardas de tiempo *han de solaparse totalmente*, es decir, ser iguales o que una esté totalmente contenida en la guarda más amplia.

En la Figura 11 se muestra un ejemplo de dos estados que son equivalentes. En este caso, el estado “off(0)” presenta dos secuencias K-futuras

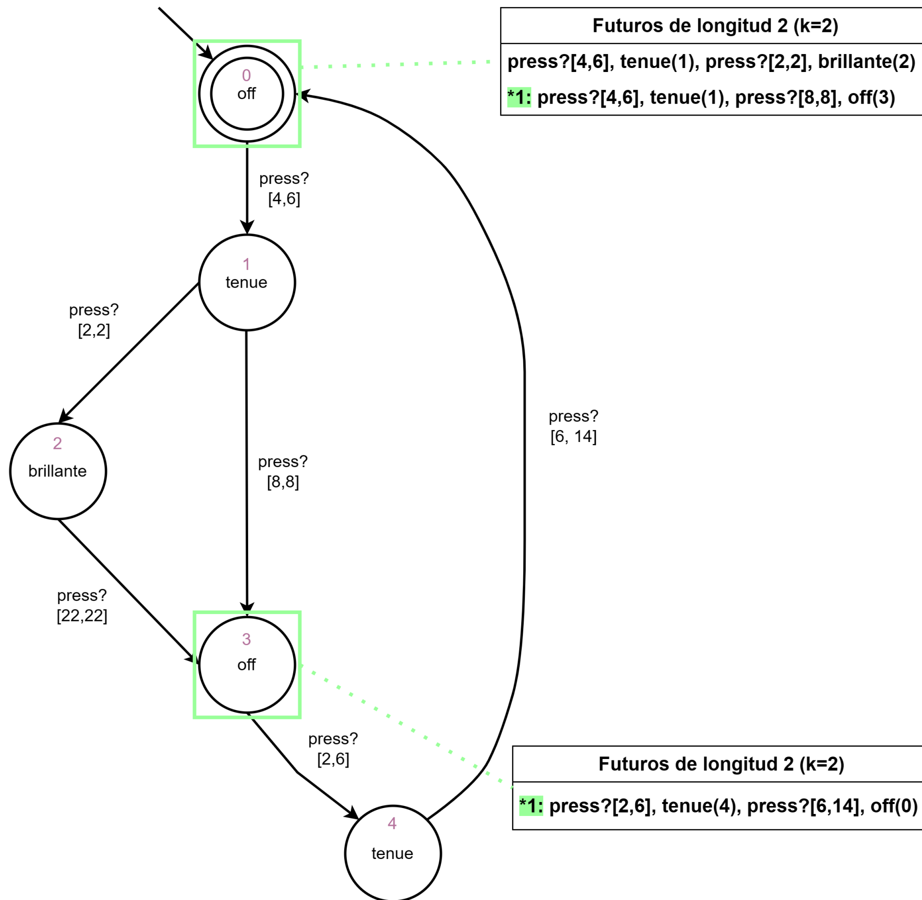


Figura 11: Ejemplo de estados equivalentes

de longitud K=2, mientras que “off(3)” solo tiene una. Por tanto, se aplica el segundo criterio de equivalencia y resulta que la única secuencia de “off(3)” tiene un equivalente en las secuencias del otro estado (las secuencias están marcadas como *1 en la figura).

Si para un estado del autómata q_i, el método `FindEq` encuentra un estado equivalente q_{eq}, los fusiona en un único estado haciendo uso de la operación `Merge`. Finalmente, devuelve si ha sido posible o no realizar una unión de estados.

Operación Merge

Esta operación mezcla dos estados equivalentes en uno solo. El proceso llevado a cabo es el siguiente:

Algoritmo 5: LookForMerge

```
1  Algorithm LookForMerge(↑ boolean):
2      qs ← A.States() ;
3      for q_i ← qs do
4          f_i ← A.KFutures(q_i) ;
5          q_eq ← A.FindEq(q_i, f_i) ;
6          merged ← False ;
7          if q_eq ≠ ϵ then
8              merged ← A.merge(q_i, q_eq) ;
9              if merged then
10                 return True
11         return False
```

1. Se conserva uno de los dos estados y el otro se elimina. En la implementación todos los estados poseen un número identificador para comprobar su antigüedad. En la mezcla se conserva el estado más antiguo.
2. Las transiciones en las está el estado eliminado pasan a ser parte del estado que se ha conservado.
3. Se unen las posibles transiciones que tienen el mismo evento, origen y destino y con guardas que se solapan.

La mezcla de estados queda reflejada en la Figura 12, donde se marcan las transiciones para identificarlas antes y después de la fusión.

5. Pruebas y evaluación del algoritmo

Esta sección presenta las pruebas realizadas sobre el algoritmo de aprendizaje descrito en la Sección 4. El objetivo de las pruebas es contestar a las siguientes cuestiones:

1. **¿Qué factores influyen en el tiempo de ejecución del algoritmo y calidad del modelo resultante?**
2. **¿Qué relevancia tienen cada una de las fases del algoritmo?**
3. **¿Cuál es el rendimiento general del algoritmo?**
4. **¿Cómo se comporta el algoritmo con trazas de sistemas en los que el tiempo no es tan relevante?**

Las pruebas se han organizado en cuatro bloques:

1. Pruebas con trazas sintéticas generadas por modelos UPPAAL.

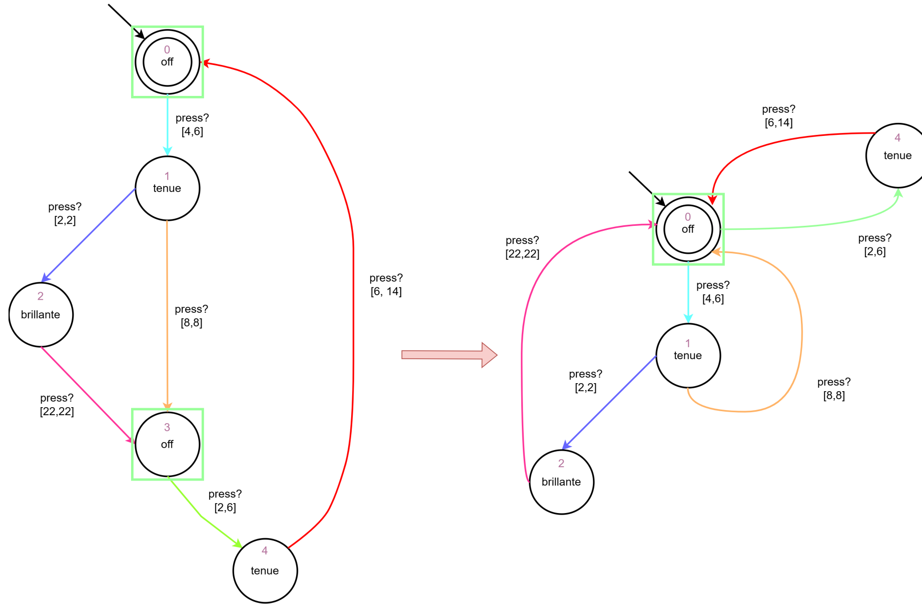


Figura 12: Operación Merge

2. Pruebas con autómatas aleatorios.
3. Comparación con el algoritmo TAG [5].
4. Evaluación del algoritmo con un sistema real.

La cuestiones 1 y 2 serán tratadas en el bloque 1 en las que se utilizan sistemas de Uppaal, ya que al proceder las trazas de modelos conocidos se puede razonar mejor acerca de los resultados obtenidos. La tercera cuestión se discutirá en los bloques 2 y 3, en los que se pondrá a prueba tanto el algoritmo en solitario (bloque 2) como comparándolo con otro algoritmo de la literatura (bloque 3). La cuarta pregunta será abordada en el cuarto bloque, en el que se utilizará un sistema real que, aunque tiene un cierto orden temporal, éste no es tan relevante en su comportamiento.

5.1. Pruebas con trazas sintéticas generadas por modelos Uppaal

En este bloque se realiza una primera batería de pruebas para comprobar el rendimiento general del algoritmo. Los sistemas a aprender son modelos de sistemas en Uppaal. Como partimos de un sistema modelado como un autómata temporizado conocido, podemos determinar si los autómatas resultantes del aprendizaje corresponden al comportamiento

	Sistema 1	**Sistema 2**	**Sistema 3**	**Sistema 4**
Número de acciones	1	1	3	4
Número de variables de estado	3	3	4	2

Tabla 1: Características de las trazas de los sistemas de UPPAAL

Número de trazas	10, 20, **50**, 80, 100
K	**2**, 3, 4, 5
Tamaño de traza (observaciones)	25, 50, **100**, 200, 400

Tabla 2: Configuración de parámetros para las pruebas con UPPAAL

de los sistemas que se quieren aprender. En primer lugar, se han modificado los ejemplos de sistemas temporizados que ofrece la herramienta UPPAAL para que los modelos produzcan trazas acordes con las utilizadas en este proyecto, añadiendo variables cuando ha sido necesario. Para obtener las trazas de estos modelos en un formato adecuado, se ha construido un programa en Java que interactúa con el motor de simulación de UPPAAL con el fin de obtener trazas de ejecución y posteriormente adaptarlas al formato del algoritmo 4.1. Estas trazas serán utilizadas para el aprendizaje de los autómatas y la validación de los mismos. La Tabla 1 muestra las características de las trazas de los 4 sistemas UPPAAL que se han aprendido con algoritmo desarrollado en el proyecto.

Se han elegido como factores de estudio el número de trazas, el tamaño de las trazas y el valor del parámetro k. La Tabla 2 muestra las configuraciones utilizadas en las pruebas. Cuando se ha estudiado un factor, los otros han sido fijados en un valor específico (marcado en negrita). Por ejemplo, si se estudia el efecto de diferentes valores del parámetro K, entonces el número de trazas es 50 y todas tienen un tamaño de 100 observaciones.

Para cada sistema se han generado 30 conjuntos de trazas positivas (que son aceptadas por el sistema original) por cada una de las configuraciones. Retomando el ejemplo anterior, se generan 30 conjuntos de trazas para cada valor de K con los demás parámetros fijados. De cada conjunto de trazas se han utilizado un 70 % para aprender el sistema y un 30 % de trazas se han reservado para la validación. Las métricas elegidas son:

1. **Media de acierto de los modelos (Mean):**

$$Mean = \frac{\sum_{n=1}^{TS} \frac{VT_n}{TT_n}}{TS}$$

$$VT = Valid\ Traces,\ TT = Total\ Traces,\ TS = Total\ Sets$$

2. **Desviación estándar de los modelos (std):**

$$std = \sqrt{\frac{\sum_{n=1}^{TS}(M_n - M)^2}{TS}}$$

$$M = Mean,\ TS = Total\ Sets$$

3. **Media de trazas de aprendizaje aceptadas (Mean train):**

$$Meantrain = \frac{\sum_{n=1}^{TS} \frac{ITT_n}{TTT_n}}{TS}$$

$ITT = Identified\ Train\ Traces,\ TTT = Total\ Train\ Traces,\ TS = Total\ Sets$

4. **Media de trazas de validación aceptadas (Mean valid):**

$$Meanvalid = \frac{\sum_{n=1}^{TS} \frac{IVT_n}{TVT_n}}{TS}$$

$IVT = Identified\ Valid\ Traces,\ TVT = Total\ Valid\ Traces,\ TS = Total\ Sets$

5. **Tiempo medio de ejecución (s)**

Tras realizar las pruebas, el número de trazas usadas en el aprendizaje es el factor que tiene una mayor importancia, como se observa en la Tabla 3. A medida que aumenta el número de trazas lo hace también la media de aciertos y disminuye la desviación estándar. En todos los casos los aciertos del conjunto de pruebas es siempre del 100 % mientras que el de validación permanece en un valor alto que en muchas ocasiones roza o alcanza el 100 % conforme aumenta el número de trazas usadas tanto en el aprendizaje como en la validación. El resto de resultados se encuentran en el anexo B, sección B.1.

La Tabla 4 muestra el impacto de cada una de las fases del algoritmo de aprendizaje. La fase 1 presenta una media de aciertos muy alta con una dispersión baja en todos los casos. La fase 2 ayuda a reducir el número de estados y transiciones que, en muchos casos, resultan ser redundantes o equivalentes a otras. En los resultados del sistema 3 se produce un aumento en la media de aciertos y una reducción de la dispersión del modelo resultante entre la fase 1 y la fase 2. Esto se debe al aumento del tamaño de las guardas cuando resultante de la mezcla de estados equivalentes. Es importante recordar que cuando se mezclan estados también se unen transiciones duplicadas (que tienen el mismo origen, destino, evento y guardas que se solapen). Este aumento del intervalo de las guardas permite la aceptación de un mayor número de trazas. Los criterios de mezcla de estados establecidos están orientados a intentar evitar un modelo sobre-aproximado en exceso que tenga un importante impacto negativo en la precisión del mismo.

Trazas	Mean (%)	std (%)	Mean train (%)	Mean valid (%)	Tiempo(s)
10	97,667	4,23	100	92,222	0,002
20	99,167	2,911	100	97,222	0,001
50	99,467	0,884	100	98,222	0,001
80	99,875	0,375	100	99,583	0,001
100	99,733	0,442	100	99,111	0,001

(a) Sistema 1

Trazas	Mean (%)	std (%)	Mean train (%)	Mean valid (%)	Tiempo(s)
10	100	0	100	100	0,005
20	100	0	100	100	0,003
50	100	0	100	100	0,006
80	100	0	100	100	0,009
100	100	0	100	100	0,012

(b) Sistema 2

Trazas	Mean (%)	std (%)	Mean training (%)	Mean valid (%)	Tiempo(s)
10	96,333	7,063	100	87,778	0,003
20	99	2,708	100	96,667	0,001
50	98,867	1,522	100	96,222	0,002
80	99,542	0,883	100	98,472	0,002
100	99,833	0,453	100	99,444	0,003

(c) Sistema 3

Trazas	Mean (%)	std (%)	Mean training (%)	Mean valid (%)	Tiempo(s)
10	99,667	1,795	100	98,889	0,005
20	100	0	100	100	0,003
50	100	0	100	100	0,007
80	100	0	100	100	0,011
100	100	0	100	100	0,013

(d) Sistema 4

Tabla 3: Impacto del número de trazas en el aprendizaje

5.2. Pruebas con autómatas aleatorios

Para este bloque de pruebas se ha construido un generador de autómatas aleatorios. En el generador se pueden configurar diferentes parámetros para controlar la complejidad de los autómatas producidos. En concreto, se puede definir el número de estados y transiciones del autómata, el número de variables de estado, los valores mínimo y máximo de las guardas de tiempo, y el número de acciones distintas. Junto al generador, se ha construido un simulador de ejecuciones para generar trazas de ejecución

Fase	Mean (%)	std (%)	Mean training (%)	Mean valid (%)	Estados	Transiciones
1	99,533	0,991	100	98,444	5	5
2	99,533	0,991	100	98,444	5	5

(a) Sistema 1

Fase	Mean (%)	std (%)	Mean training (%)	Mean valid (%)	Estados	Transiciones
1	100	0	100	100	5,133	7,2
2	100	0	100	100	3	4

(b) Sistema 2

Fase	Mean (%)	std (%)	Mean training (%)	Mean valid (%)	Estados	Transiciones
1	99	1,612	100	96,667	6	7,5
2	99,2	1,327	100	97,333	5	6

(c) Sistema 3

Fase	Mean (%)	std (%)	Mean training (%)	Mean valid (%)	Estados	Transiciones
1	100	0	100	100	7,667	11,8
2	100	0	100	100	3	5

(d) Sistema 4

Tabla 4: Impacto de las fases del algoritmo

a partir de los autómatas aleatorios. El simulador puede generar tanto trazas de ejecución válidas, que reflejan el comportamiento normal del sistema, como trazas "erróneas" que introducen algún comportamiento que no es propio del sistema.

Se han generado 1.000 autómatas aleatorios, como los especificados en la Sección 4.2, que simulan el comportamiento de sistemas reales con distintas complejidades utilizando parámetros pueden estar en los rangos que se observan en la Tabla 5. La elección de los valores concretos de estos parámetros siguen además unas reglas. Por ejemplo, no puede haber más eventos que transiciones ni que el número de transiciones provoque estados aislados. Para cada autómata, se han generado tres conjuntos de trazas, el primer conjunto son trazas usadas para el aprendizaje, el segundo son trazas válidas y el tercer conjunto son trazas erróneas. El conjunto de aprendizaje se utilizará para generar un autómata temporizado, para posteriormente realizar la validación usando los tres conjuntos. Cada conjunto está formado por 750 trazas con una longitud 300 observaciones, aunque la longitud de algunas trazas puede ser menor si se llega a un estado final del autómata y, por tanto, no hay transiciones posibles. Las métricas utilizadas en este conjunto de pruebas son las siguientes:

1. **Precisión (Precision):** Porcentaje de trazas verdaderamente positivas de todas las identificadas como positivas.

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive} = \frac{True\ Positive}{Total\ Predicted\ Positive}$$

2. **Exhaustividad (Recall):** Porcentaje de trazas realmente válidas que el modelo es capaz de identificar.

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative} = \frac{True\ Positive}{Total\ Actual\ Positive}$$

3. **Valor-F (F1-score):** Captura el balance entre la precisión y la exhaustividad.

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall}$$

4. **Exactitud (Accuracy):** Mide el porcentaje de casos en los que el modelo ha reconocido correctamente una traza.

$$Accuracy = \frac{True\ Positive + True\ Negative}{True\ Positive + True\ Negative + False\ Positive + False\ Negative}$$

5. **Media de trazas de aprendizaje aceptadas (Mean train):**

$$Meantrain = \frac{\sum_{n=1}^{TS} \frac{ITT_n}{TTT_n}}{TS}$$

$ITT = Identified\ Train\ Traces,\ TTT = Total\ Train\ Traces,\ TS = Total\ Sets$

6. **Media de trazas de validación aceptadas (Mean valid):**

$$Meanvalid = \frac{\sum_{n=1}^{TS} \frac{IVT_n}{TVT_n}}{TS}$$

$IVT = Identified\ Valid\ Traces,\ TVT = Total\ Valid\ Traces,\ TS = Total\ Sets$

7. **Media de trazas erróneas identificadas (Mean err):**

$$Meanerr = \frac{\sum_{n=1}^{TS} \frac{IET_n}{TET_n}}{TS}$$

$IET = Identified\ Err\ Traces,\ TET = Total\ Err\ Traces,\ TS = Total\ Sets$

8. **Tiempo medio de ejecución (s)**

La Tabla 6 presenta la media de los resultados obtenidos, redondeado a tres decimales, para cada uno de los 1000 autómatas. En la tabla se puede apreciar que todas las métricas presentan un porcentaje cercano o igual al 100 %. Es de especial interés el hecho de que *LearnTA*es capaz de identificar trazas inválidas. Además, el tiempo de ejecución medio se mantiene estable en torno a un segundo.

Número de estados (n_s)	[50,100]
Número de transiciones	[n_s*3,n_s*5]
Número de acciones	[5,50]
Número de variables de estado	[5,50]
Tiempo mínimo de guarda (t_{min})	[2,500]
Tiempo máximo de guarda	$[t_{min}+1, t_{min}+500]$

Tabla 5: Complejidades de los autómatas aleatorios

Prec (%)	Recall (%)	F1-score (%)	Accuracy (%)	Mean training (%)	Mean valid (%)	Mean err (%)	Tiempo(s)
100	99,934	99,967	99,956	100	99,869	100	1,479

Tabla 6: Resultados de las pruebas con autómatas aleatorios

5.3. Comparación con el algoritmo TAG

El siguiente bloque de pruebas está enfocado en comparar nuestra propuesta con el algoritmo TAG que se introdujo en la Sección 3.1. Los autómatas aprendidos por el algoritmo TAG no tienen variables de estado, por tanto, para poder comparar ambos algoritmos se han generado

autómatas sin variables de estado. Además, se ha tenido que adaptar el simulador para que también produzca trazas en el formato de entrada de TAG. Las pruebas se han realizado de una manera similar al bloque anterior. Se han generando 100 autómatas aleatorios que cumplen las reglas definidas en 4.2. En este caso, se ha reducido el número de trazas de cada conjunto a 100 trazas con una longitud de 100 observaciones para cada autómata, y también se han cambiando los parámetros de complejidad de los autómatas a valores que están dentro de los intervalos que se muestran en la Tabla 7. Esta simplificación se debe a que TAG tiene problemas de rendimiento a la hora de procesar una gran cantidad de trazas y generar autómatas de mayor tamaño, llegando a superar la hora de tiempo de ejecución, en algunos casos.

Los resultados de la comparación se muestran en la Tabla 8. El algoritmo propuesto en este proyecto es superior en casi todas las métricas utilizadas y en el tiempo medio de aprendizaje para las mismas entradas. Un aspecto destacado es que a pesar de que el algoritmo propuesto tiene menor número de estados y transiciones, que podría llevar a una sobreaproximación excesiva, presenta mejores métricas que TAG. Podríamos concluir que el algoritmo de este proyecto hace bien la tarea de mezclar estados y transiciones identificando ciclos de comportamientos en la ejecución de un sistema.

Número de estados (n_s)	[4,40]
Número de transiciones (n_t)	$[n_s * 2, n_s * 4]$
Número de acciones	[4,n_t]
Tiempo mínimo de guarda (t_{min})	[2,50]
Tiempo máximo de guarda	$[t_{min} + 1, t_{min} + 50]$

Tabla 7: Complejidades de los autómatas aleatorios en la comparación

Algoritmo	Prec (%)	Recall (%)	F1-score (%)	Accuracy (%)	Tiempo(s)	Estados	Transiciones
LearnTA	99,823	98,695	99,216	99,010	0,671	16,310	49,870
TAG	99,918	84,555	91,18	89,65	62,311	91,74	168,97

Tabla 8: Comparación de los algoritmos

5.4. Evaluación del algoritmo con un sistema real

Para el último bloque, se ha utilizado el algoritmo de aprendizaje con un sistema real. Se trata del algoritmo DASH [7] que se utiliza para la transmisión de vídeo de forma adaptativa según las condiciones de la red. En DASH intervienen dos entidades que son un servidor de vídeo streaming y una aplicación cliente. El comportamiento de este algoritmo no está fuertemente influenciado por el paso del tiempo, sino por las condiciones de la red y los diferentes mensajes que se intercambian la aplicación cliente y el servidor. Aún así, las trazas que se observan contienen información temporal sobre el instante en el que llega cada mensaje. En esta sección, se van a utilizar trazas obtenidas durante la ejecución de varias sesiones de vídeo streaming para generar un autómata que represente el intercambio de mensajes TCP entre el cliente y el servidor.

Para extraer las trazas se ha utilizado un servidor de DASH y vídeos de demostración que están disponibles online [6]. El intercambio de mensajes ha sido capturado con Wireshark, un analizador de paquetes de red. A continuación, se han aislado los paquetes pertenecientes a la comunicación y se ha eliminado el resto. Se han obtenido 92 trazas que corresponden a los fragmentos de la comunicación que se contienen las secuencias de envíos de segmentos de vídeo.

El modelo resultante del aprendizaje está en el anexo B, sección B.2. Dicho modelo presenta una estructura muy similar al comportamiento que se produce en un intercambio de mensajes en una comunicación TCP, en la que claramente hay un proceso de establecimiento de conexión mediante el conocido "three way handshake", posteriormente un envío de paquetes de datos y finalmente un cierre de la conexión mediante paquetes con el flag reset of fin activos. Teniendo en cuenta que DASH está implementado sobre HTTP y que por tanto utiliza TCP, el automata resultante es el esperado a priori. Aunque en el autómata se muestran algunas situaciones anómalas de duplicación de paquetes, se ha comprobado en las trazas que no se trata de ningún error en el aprendizaje sino que las trazas presentan verdaderamente estos comportamientos. Finalmente, usando el verificador se ha comprobado que el autómata que es capaz de reconocer correctamente todas las trazas utilizadas en el aprendizaje.

6. Conclusiones y líneas futuras

En este trabajo se ha implementado la herramienta *LearnTA* que utiliza la técnica de Automata Learning pasivo para extraer autómatas tem-

porizados a partir de observaciones de sistemas reactivos que dependen del tiempo cuya implementación es desconocida.

La estructura de los autómatas propuesta en el trabajo enriquece otras alternativas descritas en el estado del arte, permitiendo la construcción de autómatas más precisos. Concretamente, se han añadido variables en las observaciones que permiten describir de forma más fiel el sistema bajo aprendizaje. Adicionalmente, la distinción entre transiciones con y sin evento permite captar comportamientos más complejos en la evolución de los sistemas que no han sido incorporadas en ninguna aportación vista en el estado del arte.

El algoritmo de aprendizaje incorpora métodos y operaciones que reducen en gran medida el tamaño de los autómatas que se están aprendiendo pero sin llegar a generalizaciones excesivas. La evaluación de los algoritmos desarrollados en el proyecto ha demostrado que *LearnTA* posee una mayor eficiencia y resultados semejantes o mejores, en algunos casos, que otras aportaciones existentes en el estado del arte.

6.1. Trabajos Futuros

El panorama del Automata Learning en el ámbito del aprendizaje de sistemas temporizados es muy prometedor. Hay un gran abanico de posibilidades en lo referente a futuras líneas a explorar. Algunas de las más relevantes serán presentadas a continuación.

Añadir fase de aprendizaje activo
Se contempla añadir una segunda parte que conecte el autómata aprendido a un oráculo que interactúe de manera continua con un sistema. De esta manera el autómata podría podría aprender y refinarse de manera activa mediante la recepción de información en tiempo “quasi” real a la ejecución del sistema bajo aprendizaje.

Construcción de métodos híbridos
Tal y como se ha comentado en el estado del arte, existen algunos trabajos donde se intenta aplicar o incorporar diferentes técnicas para complementar la técnica Automata Learning, como el uso de data mining a los autómatas resultantes para detectar patrones. Sería interesante estudiar qué técnicas pueden tener un impacto positivo y qué resultados se podrían generar.

Diseño e implementación de lógica y verificador
En el ámbito de los métodos formales, es posible definir una lógica que recoja todos los aspectos de los autómatas que se han diseñado. Asimismo, es posible diseñar e implementar una verificador que haga uso de dicha lógica para probar diferentes propiedades en los autómatas que se aprenden.

Construcción de herramientas de visualización
Aunque hay herramientas existentes para la visualización de autómatas, están pensadas para ser lo más generales posibles con el fin de poderse adaptar a diferentes tipos de autómatas en mayor o menor medida. Por esta razón, carecen, en muchos casos, de funcionalidades adicionales y son poco interactivas.

No existen opciones que sean específicas para los autómatas temporizados que permitan funciones como búsquedas por guardas de tiempo. Además, las librerías que se utilizan en los lenguajes de programación para transformar un autómata descrito como "código" a un formato de visualización suelen ser muy limitadas, con poca documentación o incluso inexistentes.

Una línea alternativa podría ser la construcción de una herramienta de visualización de autómatas temporizados que incorpore funciones en lo referente al carácter temporal de los mismos, que sea interactiva y que se pueda utilizar con librerías que unan la parte de estructura de un autómata en un lenguaje de programación con su visualización.

Referencias

1. Alur, R., Dill, D.L.: The Theory of Timed Automata. In: Procs. of the Workshop on Real-Time: Theory in Practice, REX. Lecture Notes in Computer Science, vol. 600, pp. 45–73. Springer (1991). `https://doi.org/10.1007/BFb0031987`
2. An, J., Wang, L., Zhan, B., Zhan, N., Zhang, M.: Learning real-time automata. Science China-Information Sciences **64**(9) (SEP 2021)
3. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75**(2), 87–106 (NOV 1987)
4. Biermann, A.W., Feldman, J.A.: On the synthesis of finite-state machines from samples of their behavior. IEEE Transactions on Computers (1972)
5. Cornanguer, L.e.a.: Tag: Learning timed automata from logs. In: 36th AAAI Conf. on Artificial Intelligence. pp. 3949–3958. AAAI Conference on Artificial Intelligence (2022)
6. Dash, hls or progressive stream test. `https://bitmovin.com/demos/stream-testformat=dash&manifest=https%3A%2F%2Fcdn.bitmovin.com%2Fcontent%2Fassets%2Fart-of-motion-dash-hls-progressive%2Fmpds%2Ff08e80da-bf1d-4e3d-8899-f0f6155f6efa.mpd`

7. Estándar mpeg-dash. https://www.mpeg.org/standards/MPEG-DASH/
8. Eclipse/ide. https://www.eclipse.org/ide/, Último acceso: 2023-05-23
9. Graphviz. https://graphviz.org/, Último acceso: 2023-05-23
10. Graphviz-java. https://github.com/nidi3/graphviz-java, Último acceso: 2023-05-23
11. Jackson. https://github.com/FasterXML/jackson, Último acceso: 2022-05-19
12. Java programming language. https://docs.oracle.com/javase/7/docs/technotes/guides/language/, Último acceso: 2023-05-20
13. Json. https://www.json.org/json-es.html, Último acceso: 2022-05-19
14. ¿qué es latex? http://desarrolloweb.dlsi.ua.es/cursos/2015/herramientas-investigacion/que-es-latex, Último acceso: 2023-05-23
15. Lin, Q., Zhang, Y., Verwer, S., Wang, J.: Moha: A multi-mode hybrid automaton model for learning car-following behaviors. IEEE Transactions on Intelligent Transportation Systems **20**(2), 790–796 (FEB 2019). https://doi.org/10.1109/TITS.2018.2823418
16. What is maven? https://maven.apache.org/what-is-maven.html, Último acceso: 2023-05-20
17. Mvnrepository. https://mvnrepository.com/, Último acceso: 2023-05-20
18. Overleaf. https://en.wikipedia.org/wiki/Overleaf, Último acceso: 2023-05-23
19. Pastore, F., Micucci, D., Mariani, L.: Timed k-tail: Automatic inference of timed automata. In: 10th IEEE Int. Conf. on Software Testing, Verification and Validation (ICST). pp. 401–411. IEEE Int. Conf. on Software Testing Verification and Validation (2017)
20. The json data interchange syntax. https://www.ecma-international.org/publications-and-standards/standards/ecma-404/, Último acceso: 2023-05-19
21. Tappler, M., Aichernig, B.K., Larsen, K.G., Lorber, F.: Time to learn - learning timed automata from tests. In: Formal Modeling and Analysis of Timed Systems (FORMATS 2019). vol. 11750, pp. 216–235 (2019)
22. Uppaal. https://uppaal.org/, Último acceso: 2023-05-23
23. Vain, J., Kanter, G., Anier, A.: Learning timed automata from interaction traces. IFAC PAPERSONLINE **52**(19), 205–210 (2019)
24. Verwer, S., de Weerdt, M., Witteveen, C.: An algorithm for learning real-time automata. In: Belgian-Dutch Machine Learning Conference / BeNeLearn. p. 128–135 (2007)
25. Verwer, S., de Weerdt, M., Witteveen, C.: A likelihood-ratio test for identifying probabilistic deterministic real-time automata from positive data. In: Grammatical Inference: Theoretical Results and Applications (ICGI 2010). vol. 6339, pp. 203–216 (2010)

Apéndices

A. Tecnologías

En este apéndice se introducen todas las herramientas software utilizadas a lo largo del proyecto.

Como mención especial, se ha utilizado LaTeX para la redacción de la memoria. LaTeX es un sistema de composición de textos, enfocado en la creación de libros, artículos académicos, tesis, y en general todo tipo de documentos relacionados con el ámbito científico y técnico. Es software libre con licencia LPPL [14]. Para la redacción con LaTeX se ha utilizado Overleaf [18], que es una herramienta colaborativa en línea para la redacción y publicación de documentos de carácter científico.

Java

Java es un lenguaje de programación desarrollado por Sun Microsystems (en la actualidad Oracle) y actualmente uno de los más populares [12]. Se trata de un lenguaje de alto nivel, orientado a objetos y fuertemente tipado.

La característica principal por la que es tan popular es su portabilidad, es decir, si un programa desarrollado en Java es ejecutado en una plataforma específica, no necesita ser recompilado para ejecutarse en otra. Esta propiedad se debe a que las aplicaciones de Java son compiladas a Bytecode, un lenguaje de bajo nivel que puede ser interpretado en cualquier máquina virtual Java (JVM) independientemente del sistema donde se encuentre, como se muestra en la Figura 13. Actualmente existen JVM para varias arquitecturas de procesador (Intel y ARM las más populares) y sistemas operativos como Windows, Mac OS o Linux.

Otras características de Java son que es un lenguaje de propósito general, fácil de aprender ;y que tiene una enorme comunidad de desarrolladores que lo apoyan, por lo que existen muchas librerías e interfaces para el desarrollo de aplicaciones Java desarrolladas por terceros.

Maven

Maven es una herramienta usada para la construcción y gestión de cualquier proyecto basado en Java. [16]

Su característica principal es el facilitar el proceso de construcción de proyectos, mediante una plantilla uniforme.

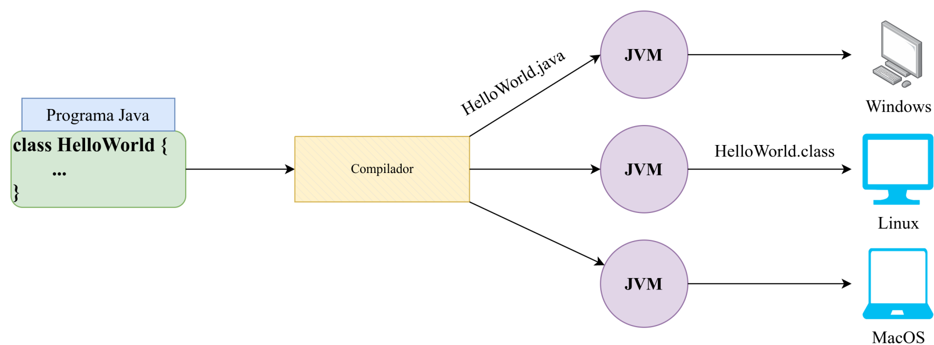


Figura 13: Esquema de la portabilidad de Java

Los proyectos hecho con Maven tienen un archivo principal llamado POM (Project Object Model), que se utiliza para describir el proyecto en construcción, sus dependencias de módulos y componentes externos como plugins.

Maven posee un repositorio [17] desde donde se descarga tanto los plugins como las dependencias necesarias automáticamente y de forma dinámica. Dicho repositorio proporciona acceso a diferentes versiones de diversos proyectos Open Source de Java.

Eclipse

Eclipse en un entorno de desarrollo integrado (IDE) multiplataforma, libre y open source conocido por su versión de Java [8], aunque también está disponible para dar soporte al desarrollo en otros lenguajes de programación como C/C++, PHP y JavaScript/TypeScript, entre otros. Incluso se puede utilizar la misma versión para dar soporte a diferentes lenguajes.

JSON

JSON ("JavaScript Object Notation", Notación de Objetos de JavaScript) [13] es un estándar [20] que define un formato ligero de texto para el intercambio de datos independiente del lenguaje. Es fácil de escribir y de entender por los humanos, mientras que es sencillo de interpretar y generar para las máquinas.

A grandes rasgos, el formato JSON se presenta como un objeto, que es una colección desordenada de pares nombre/valor (o clave/valor). Un objeto comienza con "{" y termina con "}". Cada nombre es seguido por ":" y después por el valor. Los pares nombre/valor están separados por ",". Los nombres son siempre cadenas de texto y los valores pueden ser

```json
{
    "nombre":"David",
    "edad":21
}
```

Figura 14: Ejemplo del formato JSON

una cadena de texto, un número, null, un booleano, otro objeto o un array (son estructuras que puede ir anidadas). La Figura 14 muestra un ejemplo de una cadena en formato JSON.

Jackson

Jackson [11] es una librería de código abierto para Java que se usa principalmente para trabajar con datos en formato JSON. Las operaciones más utilizadas consisten en la conversión de objetos Java en una representación en notación JSON (serializar) y el proceso inverso (deserializar), es decir, transformar una cadena de texto en formato JSON en una instancia de un objeto Java con la misma información.

Esta librería permite también trabajar y manipular otros tipos de codificaciones y formatos como los conocidos XML, CSV y YAML. Además presenta una estructura modularizada en el que se pueden agregar o quitar módulos de funcionalidad según la necesidad.

La librería Jackson está disponible tanto en el repositorio Maven, como en Gradle (otro gestor de proyectos para Java), y además también está disponible una versión jar para añadir manualmente a los proyectos.

Uppaal

Uppaal es una herramienta para el modelado, validación y verificación de sistemas en tiempo real, el modelado se realiza mediante redes de autómatas temporizados, ampliando su información con tipos de datos como enteros y matrices.

Tiene diferentes extensiones que se especializan en ámbitos concretos como sistemas online.

Está escrito en C++ pero tanto su interfaz gráfica como muchas librerías de utilidades están contenidas como archivos JAR que hay que ejecutar con un entorno de ejecución de Java (Java Runtime Environment-JRE).

Está disponible tanto en Windows como sistemas macOS y Linux.

Graphviz Java

Graphviz [9] es un software de código abierto para la visualización de grafos. Graphviz toma descripciones de grafos y crea diagramas en formatos como PNG y SVG para páginas web. Además, permite la personalización de estos grafos permitiendo opciones como modificar el color, el tipo de fuente, el estilo de las líneas y aplicar formas personalizadas.

Graphviz-java [10] es una librería para usar graphviz con Java. Permite crear modelos usando código de Java y convertirlos en diagramas.

B. Resultados de las pruebas

B.1. Pruebas con trazas sintéticas generadas por modelos Uppaal

Longitud	Mean (%)	std (%)	Mean training (%)	Mean valid (%)	Tiempo(s)
25	99,6	0,8	100	98,667	< 0,001
50	99,6	1,2	100	98,667	< 0,001
100	99,4	1,052	100	98	0,001
200	99,6	0,952	100	98,667	0,001
400	99	1,528	100	96,667	0,002

(a) Sistema 1

Longitud	Mean (%)	std (%)	Mean training (%)	Mean valid (%)	Tiempo(s)
25	100	0	100	100	0,002
50	100	0	100	100	0,003
100	100	0	100	100	0,006
200	100	0	100	100	0,012
400	100	0	100	100	0,024

(b) Sistema 2

Longitud	Mean (%)	std (%)	Mean training (%)	Mean valid (%)	Tiempo(s)
25	99,533	0,846	100	98,444	0,001
50	99,467	1,258	100	98,222	0,001
100	99,4	1,052	100	98	0,001
200	98,667	1,955	100	95,556	0,003
400	99,333	1,075	100	97,778	0,005

(c) Sistema 3

Longitud	Mean (%)	std (%)	Mean training (%)	Mean valid (%)	Tiempo(s)
25	100	0	100	100	0,002
50	100	0	100	100	0,003
100	100	0	100	100	0,007
200	100	0	100	100	0,014
400	100	0	100	100	0,028

(d) Sistema 4

Tabla 9: Impacto de la longitud de las trazas en el aprendizaje

K	Mean (%)	std (%)	Mean training (%)	Mean valid (%)	Tiempo(s)
2	99,533	0,991	100	98,444	0,001
3	99,667	0,907	100	98,889	0,001
4	99,667	0,907	100	98,889	0,001
5	99,533	0,846	100	98,444	0,002

(a) Sistema 1

K	Mean (%)	std (%)	Mean training (%)	Mean valid (%)	Tiempo(s)
2	100	0	100	100	0,006
3	100	0	100	100	0,006
4	100	0	100	100	0,006
5	100	0	100	100	0,007

(b) Sistema 2

K	Mean (%)	std (%)	Mean training (%)	Mean valid (%)	Tiempo(s)
2	99	1,770	100	96,667	0,001
3	99,133	1,522	100	97,111	0,001
4	99,2	1,327	100	97,333	0,002
5	99,467	1,024	100	98,222	0,002

(c) Sistema 3

K	Mean (%)	std (%)	Mean training (%)	Mean valid (%)	Tiempo(s)
2	100	0	100	100	0,007
3	100	0	100	100	0,007
4	100	0	100	100	0,007
5	100	0	100	100	0,007

(d) Sistema 4

Tabla 10: Impacto del valor del parámetro K en el aprendizaje

B.2. Modelo aprendido de las trazas del algoritmo DASH