TESIS DE LICENCIATURA

MAURO CHOJRIN - PABLO CECCONI

Verificación de Sistemas Críticos a partir de Trazas de Ejecución usando VTS

Departamento de Computación

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Julio 2015

DIRECTOR:
Victor Braberman

Buenos Aires, Julio 2015

ABSTRACT

There is a wide variety of industry fields for which critical systems are a fundamental piece. This applications have control and real time aspects which make development tasks specially difficult and their incorrect behavior could lead to big losses, both material and human. Is for this reason that it's vital to have tools that provide a high level of confidence about the absense of defects in such applications. One of the most used technics to accomplish such a goal is Model Checking, which requires a formal specification of the requirements to be verified.

An adequate tool for expressing complex real time requirements in a visual and user friendly way is VTS, a scenario based graphic language originally designed to assist in systems verification through Model Checking.

In this paper we'll study the applicability and limitations of VTS for the specification and verification of critical systems based on execution logs, we'll define a set of extensions needed to overcome the found limitations and provide a toolset to aid in the processes of log generation, analysis and verification against extended VTS expressed scenarios.

RESUMEN

Existe una gran variedad de campos de la industria en los cuales son aplicables sistemas críticos. Estas aplicaciones tienen aspectos referidos al control y al tiempo real que dificultan las tareas de desarrollo y su incorrecto funcionamiento puede desencadenar grandes pérdidas, tanto materiales como humanas. Por esta razón, es de suma importancia contar con herramientas que permitan obtener un buen margen de certeza acerca de la ausencia de defectos en dichas aplicaciones. Una de las técnicas ampliamente utilizadas para lograr dicho objetivo es la Verificación mediante Modelos (modelchecking) la cual requiere de una descripción formal de los requerimientos que se desean verificar.

Una herramienta adecuada para expresar complejos requerimientos de tiempo real basados en eventos de forma visual y amigable es VTS, un lenguaje gráfico basado en escenarios diseñado originalmente para asistir en la verificación de sistemas mediante modelos.

En el presente trabajo estudiamos la aplicabilidad y limitaciones de VTS para especificar escenarios para la verificación de sistemas críticos a partir de trazas de ejecución, definimos un conjunto de extensiones necesarias para superar las limitaciones halladas y proveemos un conjunto de herramientas para facilitar el proceso de generación, análisis y verificación de las trazas respecto de escenarios expresados en VTS extendido.

Los países ricos lo son porque dedican dinero al desarrollo científico-tecnológico, y los países pobres lo siguen siendo porque no lo hacen.
La ciencia no es cara, cara es la ignorancia.

— Bernardo Alberto Houssay

AGRADECIMIENTOS

A la Universidad de Buenos Aires y en particular a los docentes de la carrera de Ciencias de la Computación de la Facultad de Ciencias Exactas por la dedicación y el esfuerzo que día a día ponen en transmitir el conocimiento. A nuestros directores que a pesar de los vaivenes nunca perdieron la fé en nosotros y a nuestras familias por el apoyo y la paciencia.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ABREVIATURAS

VTS	Visual Timed Scenarios
API	Application Programming Interface
LTL	Linear Temporal Logic
TCTL	Timed Computation Tree Logic
XML	Extensible Markup Language

1 INTRODUCCIÓN

Existe una gran variedad de campos de la industria en los cuales son aplicables sistemas críticos, entre ellos aeronáutica, electrónica, equipos de salud, etc. Estas aplicaciones generalmente tienen aspectos referidos al control y al tiempo real que dificultan las tareas de desarrollo y la verificación de su correcto funcionamiento.

Las aplicaciones críticas suelen estar presentes en dispositivos embebidos y sus fallas pueden desencadenar grandes pérdidas, tanto materiales como humanas. Por esta razón, es de suma importancia contar con herramientas que permitan obtener un buen margen de certeza acerca de la ausencia de defectos en dichas aplicaciones.

Una serie de técnicas ampliamente utilizadas para reducir los riesgos inherentes al desarrollo de este tipo de aplicaciones es la Verificación Asistida por Computadoras (Computer Aided Verification). Particularmente la Verificación mediante Modelos (Model Checking) constituye un enfoque prometedor dentro del espectro de aproximaciones a la solución a este problema.

La aplicación del Model Checking requiere de una descripción formal de los requerimientos de tiempo real cuyo cumplimiento se desea verificar. Dado que elaborar esta descripción es una tarea difícil y muy propensa a errores es fundamental contar con herramientas, tanto conceptuales como de software, para facilitar el proceso.

Existen diferentes herramientas con varios grados de abstracción así como sintaxis y semánticas distintas para describir los universos que se intentan analizar. Una de estas herramientas es VTS[1], un lenguaje gráfico basado en escenarios que permite expresar complejos requerimientos de tiempo real basados en eventos de forma visual y amigable.

Con VTS es posible expresar una amplia gama de escenarios y predicados sobre éstos, los cuales son traducidos en autómatas que sirven como entrada para una herramienta de verificación de modelos.

La verificación mediante modelos permite validar un modelo de un sistema para asegurar que satisfaga una especificación formal determinada y puede aplicarse en las diversas etapas del desarrollo de software. Sin embargo, esta técnica es insuficiente para asegurar la ausencia de defectos en las aplicaciones ya que la correctitud del modelo no implica la correctitud de la implementación. Esto se debe a diversos factores: los múltiples errores que podrían introducirse en las diferentes etapas del desarrollo del sistema, la existencia de datos que sólo están disponibles en tiempo de ejecución, el hecho de que el comportamiento del sistema podría depender fuertemente de condiciones del entorno de ejecución, etc.

Por esta razón, cuando se trata de sistemas críticos, es necesario complementar la verificación mediante modelos con otras técnicas como la verificación a partir de trazas de ejecución o la verificación en tiempo de ejecución (runtime verification), a veces conocidas como testing pasivo, que permiten aumentar el grado de certeza acerca de la correctitud de la implementación respecto a su especificación.

Estas técnicas de verificación combinan la verificación formal con la ejecución del programa bajo estudio detectando fallas mediante la observación pasiva de su comportamiento durante su operación normal. El comportamiento observado del sistema, por ejemplo a través de las trazas de ejecución, puede ser monitoreado y verificado dinámicamente contra una especificación. Por lo general dicha especificación se escribe empleando formalismos que pueden expresar restricciones temporales, tales como las fórmulas LTL (Lógica temporal lineal) [15], autómatas [11] o diagramas de estado [9]. A diferencia de la verificación mediante modelos donde se trabaja con un modelo simplificado del sistema a verificar, estas técnicas trabajan sobre la ejecución del sistema real. Además, el hecho de lidiar casi exclusivamente con trazas finitas permite aplicar técnicas de verificación más eficientes y veloces.

En el presente trabajo estudiaremos la aplicabilidad y limitaciones de VTS para la verificación de sistemas críticos a partir de trazas de ejecución y proveeremos un conjunto de herramientas para facilitar el proceso de generación, análisis y verificación de las trazas respecto de escenarios expresados en VTS.

Si bien VTS no fue diseñado originalmente para el objetivo específico que perseguimos, existen múltiples razones para elegirlo por sobre otras alternativas similares ya que:

- Es un lenguaje gráfico
- Es un lenguaje formal
- Permite especificar requerimientos temporales

Además, una de las características más apreciables y atractivas de VTS es la posibilidad de expresar requerimientos de tiempo real de forma visual y amigable, evitando el uso de formalismos y lógicas poderosas pero complicadas tales como TCTL (Timed Computation Tree Logic) [4] y el uso explícito de observadores cuya especificación usualmente resulta, en la práctica, engorrosa y muy propensa a errores incluso para quienes tienen experiencia en el uso de métodos formales.

A su vez, el uso de escenarios parciales negativos, es decir, que expresan violaciones de los requerimientos del sistema, constituye una estrategia clave para lidiar con el problema de expresar propiedades basadas en eventos y permitir concentrarse en los requerimientos verdaderamente importantes en lugar de tener que especificar todo el comportamiento esperado del sistema, lo que usualmente es una tarea muy dificil.

Por otra parte, dado que VTS puede ser utilizado para verificación mediante modelos, su aplicación a la verificación de sistemas a partir de trazas de ejecución permitiría evitar la necesidad de aprender y dominar un nuevo lenguaje (y el conjunto de herramientas asociados) a la vez que reutilizar escenarios ya escritos para la verificación del modelo en la verificación de las trazas de ejecución realizando mínimos ajustes.

1.1 ESTRUCTURA DEL TRABAJO

Nuestro trabajo está organizado en tres partes. Durante la primera (Capítulos 1 al 3) daremos un recorrido por el contexto del problema que estudiaremos. Mostraremos cuáles son los objetivos de nuestro estudio y presentaremos las herramientas básicas de las cuales partimos.

La segunda parte está constituida por el capítulo 4. Se trata del análisis de la aplicabilidad de VTS a la verificación de sistemas a partir de trazas de ejecución.

Durante este capítulo presentaremos una herramienta desarrollada por nosotros (TraceIt!) para facilitar la generación de dichas trazas, analizaremos un caso de estudio y presentaremos las limitaciones que VTS tiene al ser aplicado al universo en estudio.

La tercera parte del trabajo (Capítulos 5 al 8) consiste en la propuesta de extensión del lenguaje VTS para incorporar el uso de variables.

A lo largo de esta parte presentaremos:

- La sintaxis y semántica de VTS extendido.
- Las herramientas que desarrollamos a modo de prueba de concepto de nuestro enfoque para la resolución del problema.
- Un algoritmo capaz de generar un autómata temporizado que permita la verificación de propiedades expresadas como escenarios VTS extendidos.
- Un caso de estudio, del cual analizaremos los resultados obtenidos.
- Nuevos desafíos para futuros trabajos que puedan superar las limitaciones alcanzadas en el presente trabajo.

2 OBJETIVO

2.1 MOTIVACIÓN

El correcto funcionamiento de los sistemas de computación, cada día más presentes en todos los órdenes de nuestras vidas, se hace constantemente más crucial para el desarrollo de las actividades humanas. Existen algunos tipos de sistemas en los que la detección de fallas una vez que se encuentran en funcionamiento productivo tiene aparejado un costo elevado tanto en dinero como en riesgo para vidas humanas. En particular, los sistemas con restricciones en sus tiempos de respuesta están cobrando mayor presencia en el universo de aplicaciones utilizadas comúnmente, tanto por individuos como por organizaciones (sistemas de información financiera online, control de procesos automatizados en fábricas, o algunos más críticos, tales como sistemas de monitoreo de pacientes en hospitales, aplicaciones militares de alta precisión o de control aéreo). La complejidad para la verificación de estos sistemas ha sido un problema para la comunidad de desarrolladores de los mismos desde su aparición, y si bien existe una gran cantidad de enfoques dife-rentes para la superación de estas dificultades [5, 14, 6, 2], ninguno se ha impuesto aún por sobre los otros, dando como resultado un campo de exploración amplio y una necesidad concreta de mejorar las herramientas disponibles para garantizar la calidad de los productos de software.

2.2 OBJETIVO GENERAL

Proveer un conjunto de herramientas fáciles de usar y a la vez poderosas para la verificación de sistemas críticos mediante trazas de ejecución. Nos proponemos estudiar la aplicabilidad de la utilización de un lenguaje formal de especificación de requerimientos temporales a la verificación de sistemas de respuesta crítica mediante el análisis de trazas de ejecución. Buscamos con esto poner de manifiesto las ventajas de este enfoque, así como también notar sus puntos débiles y contribuir al mejoramiento de esta forma de comprobación del correcto funcionamiento del software.

2.3 OBJETIVOS ESPECÍFICOS

Este trabajo se centra en los siguientes objetivos:

1. Estudiar la aplicabilidad y limitaciones de VTS para la verificación de sistemas críticos con trazas de ejecución.
2. Extender VTS agregándole variables asociadas a los eventos de forma tal de aumentar lo suficiente su poder expresivo como para permitir especificar el tipo de escenarios que suelen observarse al verificar sistemas implementados mediante el análisis de sus trazas de ejecución.

3. Desarrollar un conjunto de herramientas que sirvan como prueba de concepto de las técnicas de verificación propuestas y que a su vez sirvan como base a futuros desarrollos basados en este enfoque.

Consideramos este un enfoque interesante dado que se trata de una forma simple de llevar el trabajo formal a un entorno real de ejecución, sin tener un alto impacto en el desarrollo de los sistemas a ser verificados, otro de los objetivos primarios de nuestro campo de estudio.

Centraremos nuestra atención al proceso de verificación en tiempo de ejecución (Runtime Verification).

Utilizaremos una serie de herramientas existentes al momento de comenzar nuestro trabajo, las cuales serán explicadas en subsiguientes secciones de esta tesis, entre ellas:

- VTS: Un lenguaje formal de especificación de requisitos temporales
- Un sistema de generación de trazas de ejecución
- Un sistema de verificación de esas trazas contra la especificación realizada utilizando el lenguaje seleccionado

A su vez, desarrollaremos aquellas que nos sean necesarias para complementar a las primeras. Realizaremos análisis de casos de estudio que permitan ejemplificar lo expuesto durante el trabajo, así como también demostrar los alcances del enfoque elegido y las capacidades de las herramientas desarrolladas.

3 ANTECEDENTES

3.1 INTRODUCCIÓN A VTS

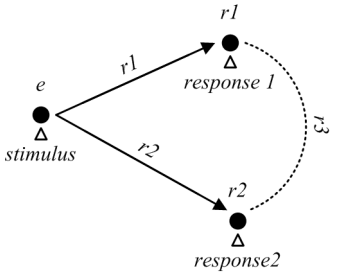
VTS (Visual Timed Event Scenarios) [1] es un lenguaje visual para la definición de requerimientos temporales complejos basados en eventos. El formalismo subyacente está basado en órdenes parciales y permite expresar requerimientos de tiempo real a partir de una notación gráfica basada en *escenarios* de forma visual y amigable.

El uso de escenarios parciales en contraposición a la especificación de todo el comportamiento esperado de un sistema, lo que —incluso de manera abstracta— es una tarea muy dificil, es una de las estrategias claves de VTS para lidiar con el problema de expresar propiedades basadas en eventos. En particular, el enfoque de VTS consiste en describir gráficamente aquellos escenarios genéricos del modelo analizado que violan los requerimientos del sistema. En otras palabras, éstos escenarios —llamados *anti-escenarios* o *escenarios prohibidos*— se emplean para expresar comportamientos no deseados del sistema bajo análisis. Más formalmente, un escenario VTS es un orden parcial anotado de eventos relevantes que puede considerarse un *patrón*, en el sentido que denota a un conjunto (posiblemente infinito) de ejecuciones del sistema que podrían ajustarse al mismo.

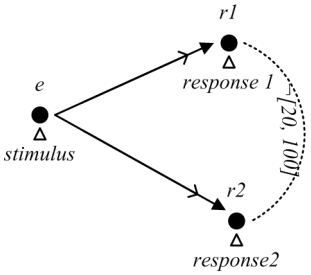
VTS fue concebido para predicar existencialmente sobre ejecuciones. Esto es, para formular una familia de preguntas simples pero a la vez relevantes de la forma "¿Existe una potencial ejecución del sistema que pueda ajustarse a este escenario genérico?". Estas preguntas, que están íntimamente relacionadas con el concepto de anti-escenario, resultan ser decidibles y es posible verificar si un modelo basado en autómatas temporizados del sistema bajo análisis satisface un determinado escenario VTS.

3.1.1 *Sintáxis de VTS*

Un escenario VTS es básicamente un grafo dirigido y acíclico donde los nodos o puntos representan la ocurrencia de un evento o un instante en el tiempo y los ejes representan dependencias de causalidad entre los puntos. Los puntos pueden etiquetarse (opcionalmente) con uno o más símbolos del alfabeto que representa los eventos posibles en el sistema y en ese caso el punto representa la ocurrencia de uno de dichos eventos durante la ejecución. Dos puntos conectados mediante una flecha indican una precedencia del origen respecto del destino. Las flechas, al igual que los puntos, pueden etiquetarse (opcionalmente) con un conjunto de eventos para indicar, en este caso, que dichos eventos están prohíbidos entre ambos puntos. También es posible incluir un intervalo temporal (abierto o cerrado) sobre los ejes para indicar la separación temporal que debe haber entre los puntos de sus extremos. La ausencia de restricciones de causalidad entre dos puntos permite modelar no determinismo (entre eventos de un mismo proceso) y concurrencia (entre eventos de distintos procesos). Los siguientes ejemplos servirán para clarificar estos conceptos.



a) Respuestas separadas



b) Respuestas correlacionadas

Figura 1: Ejemplos de escenarios VTS

El escenario en la Fig. 1(a) expresa un predicado que es verdadero en una ejecución si y sólo si la misma contiene un estímulo e seguido por dos respuestas (los siguientes eventos r1 y r2) no separadas por otra respuesta r3. Los triángulos debajo de los puntos se emplean para asignarles un nombre opcional. Las flechas desde stimulus a response1 y 2 indican que e ocurre antes que r1 y r2. Para establecer que response1 es la primer ocurrencia de su etiqueta después de e (i.e. no hay otro r1 entre e y r1), la flecha stimulus-response1 se etiquetó con r1. Además, para expresar la condición "... no separadas por otra respuesta r3" se une a response1 y 2 mediante una línea punteada etiquetada r3. En este caso se emplea una línea y no una flecha debido a que no hay precedencia entre response1 y response2 (su orden relativo es indiferente). Para ilustrar cuando una determinada ejecución se *ajusta* al escenario VTS de la Fig. 1(a), supongamos que tenemos las siguientes secuencias de eventos:

$$s1 : \ldots a, e, b, c, r1, d, r3, r2, z \ldots$$
$$s2 : \ldots a, e, b, c, r1, d, r2, f, r3, z \ldots$$
$$s3 : \ldots a, e, b, r2, r1, c, r3, r2, z \ldots$$

Entonces las secuencias s2 y s3 se ajustan al escenario, porque los primeros eventos r1 y r2 despues del único evento e no tienen ningún evento r3 de por medio. Por lo tanto, si esto debiera ser interpretado como un escenario *negativo* o *anti-escenario*, deberíamos descartar aquellos sistemas que produzcan s2 y s3 por violar nuestros requerimientos.

Veamos ahora un escenario más complejo que incluye restricciones temporales: una respuesta correlacionada. La Fig. 1(b) muestra un escenario VTS diseñado para la detectar la violación de un requerimiento que dice que siempre que haya dos respuestas que sigan a un estímulo dado e (i.e. los siguientes eventos r1 y r2), éstas deberían estar separa-

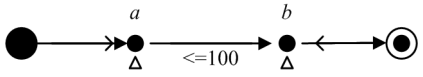


Figura 2: Escenario VTS con símbolos de Comienzo y Fin

das por al menos 20 pero no más de 100 unidades de tiempo. Como en la Fig. 1(a), las flechas indican el orden relativo entre e, r1 y r2. Puede observarse también aquí el uso de una abreviatura para un sub-patrón muy frecuente: el caso en que un cierto punto representa la *siguiente* ocurrencia de un evento después de otro. La abreviatura consiste en una segunda flecha (abierta) cerca de r1 (o r2) y es equivalente a agregar r1 (o r2) como evento prohibido sobre la flecha. A su vez, para expresar que no hay otro evento e después de un e particular y antes de un evento r1 (i.e. para señalar el e exactamente *previo* a r1) hay una notación simétrica: una flecha abierta cerca del extremo de e.

Para verificar la correspondencia entre secuencias y escenarios que incluyen restricciones temporales éstas deben incluir *marcas de tiempo* (timestamps). Consideremos las secuencias anteriores pero con una marca de tiempo agregada a cada evento (por simplicidad usamos números naturales para las marcas de tiempo pero VTS admite cualquier número real no-negativo):

$$\begin{array}{lcccccccccccc} s1: \cdots & a & & e & & b & & c & r1 & d & r3 & r2 & z & \cdots \\ & 12 & & 15 & & 39 & & 50 & 72 & 123 & 140 & 148 & 155 & \end{array}$$

$$\begin{array}{lcccccccccccc} s2: \cdots & a & e & b & & c & r1 & d & r2 & f & r3 & & z & \cdots \\ & 3 & 7 & 12 & & 88 & 109 & 111 & 114 & 121 & 125 & & 152 & \end{array}$$

$$\begin{array}{lcccccccccc} s3: \cdots & a & e & b & r2 & r1 & c & r3 & r2 & & z & \cdots \\ & 5 & 7 & 69 & 78 & 87 & 100 & 146 & 152 & & 199 & \end{array}$$

Puede observarse aquí que la secuencia s1 no se ajusta al escenario de la Fig. 1(b) ya que la distancia temporal entre r1 y r2 ($148 - 72 = 76$) no está incluída en el intervalo $\neg[20, 100]$. Lo opuesto ocurre con la secuencia s2 donde la distancia entre respuestas es de $114 - 109 = 6$; y con s3 donde $87 - 78 = 9$.

VTS también posee dos símbolos especiales para indicar que algo ocurre (o no) desde el comienzo o hasta el final de una ejecución dada: un gran círculo lleno para el *comienzo* y dos círculos concéntricos para el *fin*. La Fig. 2 muestra un escenario que expresa que la distancia temporal desde el primer a hasta el último b (ambos *en toda la ejecución*) es de a lo sumo 100 unidades de tiempo.

Finalmente, VTS posee una notación para identificar al *primero* y al *último* evento en un conjunto: un punto unido a cada evento del conjunto por medio de una línea punteada terminada en un círculo vacío para el caso del *primero* y en un círculo lleno para el caso del *último*. La Fig. 3 describe un caso en el que un sentinela (watchdog) se enciende (*wd on*) al menos 50 unidades de tiempo antes de que ocurra cualquier evento y no se apaga hasta tanto no hayan ocurrido todos los eventos monitoreados. La misión del sentinela consiste en emitir una alarma cuando el intervalo de tiempo entre la ocurrencia del primer y el último evento sea menor a 100 unidades de tiempo. Como se trata de un escenario negativo, las trazas (defectuosas) que se ajusten a él serán aquellas en que se cumplen todas las condiciones especificadas sin que el sentinela emita la alarma.

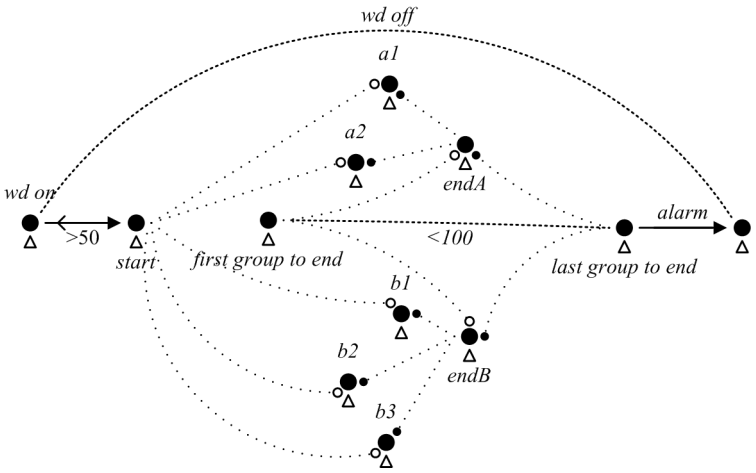


Figura 3: Uso de *primero* y *último* en VTS

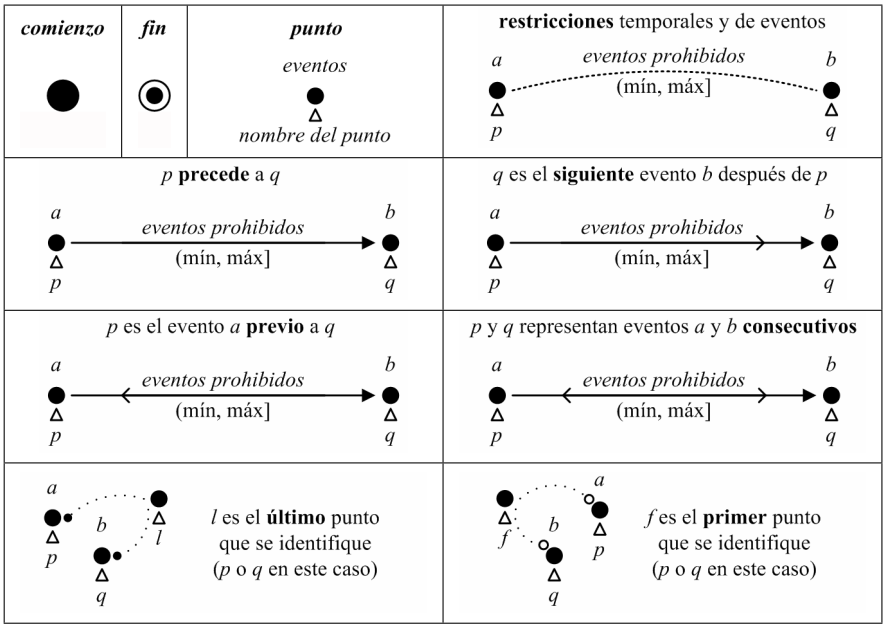


Figura 4: Notación gráfica de VTS

La Fig. 4 muestra la notación gráfica completa de VTS.

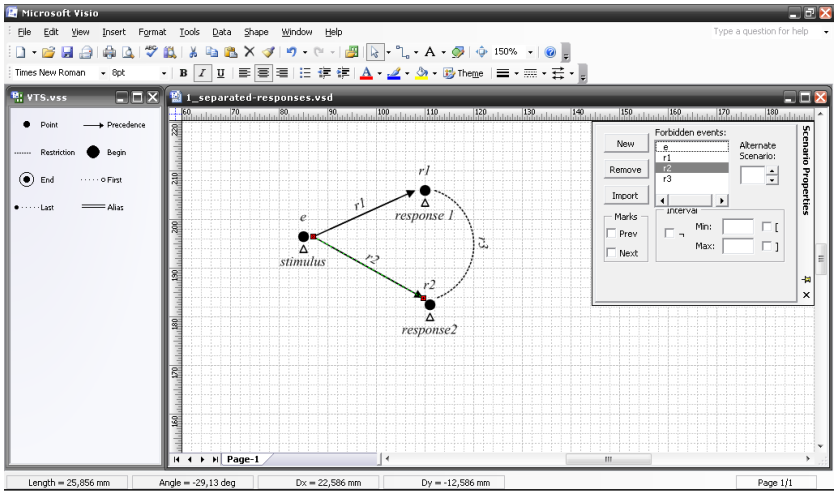
3.2 HERRAMIENTAS AUXILIARES

La implementación de VTS cuenta con un conjunto de herramientas auxiliares entre las que se cuentan un editor gráfico con validación sintáctica y un traductor a autómatas observadores que pueden servir como entrada para herramientas de verificación de modelos como UPPAAL. En esta sección introducimos brevemente cada una de estas herramientas auxiliares que utilizaremos luego para realizar nuestro estudio.

3.2.1 *Herramientas provistas con VTS*

VTS provee un editor gráfico de escenarios que funciona como un complemento (cuenta con un template y un stencil) para el conocido editor de diagramas Microsoft Visio. Cuenta con validación sintáctica y permite guardar los escenarios editados en formato XML. Estos archivos pueden ser luego procesados empleando el componente VTS Translator que permite traducir los escenarios VTS en autómatas observadores aptos para las herramientas de verificación de modelos Kronos (Yovine [16]) y Uppaal.

La siguiente captura de pantalla muestra la edición de un escenario VTS empleando el editor gráfico de escenarios con Visio 2007:



3.2.2 *Obslice*

ObsSlice Braverman et al. [7] es una herramienta de optimización pensada para la verificación de redes de autómatas temporales usando observadores virtuales desarrollada dentro del mismo grupo de investigación autor de VTS. Es capaz de descubrir automáticamente el conjunto de elementos del modelo que pueden ignorarse en cada locación del autómata observador realizando una síntesis de las dependencias de información entre componentes. A partir de una red de autómatas temporizados ObsSlice genera una red transformada que es equivalente pero donde ha sido eliminada toda actividad irrelevante mitigando la explosión del espacio de estados y generando un efecto positivo -a veces dramático- en el desempeño de las herramientas de verificación.

La combinación de VTS con ObsSlice provee a los ingenieros de software una estrategia efectiva para casos en que las herramientas de verificación de modelos enfrentan límites de recursos ya que se ha comprobado empíricamente que la aplicación de estas técnicas aporta especiales beneficios en el caso de escenarios negativos que predican sobre modelos que comprenden múltiples actividades temporizadas concurrentes.

Es interesante notar, además, que ObsSlice cuenta con un modo de ejecución que permite combinar autómatas cuyas etiquetas coincidan. Esta funcionalidad permite utilizarlo como nexo entre un conjunto de

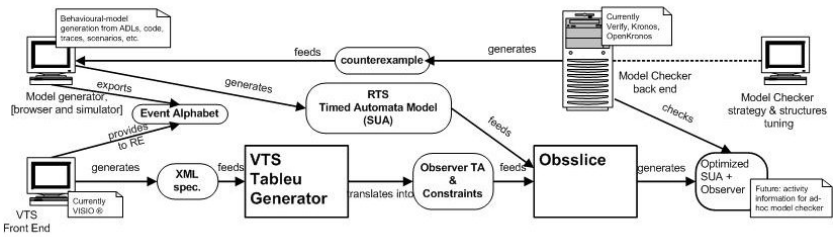


Figura 5: Integración entre VTS y ObsSlice

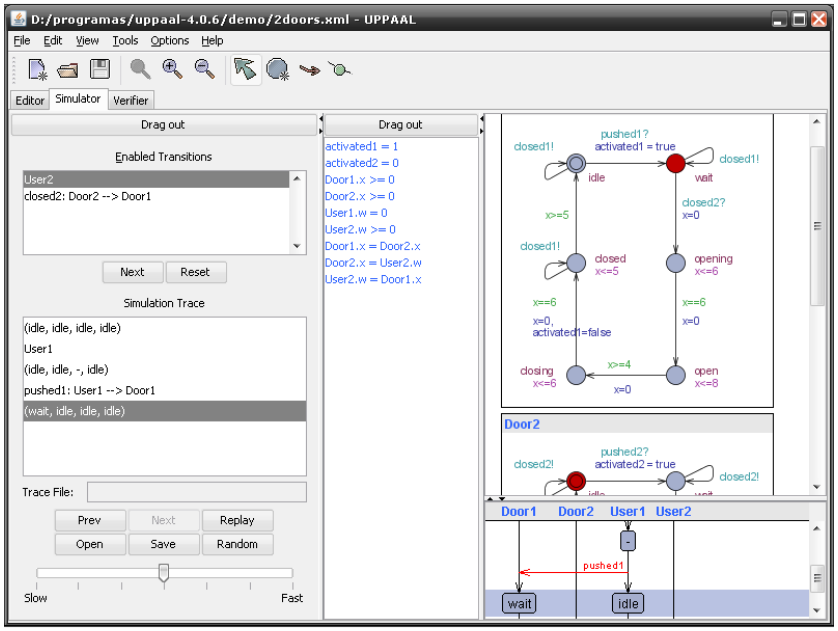
trazas, escenarios expresados en VTS y herramientas de verificación de modelos como Uppaal o Kronos.

El gráfico de la Fig. 5 ilustra la integración entre VTS y ObsSlice

3.2.3 *UPPAAL*

UPPAAL Larsen et al. [12] es un entorno integrado de herramientas para modelado, validación y verificación de sistemas de tiempo real basado en técnicas de resolución de restricciones al vuelo desarrolladas en conjunto por la Universidad de Uppsala y la Universidad de Aalborg. Es apropiado para sistemas que pueden modelarse como una colección de procesos no-determinísticos con estructuras de control y relojes con valores reales que se comunican a través de canales y/o variables compartidas. Las áreas de aplicación típicas incluyen controladores de tiempo real y protocolos de comunicación, en particular aquellos en los que los aspectos temporales son críticos. Está diseñado principalmente para chequear invariantes y alcanzabilidad explorando el espacio de estados de un sistema. Estas características hacen que Uppaal sea adecuado para la verificación de sistemas derivados de la composición de autómatas observadores generados por VTS Translator a partir de escenarios VTS y autómatas temporizados derivados de trazas de ejecución de sistemas, en los que se desea, precisamente, verificar si las trazas satisfacen los escenarios VTS, es decir, si violan requerimientos del sistema. La siguiente captura de pantalla muestra el simulador de Uppaal en acción. En el panel superior derecho se observa cada uno de los autómatas que forman el sistema y en el panel inferior derecho un diagrama de secuencias de mensajes muestra cada instancia de un autómata en una entidad del diagrama y cada evento (sincronizado)

como un mensaje:



Parte I

VTS PARA VERIFICACIÓN DE SISTEMAS A PARTIR DE TRAZAS DE EJECUCIÓN

4

VTS PARA VERIFICACIÓN DE SISTEMAS MEDIANTE TRAZAS DE EJECUCIÓN

En este capítulo presentamos un estudio de la aplicabilidad de VTS a la verificación de sistemas críticos a partir de trazas de ejecución en una plataforma de tiempo real (Windows CE). Comenzamos con una introducción al lenguaje VTS y su notación gráfica ilustrándola con escenarios de ejemplo de su aplicación. Continuamos con una breve descripción del conjunto de herramientas que empleamos para la edición de escenarios, traducción de los escenarios en autómatas observadores, composición de dichos autómatas con los generados a partir de trazas de ejecución y verificación de que dichas trazas satisfacen (violan) las propiedades expresadas en los escenarios.

4.1 TRACEIT!

Generador de trazas de ejecución para aplicaciones distruibuidas de tiempo real

Actualmente las comunidades académicas y las áreas de desarrollo e investigación de la industria están prestando mucha atención al uso de herramientas de verificación automáticas tales como verificadores de modelos para lidiar con los desafíos que surgen durante el desarrollo de sistemas distribuidos y embebidos. Como en muchas aplicaciones grandes es inviable realizar un cubrimiento completo de su comportamiento se están estudiando nuevos enfoques donde se resigna cubrimiento en pos de una mayor escalabilidad. Dentro de esta clase de propuestas se encuentran el análisis de trazas de ejecución y el monitoreo en tiempo de ejecución, donde las trazas de ejecución se comparan contra una especificación formal de requerimientos. Por ejemplo, MaC tool [13], TempRover [8] y PathExplorer [10] verifican si trazas de eventos satisfacen especificaciones basadas en algún tipo de lógica temporal, monMSC [MON] verifica la conformidad de registros de eventos contra especificaciones basadas en Diagramas de Secuencias de Mensajes (Message Sequence Charts), mientras que AsmL Test [ASML] realiza análisis de trazas basado en estados para especificaciones de Máquinas de Estados Abstractas (Abstract State Machines). En todos los casos, los desarrolladores necesitan que sus aplicaciones generen registros de estados o eventos apropiados para poder tomar ventaja de estas herramientas. Nuestro objetivo es estudiar la forma de aprovechar el poder expresivo y la facilidad de uso de VTS para la especificación formal de requerimientos y utilizarlo para realizar verificación de software a partir de trazas de ejecución del sistema implementado, tal como lo hacen las herramientas mencionadas arriba. En pos de ese objetivo concebimos TraceIt!: una herramienta nueva que permite que aplicaciones embebidas que corran bajo el sistema operativo Windows CE .NET y .NET Compact Framework puedan registrar eventos temporizados relevantes de forma sencilla. Para mostrar cómo esta tecnología de instrumentación puede formar parte de un conjunto de herramientas más grande demostramos, como parte de nuestra implementación, la utilización de registros de eventos producidos con diferentes tipos de

herramientas de visualización y chequeo de conformidad. Por ejemplo, las trazas se pueden visualizar en forma amigable usando editores de diagramas de secuencias de mensajes provistos con herramientas CASE como Telelogic Tau, Rational Rose RT o incluso con verificadores de modelos como Uppaal realizando previamente una conversión de las trazas a autómatas temporizados. También se puede chequear si las trazas generadas conforman un determinado protocolo usando especificaciones expresadas por medio de hMSCs (Diagramas de secuencias de mensajes de alto nivel, ITU Z.120 [MSC]) por medio de una herramienta ([MON]) desarrollada recientemente por el mismo el grupo de investigación autor de VTS. Estas funcionalidades se aplican, típicamente, para detección y diagnóstico de discrepancias en el orden o la puntualidad✎ de eventos o para medir cubrimiento durante las pruebas de integración de sistemas distribuidos. Además, las trazas generadas pueden ser analizadas automáticamente para determinar si satisfacen escenarios que violan requerimientos de tiempo real tales como frescura, respuesta acotada, correlación, etc. que pueden expresarse fácilmente usando VTS.

✎ *Elegimos traducir la palabra "timing" como "puntualidad" para no perder la referencia implícita a un reloj. En otros contextos, con el mismo significado, un suceso con el correcto "timing" debería traducirse como "oportuno". Sin embargo, esa palabra tiene otras connotaciones que en este caso la harían más ambigua que la elegida.*

4.1.1 *Diseño e Implementación de TraceIt!*

Algunos requisitos deseables para una herramienta de generación de trazas son:

1. Facilidad de integración para el desarrollador.
2. Baja interferencia en el funcionamiento de la aplicación observada de forma que las trazas generadas sean representativas del sistema real.
3. Alta precisión de las marcas de tiempo registradas.

TraceIt! es escencialmente una API que provee una capa de abstracción sobre la generación de trazas de eventos en tiempo de ejecución para aplicaciones embebidas que corran bajo Windows CE .NET y .NET Compact Framework.

Para la contrucción de TraceIt! aprovechamos las siguientes características del sistema operativo Windows CE .NET:

- CeLog: una API de bajo nivel para registro de eventos con marcas de tiempo
- Las herramientas de depuración de Windows CE
- El soporte para la biblioteca estándar de tipos abstractos de C++ bajo Windows CE .NET

Además, durante las etapas de implementación y prueba también aprovechamos algunas facilidades de sincronización de la plataforma Windows CE, especialmente las primitivas de administración de zonas críticas. Las herramientas de desarrollo empleadas durante el proyecto fueron:

- Platform Builder
- Windows CE Emulator para procesadores Intel x86
- Remote Kernel Tracker

- Remote Performance Monitor
- La utilidad "readlog" para realizar procesamientos auxiliares de los registros capturados mediante la API CeLog
- Embedded Visual C++

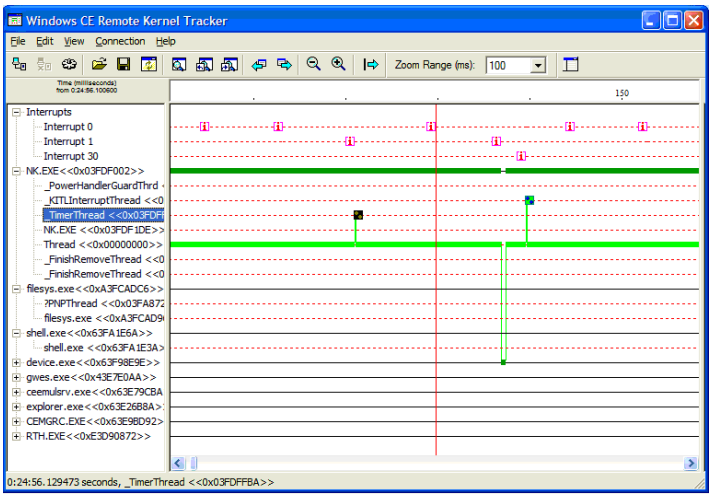
La API de TraceIt! (para una descripción detallada de sus métodos consultar el apéndice correspondiente) ofrece a los desarrolladores primitivas para registrar eventos de envío y recepción de mensajes a través de canales definibles por el desarrollador y eventos genéricos con una o dos marcas de tiempo (en este último caso la API provee primitivas para tomar una marca de tiempo previa y una posterior a la ocurrencia del evento).

TraceIt! está implementado en C++ y sólo está disponible para aplicaciones escritas en dicho lenguaje. Sin embargo, esto no debería resultar una limitación ya que es el único lenguaje de alto nivel en el que pueden escribirse aplicaciones de tiempo real bajo Windows CE y se trata, por otra parte, de uno de los lenguajes más populares dentro de la comunidad de desarrolladores de sistemas de tiempo real. La consideración más importante a la hora de migrar TraceIt! a otras plataformas es que se apoya en la API CeLog para la generación de timestamps atómicos y loggeo y por lo tanto, ésta debe reemplazarse por una API equivalente para poder garantizar la precisión requerida

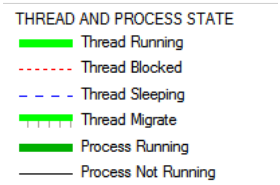
CeLog: Registro de eventos a bajo nivel bajo Windows CE .NET

Uno de los problemas fundamentales que se enfrentan al implementar un generador de trazas está relacionado con la precisión en la obtención de las marcas de tiempo. Por lo general, los sistemas operativos modernos, sobre todo aquellos de tiempo real, incluyen soluciones de registro de eventos con marcas de tiempo en el propio kernel con el objetivo de garantizar la máxima precisión y atomicidad en la ejecución dichas funciones. Naturalmente, la eficiencia en la implementación del código que registra las trazas es crucial ya que uno de sus usos más frecuentes es utilizarlas para diagnosticar problemas de desempeño en entornos de producción altamente sobrecargados. Por lo tanto, para ser aceptable, un mecanismo de registro de trazas debe tener el menor impacto posible sobre el sistema a ser observado. Windows CE provee un subsistema de registro de eventos llamado CeLog que se encuentra implementado como una biblioteca que puede cargarse dinámicamente en el Kernel y permite registrar un cierto número de eventos de kernel predefinidos. Además, CeLog también expone una biblioteca de funciones que pueden ser usadas para registrar eventos definidos por el usuario o para controlar el seguimiento de eventos. CeLog resultó central para la implementación de TraceIt! ya que nos permitió abstraernos del problema de la atomicidad necesaria para la obtención de las marcas de tiempo y concentrarnos en ofrecer una API adecuada y amigable para el desarrollador que lo abstrayera de la complejidad de interactuar con funciones del Kernel del sistema operativo. CeLog colecta y almacena datos en una ubicación configurable por el usuario, que puede ser tanto un directorio dentro de la computadora huesped -durante la fase de desarrollo- o bien el directorio raiz del sistema de archivos del dispositivo embebido. La forma más simple de recolectar datos registrados mediante CeLog es conectándose al dispositivo con una herramienta llamada Remote Kernel Tracker, incluída con Platform

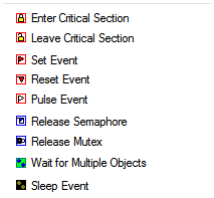
Builder. Esta herramienta permite recolectar y visualizar información de CeLog en tiempo real transportando los datos desde el dispositivo a la PC y mostrándolos gráficamente. La vista gráfica que provee Remote Kernel Tracker ayuda a visualizar la actividad del sistema y hacerse una idea general de las interacciones entre hilos de ejecución y procesos:



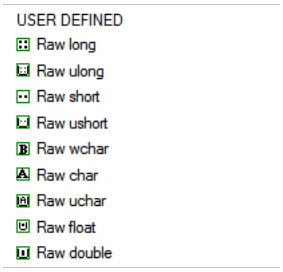
Las siguientes referencias ayudan a comprender mejor el gráfico de arriba y apreciar el enorme valor agregado del mismo en el proceso de depuración:



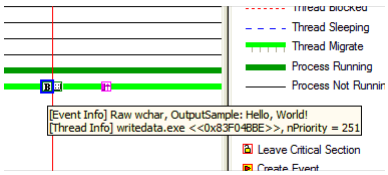
Los siguientes íconos son empleados por Remote Kernel Tracker para representar diferentes eventos de la dinámica de planificación de la ejecución de procesos y son de enorme utilidad para ayudar a comprender lo que está sucediendo en el sistema:



Además, los tipos de datos definidos por el usuario y registrados a través de CeLog también tienen sus propios íconos para ayudar a identificarlos:



Incluso es posible visualizar los textos asociados a dichos eventos directamente desde el Remote Kernel Tracker:



También es posible recolectar datos de CeLog construyendo una imagen de un dispositivo con Platform Builder y habilitando la opción de seguimiento de eventos (event tracking). En este caso, cuando el dispositivo basado en Windows CE comienza a ejecutar, la aplicación CeLogFlush.exe crea un archivo llamado Celog.clg en un directorio de la PC huesped definido por el usuario. Este archivo contiene datos crudos en formato binario y es actualizado periódicamente mientras el dispositivo está corriendo o puede forzarse su actualización mediante la ejecución de CeLogFlush.exe. Este último esquema fue el que utilizamos durante el desarrollo y las pruebas de TraceIt! Construimos una imagen personalizada para un dispositivo embebido basado en Windows CE utilizando Platform Builder y habilitamos la opción de seguimiento de eventos para poder capturar la salida de TraceIt! en un archivo crudo para su posterior procesamiento.

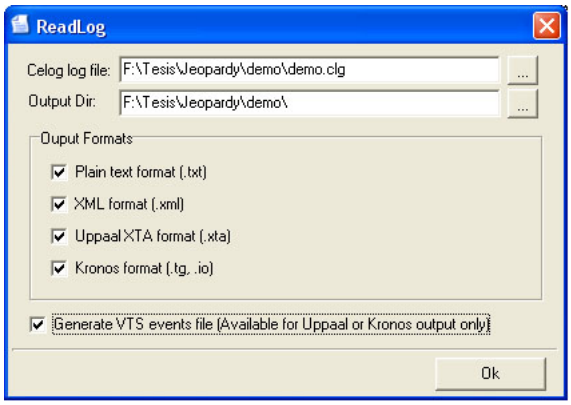
4.1.2 *Readlog*

Si bien el kit de desarrollo de aplicaciones para Windows CE provee una utilidad de línea de comandos que permite leer los archivos binarios de extensión .clg generados por CeLog en la PC huesped, se trata de una solución limitada ya que sólo permite realizar una conversión del archivo a formato de texto plano y no posee capacidades de filtrado u análisis de los eventos contenidos en él.

Además, los datos capturados mediante CeLog se dividen en zonas similares a las zonas de depuración, lo que permite filtrar el registro de eventos de salida y recolectar de él sólo los eventos en los que uno está interesado. Las zonas son definibles por el usuario tanto desde la PC de desarrollo como desde el dispositivo embebido y esto último es precisamente lo que hace TraceIt!

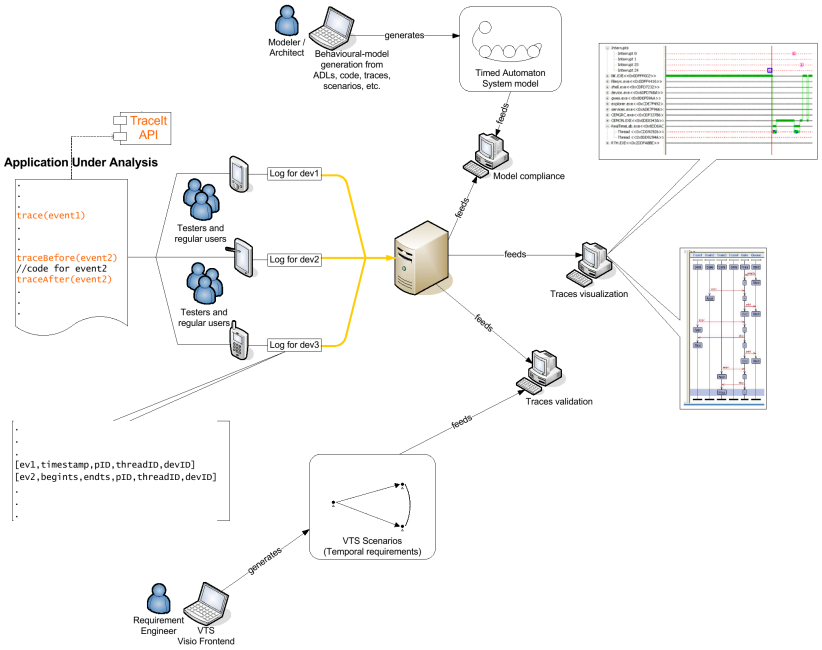
Por esta razón decidimos desarrollar una versión de ReadLog, más poderosa y amigable que la utilidad provista con Windows CE, que nos permitiera procesar el archivo de salida de CeLog y aprovechar las zonas para extraer sólo la información generada por TraceIt! y convertirla fácilmente en múltiples formatos de salida para diversos propósitos.

Nuestra versión de Readlog es una aplicación gráfica para Windows de 32-bit desarrollada en C++ y fue diseñada de modo tal que fuera fácilmente extensible para la incorporación de nuevos filtros de procesamiento y formatos de salida.



4.2 ARQUITECTURA DE LA SOLUCIÓN

El siguiente gráfico ilustra la arquitectura de la solución implementada, incluyendo la interacción con los escenarios VTS y el verificador de modelos Uppaal:



4.3 CASO DE ESTUDIO: JEOPARDY

En esta sección presentamos, a modo de caso de estudio, una aplicación para Windows CE que desarrollamos con el objetivo de estudiar la aplicación práctica de TraceIt! y el conjunto de herramientas ya descripto para verificación de propiedades de tiempo real especificadas en VTS a partir de trazas de ejecución obtenidas con TraceIt!

La aplicación propuesta se trata de una adaptación de un popular programa de entretenimientos llamado *Jeopardy*, y su funcionamiento general puede resumirse así:

- En cada ronda, el conductor del programa dice una frase, que constituye la respuesta a una pregunta que los participantes deben acertar eligiendo de un conjunto de preguntas predefinidas que visualizarán una vez que comience a correr el tiempo.

- Cada participante cuenta con algún dispositivo que le permite visualizar las opciones de preguntas y seleccionar la que considere correcta. Cada ronda dura un lapso determinado de tiempo, pasado el cual, si el participante no ha seleccionado ninguna opción, se considera que ha renunciado a participar en esa ronda.
- Se evalúan las elecciones de los jugadores en el orden en que se reciben y gana la ronda (sumando puntos) el primer jugador en elegir la pregunta correcta, mientras que aquellos que hubieran elegido una opción incorrecta son penalizados mediante el descuento de una cierta cantidad de puntos.
- El juego continúa hasta completarse un cierto número predeterminado de rondas.

CONDICIONES GENERALES

- Por cada ronda sólo puede existir un ganador, el primero que acierta la respuesta correcta, y este suma 10 puntos. Todos los demás, incluso aquellos que hubieran contestado correctamente no suman ni restan puntos.
- En cada ronda todos los jugadores que seleccionan una opción incorrecta, en el tiempo estipulado, pierden 15 puntos.
- Si un jugador decide no participar de una ronda no suma ni resta puntos.

4.3.1 *Implementación*

La arquitectura de la aplicación de ejemplo consta de los siguientes 1 servidor y n clientes.

EL SERVIDOR Tiene a su cargo las siguientes tareas:

- Llevar adelante el juego (administrar las rondas, puntos, etc...)
- Seleccionar en cada ronda la respuesta y las preguntas asociadas a la misma
- Enviar las preguntas a los participantes una vez que el conductor lo decide (en el ejemplo este paso será automático).
- Esperar durante el lapso estipulado las respuestas de los clientes o hasta que todos hayan respondido.
- Procesar las respuestas a medida que estas son recibidas.
- Determinar el ganador y los perdedores, si los hubiere.

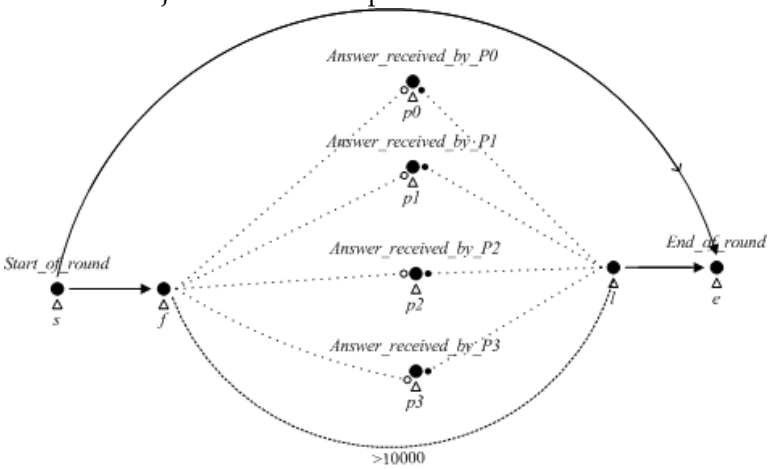
EL CLIENTE Tiene a su cargo las siguientes tareas:

- Esperar a que el servidor le envíe las preguntas.
- Enviar al servidor la confirmación de que se han recibido las preguntas.
- Seleccionar una pregunta (o no) según la preferencia del jugador (en el ejemplo será al azar).
- Enviar al servidor la selección tan pronto como el jugador lo decida (en el ejemplo este tiempo será aleatorio).

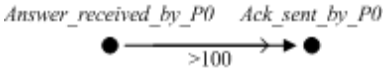
4.3.2 *Propiedades a verificar*

A continuación se enumeran las propiedades a verificar y se muestran los escenarios correspondientes, expresados en VTS. Téngase presente que se trata de escenarios negativos, es decir, que expresan una propiedad que no debería ser satisfecha por ninguna traza de ejecución del sistema implementado:

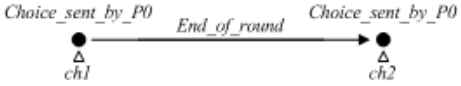
- El tiempo transcurrido entre la recepción de las preguntas por parte del primer jugador y el último no debe superar un cierto umbral. El objetivo de este test es asegurar que ningún participante tiene ventajas sobre sus competidores.



- El tiempo en que el cliente procesa las preguntas recibidas y envía la confirmación no puede ser mayor que cierta cota. Este test verifica que el cliente este funcionando correctamente.

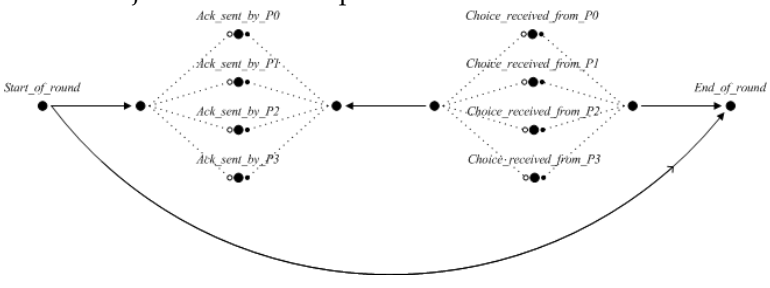


- Un jugador no puede enviar más de una selección por ronda.

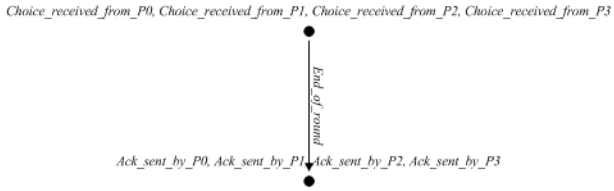


- El tiempo que toma al servidor determinar el resultado de la ronda una vez que ha finalizado la etapa de recepción de mensajes de los jugadores no puede superar un límite preestablecido. Este test verifica el funcionamiento del servidor, apunta a que el juego sea dinámico.

- El servidor no puede recibir ninguna opción por parte de un jugador hasta tanto todos los participantes hayan recibido las preguntas. El objetivo de este test es asegurar que ningún particpante tiene ventajas sobre sus competidores.



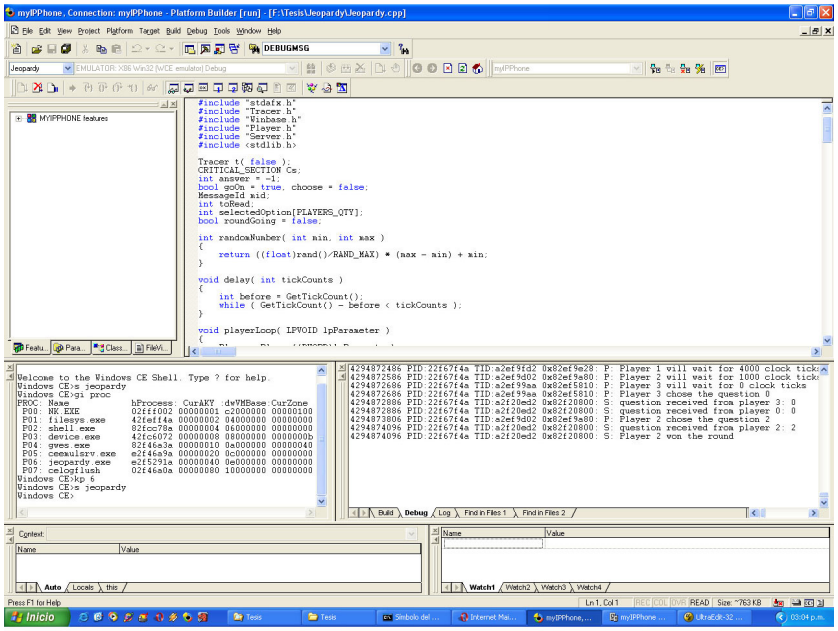
Nótese que este escenario puede reescribirse en forma más compacta del siguiente modo:



- El orden de procesamiento de las preguntas debe coincidir con el orden de envío de las mismas.

4.4 RESULTADOS

Hicimos una corrida de Jeopardy de 25 rondas y obtuvimos así un archivo demo.clg con todos los eventos registrados por TraceIt! vía CeLog. La siguiente captura de pantalla muestra la pantalla principal de Platform Builder en el momento de realizar la corrida. En la consola de depuración, que se encuentra en la esquina inferior derecha, pueden observarse algunos mensajes emitidos por TraceIt! durante la ejecución de Jeopardy que permiten seguir el desarrollo del juego:

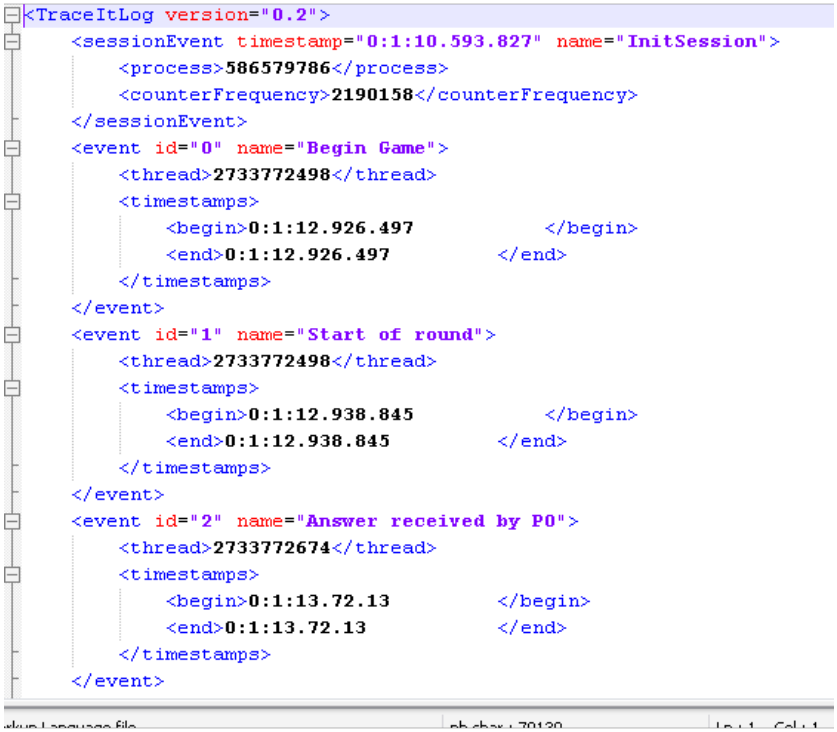


Empleando nuestra versión de Readlog transformamos el log obtenido en los siguientes formatos: texto plano, xml, uppaal (.xta) y kronos (.tg, .io, .esp) y generamos además el listado de eventos para VTS. A continuación mostramos un extracto del resultado de la conversión de la traza a formato de texto plano:

```
#
# TraceIt! Log Start
#
0:1:10.593.827 Init Session PID: 586579786 Counter Freq: 2190158
0:1:12.926.497 Atomic Event Ev Id: 0 Thrd Id: 2733772498 Name: "Begin Game"
0:1:12.938.845 Atomic Event Ev Id: 1 Thrd Id: 2733772498 Name: "Start of round"
0:1:13.72.13 Atomic Event Ev Id: 2 Thrd Id: 2733772674 Name: "Answer received by P0"
0:1:13.72.928 Atomic Event Ev Id: 3 Thrd Id: 2733772674 Name: "Ack sent by P0"
0:1:13.173.609 Atomic Event Ev Id: 4 Thrd Id: 2733613010 Name: "Answer received by P1"
0:1:13.184.837 Atomic Event Ev Id: 5 Thrd Id: 2733613010 Name: "Ack sent by P1"
0:1:13.303.468 Atomic Event Ev Id: 6 Thrd Id: 2733612290 Name: "Answer received by P2"
```

```
0:1:13.323.425 Atomic Event Ev Id: 7 Thrd Id: 2733612290 Name: "Ack sent by P2"
0:1:13.508.794 Atomic Event Ev Id: 8 Thrd Id: 2733611434 Name: "Answer received by P3"
0:1:13.519.834 Atomic Event Ev Id: 9 Thrd Id: 2733611434 Name: "Ack sent by P3"
0:1:13.782.721 Atomic Event Ev Id: 10 Thrd Id: 2733772498 Name: "Everybody ready"
0:1:16.96.504 Before Event Ev Id: 11 Thrd Id: 2733612290 Name: "Choice sent by P2"
0:1:16.98.675 End Event Ev Id: 11
0:1:16.520.253 Atomic Event Ev Id: 12 Thrd Id: 2733772498 Name: "Choice received from P2"
0:1:18.386.147 Before Event Ev Id: 13 Thrd Id: 2733613010 Name: "Choice sent by P1"
0:1:18.386.265 End Event Ev Id: 13
0:1:19.20.722 Atomic Event Ev Id: 14 Thrd Id: 2733772498 Name: "Choice received from P1"
0:1:21.773.152 Atomic Event Ev Id: 15 Thrd Id: 2733772498 Name: "End of round"
0:1:21.777.978 Atomic Event Ev Id: 16 Thrd Id: 2733772498 Name: "Start of round"
0:1:22.90.54 Atomic Event Ev Id: 17 Thrd Id: 2733613010 Name: "Answer received by P1"
0:1:22.93.411 Atomic Event Ev Id: 18 Thrd Id: 2733613010 Name: "Ack sent by P1"
0:1:22.191.759 Atomic Event Ev Id: 19 Thrd Id: 2733612290 Name: "Answer received by P2"
0:1:22.192.126 Atomic Event Ev Id: 20 Thrd Id: 2733612290 Name: "Ack sent by P2"
0:1:25.371.401 Atomic Event Ev Id: 21 Thrd Id: 2733772674 Name: "Answer received by P0"
0:1:25.371.764 Atomic Event Ev Id: 22 Thrd Id: 2733772674 Name: "Ack sent by P0"
0:1:25.866.974 Atomic Event Ev Id: 23 Thrd Id: 2733611434 Name: "Answer received by P3"
0:1:25.867.133 Atomic Event Ev Id: 24 Thrd Id: 2733611434 Name: "Ack sent by P3"
0:1:26.72.534 Atomic Event Ev Id: 25 Thrd Id: 2733772498 Name: "Everybody ready"
0:1:26.218.784 Before Event Ev Id: 26 Thrd Id: 2733772674 Name: "Choice sent by P0"
0:1:26.218.891 End Event Ev Id: 26
0:1:26.908.528 Atomic Event Ev Id: 27 Thrd Id: 2733772498 Name: "Choice received from P0"
0:1:30.716.565 Before Event Ev Id: 28 Thrd Id: 2733613010 Name: "Choice sent by P1"
0:1:30.716.675 End Event Ev Id: 28
0:1:31.222.216 Atomic Event Ev Id: 29 Thrd Id: 2733772498 Name: "Choice received from P1"
0:1:33.911.67 Atomic Event Ev Id: 30 Thrd Id: 2733772498 Name: "End of round"
0:1:33.928.87 Atomic Event Ev Id: 31 Thrd Id: 2733772498 Name: "Start of round"
0:1:34.92.216 Atomic Event Ev Id: 32 Thrd Id: 2733772674 Name: "Answer received by P0"
0:1:34.93.945 Atomic Event Ev Id: 33 Thrd Id: 2733772674 Name: "Ack sent by P0"
0:1:34.233.484 Atomic Event Ev Id: 34 Thrd Id: 2733613010 Name: "Answer received by P1"
0:1:34.233.880 Atomic Event Ev Id: 35 Thrd Id: 2733613010 Name: "Ack sent by P1"
0:1:37.169.114 Atomic Event Ev Id: 36 Thrd Id: 2733611434 Name: "Answer received by P3"
0:1:37.169.282 Atomic Event Ev Id: 37 Thrd Id: 2733611434 Name: "Ack sent by P3"
0:1:38.742.662 Atomic Event Ev Id: 38 Thrd Id: 2733612290 Name: "Answer received by P2"
0:1:38.742.850 Atomic Event Ev Id: 39 Thrd Id: 2733612290 Name: "Ack sent by P2"
0:1:39.93.980 Atomic Event Ev Id: 40 Thrd Id: 2733772498 Name: "Everybody ready"
0:1:39.510.983 Before Event Ev Id: 41 Thrd Id: 2733612290 Name: "Choice sent by P2"
0:1:39.511.109 End Event Ev Id: 41
0:1:39.913.972 Atomic Event Ev Id: 42 Thrd Id: 2733772498 Name: "Choice received from P2"
0:1:45.565.806 Before Event Ev Id: 43 Thrd Id: 2733611434 Name: "Choice sent by P3"
0:1:45.565.915 End Event Ev Id: 43
0:1:45.851.156 Atomic Event Ev Id: 44 Thrd Id: 2733772498 Name: "Choice received from P3"
0:1:46.759.927 Atomic Event Ev Id: 45 Thrd Id: 2733772498 Name: "End of round"
0:1:46.769.585 Atomic Event Ev Id: 46 Thrd Id: 2733772498 Name: "Start of round"
0:1:47.135.412 Atomic Event Ev Id: 47 Thrd Id: 2733612290 Name: "Answer received by P2"
0:1:47.135.802 Atomic Event Ev Id: 48 Thrd Id: 2733612290 Name: "Ack sent by P2"
...
0:6:6.308.129 Before Event Ev Id: 365 Thrd Id: 2733613010 Name: "Choice sent by P1"
0:6:6.308.247 End Event Ev Id: 365
0:6:6.848.444 Atomic Event Ev Id: 366 Thrd Id: 2733772498 Name: "Choice received from P1"
0:6:7.498.800 Before Event Ev Id: 367 Thrd Id: 2733611434 Name: "Choice sent by P3"
0:6:7.498.911 End Event Ev Id: 367
0:6:7.805.297 Atomic Event Ev Id: 368 Thrd Id: 2733772498 Name: "Choice received from P3"
0:6:10.458.918 Atomic Event Ev Id: 369 Thrd Id: 2733772498 Name: "End of round"
0:6:10.459.460 Atomic Event Ev Id: 370 Thrd Id: 2733772498 Name: "Game Over"
0:6:11.39.830 End Session PID: 586579786
#
# TraceIt! Log End
#
```

A continuación mostramos un extracto de la misma corrida pero convertida mediante Readlog a formato XML:



```
<TraceItLog version="0.2">
    <sessionEvent timestamp="0:1:10.593.827" name="InitSession">
        <process>586579786</process>
        <counterFrequency>2190158</counterFrequency>
    </sessionEvent>
    <event id="0" name="Begin Game">
        <thread>2733772498</thread>
        <timestamps>
            <begin>0:1:12.926.497          </begin>
            <end>0:1:12.926.497        </end>
        </timestamps>
    </event>
    <event id="1" name="Start of round">
        <thread>2733772498</thread>
        <timestamps>
            <begin>0:1:12.938.845          </begin>
            <end>0:1:12.938.845        </end>
        </timestamps>
    </event>
    <event id="2" name="Answer received by P0">
        <thread>2733772674</thread>
        <timestamps>
            <begin>0:1:13.72.13        </begin>
            <end>0:1:13.72.13          </end>
        </timestamps>
    </event>
```

4.4.1 *Proceso*

El proceso de verificación de los escenarios contra las trazas consistió de los siguientes pasos:

1. Para cada escenario armamos el correspondiente `.io` ya que el translator no lo genera y el .io que genera ReadLog con todos los eventos no es compatible ObsSlice porque tiene más eventos de los que realmente figuran en el `.tg`.
2. Guardamos copias de todos los observadores y sus correspondientes `.io` en la forma `Obs-Escenario<n>.{tg|io}`
3. Poniendo cada uno de ellos como `Obs.tg` y `Obs.io` respectivamente corrimos el ObsSlice y generamos un archivo de Uppaal que fuimos nombrando `demo.Escenario<n>.xta`
4. Para cada archivo de Uppaal generado verificamos si era alcanzable el estado de *ACCEPT* y luego capturamos algunas pantallas.

La Fig. 6 ilustra el proceso completo de verificación de aplicaciones usando VTS y el conjunto de herramientas descripto aquí.

4.5 LIMITACIONES DE VTS

Del estudio realizado hasta el momento, podemos concluir que contar con un lenguaje formal, gráfico y sencillo como VTS constituye un punto de apoyo importante en el logro de nuestro objetivo: contar con un set de herramientas que permitan agilizar el proceso de verificación

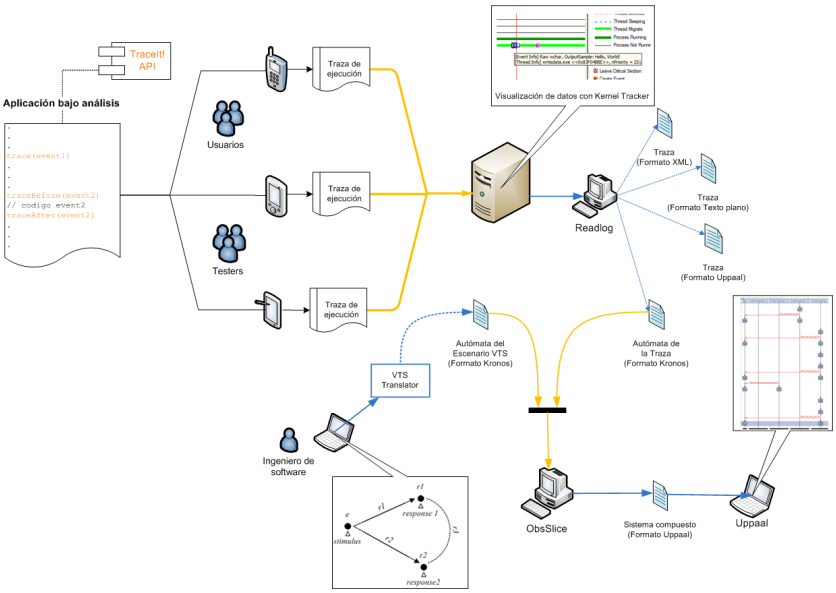


Figura 6: Proceso de verificación de aplicaciones con VTS

de sistemas mediante el análisis de trazas de ejecución. Sin embargo, al intentar generalizar la aplicación de VTS en su versión original, nos encontramos con limitaciones inherentes a su diseño (cabe recordar en este punto que VTS fue pensado como un lenguaje para realizar verificación de modelos, no de sistemas efectivamente implementados, y si bien se espera una fuerte correlación entre ambos, los universos abarcados por cada uno no son exactamente los mismos), especialmente en cuanto a los siguientes puntos:

- La única relación que puede establecerse entre los eventos es de índole temporal (sucedió antes que)
- Un evento sólo es distinguible por su nombre (Lo cual implica que los eventos no pueden ser agrupados o filtrados, resultando en una relación sumamente estrecha entre una traza en particular y el escenario descripto lo cual impacta en el poder de abstracción del lenguaje).

Estos puntos toman especial relevancia en aquellos casos en los que el universo de eventos es muy grande o peor aún, no es acotado.

Revisando nuevamente el caso de estudio presentado notaremos que si el número de jugadores fuese muy grande la confección de los escenarios se volvería impráctica, por ejemplo en el caso de los escenarios 1, 4 y 5.

La insuficiencia de poder expresivo se hace más evidente al querer generalizar las propiedades expresadas en los escenarios 2 y 3. Si volvemos a leer la explicación informal de la propiedad a verificar ("El tiempo en que el cliente procesa las preguntas recibidas y envía la confirmación no puede ser mayor que cierta cota. Este test verifica que el cliente este funcionando correctamente"), notaremos que el escenario no expresa en forma exacta lo que la descripción informal pretende, ya que en la versión original de VTS no existe la posibilidad de predicar sobre un evento en forma genérica, dicho de otro modo, la forma de verificar que el sistema cumple la propiedad sería escribir tantos escenarios como jugadores intervengan en el desarrollo de una partida

en particular. Obviamente, esto solamente es posible si se conoce de antemano una cota para este número, en caso contrario el problema ya no es de índole práctica si no que excede las posibildades de la herramienta.

Aún asumiendo que el escenario pudiera ser descrito con las herramientas con las que se cuenta en esta etapa, su aplicabilidad práctica es bastante dudosa, ya que se perdería un aspecto fundamental de la filosofía de VTS, la cual constituye la simplicidad de su notación gráfica.

Parte II

EXTENSIÓN DE VTS

5 EXTENSIÓN DE VTS

5.1 INTRODUCCIÓN

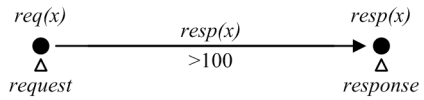
El trabajo presentado en el capítulo anterior nos llevó a la conclusión de que para que VTS resultara más aplicable para la verificación de escenarios a partir de trazas de ejecución de sistemas se necesitaba aumentar su poder expresivo.

En particular, dado que una de las principales limitaciones encontradas en VTS era la imposibilidad de relacionar eventos de forma sencilla, decidimos dotar a los eventos de una semántica más fina mediante la introducción de variables asociadas a los mismos. Mediante este mecanismo se hace posible, por ejemplo, relacionar un evento con el identificador del hilo de ejecución que lo causó y posteriormente predicar sobre otros eventos del mismo hilo. Esto permite expresar escenarios que de otro modo serían sumamente complejos en VTS estándar (cuando no imposibles). Si bien se podría definir estos escenarios recurriendo a la enumeración completa de los conjuntos de valores que las variables pueden asumir; eso requeriría conocer de antemano estos valores, lo que no siempre resulta factible.

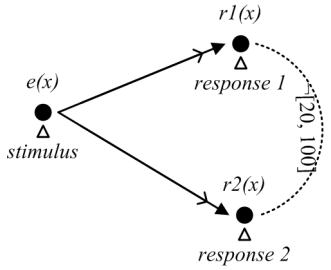
Durante esta sección presentaremos nuestra extensión a eventos con variables asociadas. Veremos la nueva sintaxis gráfica, explicaremos su semántica, analizaremos un caso de estudio y mostraremos los detalles de la implementación realizada. A su vez discutiremos los resultados obtenidos y propondremos futuras mejoras.

5.2 SINTAXIS GRÁFICA DE VTS EXTENDIDO

Al igual que en VTS estándar, un escenario VTS Extendido es básicamente un grafo dirigido y acíclico donde los nodos o puntos representan la ocurrencia de un evento o un instante en el tiempo y los ejes representan dependencias de causalidad entre los puntos. Los puntos pueden etiquetarse (opcionalmente) con uno o más símbolos del alfabeto que representa los eventos posibles en el sistema y en ese caso el punto representa la ocurrencia de uno de dichos eventos durante la ejecución. A su vez, en cada punto, los símbolos que representan a los eventos pueden (también opcionalmente) ir acompañados de una variable arbitraria. Nótese que de esa forma VTS extendido resulta totalmente compatible hacia atrás con VTS estándar, es decir, que un escenario VTS estándar es un caso particular de un escenario VTS Extendido. Dos puntos conectados mediante una flecha indican una precedencia del origen respecto del destino y las flechas, al igual que los puntos, pueden etiquetarse con un conjunto de eventos (opcionalmente acompañados de variables) para indicar, en este caso, que dichos eventos (o eventos con una determinada variable) están prohibidos entre ambos puntos. Al igual que antes, también es posible incluir un intervalo temporal (abierto o cerrado) sobre los ejes para indicar la separación temporal que debe haber entre los puntos de sus extremos. La ausencia de restricciones de causalidad entre dos puntos permite modelar no determinismo (entre eventos de un mismo proceso) y concurrencia



a) Respuesta acotada



b) Respuestas correlacionadas

Figura 7: Ejemplos de escenarios VTS Extendido

(entre eventos de distintos procesos). Los siguientes ejemplos servirán para clarificar estos conceptos.

El escenario de la Fig. 7(a) expresa un predicado que es verdadero en una ejecución si y sólo si la respuesta a un pedido x demora más de 100 unidades de tiempo. Los triángulos debajo de los puntos se emplean para asignarles un nombre opcional. La flecha que une request con response indica que request ocurre antes que response. Para establecer que response es la primer ocurrencia de resp(x) después de request la flecha request-response se etiquetó con resp(x). Nótese que todo esto es idéntico a VTS estándar con la diferencia de que en aquél, al no contar con variables, no era posible asociar un pedido con su correspondiente respuesta. Para ilustrar cuando una determinada ejecución se *ajusta* al escenario VTS de la Fig. 7(a), supongamos que tenemos las siguientes secuencias de eventos:

$$s1 : \ldots \overset{(a,2)}{12}\ \overset{(e,7)}{15}\ \overset{(b,1)}{39}\ \overset{(c,9)}{50}\ \overset{(req,8)}{72}\ \overset{(d,2)}{123}\ \overset{(resp,8)}{140}\ \overset{(f,9)}{148}\ \overset{(z,7)}{155}\ \ldots$$

$$s2 : \ldots \overset{(a,3)}{3}\ \overset{(e,8)}{7}\ \overset{(b,2)}{12}\ \overset{(req,4)}{18}\ \overset{(c,5)}{59}\ \overset{(d,7)}{111}\ \overset{(r,2)}{114}\ \overset{(f,3)}{121}\ \overset{(resp,4)}{125}\ \overset{(z,9)}{152}\ \ldots$$

$$s3 : \ldots \overset{(a,9)}{5}\ \overset{(e,1)}{7}\ \overset{(b,3)}{69}\ \overset{(d,7)}{78}\ \overset{(req,5)}{87}\ \overset{(c,8)}{100}\ \overset{(f,6)}{146}\ \overset{(resp,5)}{152}\ \overset{(z,2)}{199}\ \ldots$$

Entonces las secuencias s1 y s3 no se ajustan al escenario porque la distancia entre los eventos req y resp que aparecen es menor que 100. En cambio, la secuencia s2 se ajusta al escenario y por lo tanto, si esto debiera ser interpretado como un escenario *negativo* o *anti-escenario*, deberíamos descartar aquellos sistemas que produzcan s2 por violar nuestros requerimientos.

La Fig. 8 muestra la notación gráfica completa de VTS Extendido.

5.3 FORMALIZACIÓN

Llamaremos $\mathcal{J}_{\mathbb{N}}$ al conjunto de intervalos delimitados por enteros de los números reales positivos. Una restricción temporal es una fórmula de la forma θ o $\neg\theta$, siendo θ un intervalo en $\mathcal{J}_{\mathbb{N}}$. Φ es el conjunto de todas las restricciones temporales.

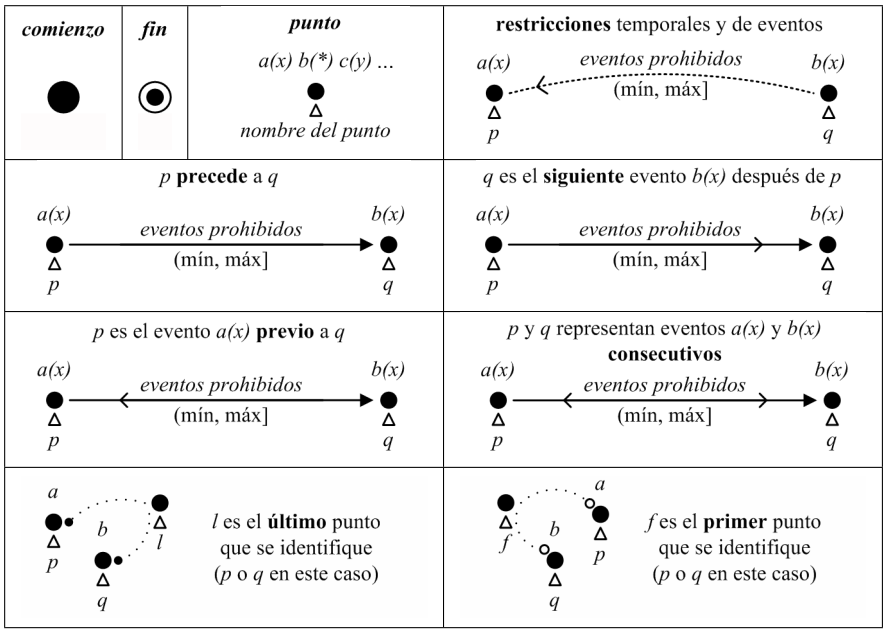


Figura 8: Notación gráfica de VTS Extendido

Definición 1 (Secuencia):

Dado un conjunto C, una *secuencia sobre* C es una secuencia (posiblemente infinita) de elementos de C. Dada una secuencia s, $|s|$ es su longitud (decimos que $|s| \stackrel{def}{=} \infty$ cuando s es infinita). $\Pi(s) = \{i \in \mathbb{N}/0 \leqslant i < |s|\}$ es el conjunto de posiciones de s. Dados $i, j \in \Pi(s)$, si s_i es el elemento i-ésimo de s; $s_{i]}$ es el prefijo que finaliza en la posición i; $s_{[i}$ es el sufijo que comienza en la posición i y $s_{[i,j]}$ es la subsecuencia desde la posición i hasta la j (Si $i > j$,$s[i, j] = s[j, i]$). Usar '(' o ')' en lugar de '[' o ']' significa que los extremos no están incluidos. Llamamos $\mathtt{first}(s)$ al primer elemento de la secuencia y $\mathtt{last}(s)$ al último.

Definición 2 (Secuencia temporal):

Una *secuencia temporal* es una secuencia débilmente incremental de timestamps (números reales no negativos). Dada una secuencia temporal finita τ definimos $\Delta(\tau)$ como el tiempo transcurrido durante τ . $\Delta(\tau) = \mathtt{last}(\tau) - \mathtt{first}(\tau)$ o 0 si $|\tau| = 0$.

Definición 3 (Traza):

Una traza sobre el conjunto $C' = C \times \{\beta \cup \lambda\}$ es un par $\langle s, \tau \rangle$ donde s es una secuencia sobre $C' \cup \{(\lambda, \lambda)\}$ y τ es una secuencia temporal de la misma longitud y β es un conjunto arbitrario de valores. Definimos conjuntamente dos *macros* $\mathtt{evento}(x)$ y $\mathtt{valor}(x)$ que proyectan el primero y segundo elemento de un elemento de s respectivamente.

5.4 SINTAXIS DE VTS EXTENDIDO

Definición 4 (Escenario):

Un escenario VTS extendido es una tupla $\langle \Sigma, P, \ell, <, \gamma, \delta, V \rangle$ donde Σ es un conjunto finito de eventos; P es un conjunto finito de puntos; V es un conjunto finito de nombres de variables;

$\ell : P \to F$ donde F es el conjunto de funciones $\{f / f : \Sigma \to V \cup \{\lambda\}\}$

$<\subseteq P \uplus \{0\} \times P \uplus \{\infty\}$ es una relación de precedencia entre puntos (0 y ∞ representan el comienzo y el final de la ejecución respectivamente).

$\gamma : P \uplus \{0, \infty\} \times P \uplus \{0, \infty\} \to 2^{\Sigma \times V \uplus \lambda}$ asigna a cada par de puntos el conjunto de eventos (y eventualmente nombre de la variable asociada a ese evento) que están prohibidos entre ellos;

y $\delta : P \uplus \{0\} \times P \uplus \{0\} \to \Phi$ asigna a cada par de puntos la restricción para el tiempo transcurrido entre ellos. La clausura transitiva $<^+$debe ser un orden parcial sobre $P \uplus \{0, \infty\}$. Para todos los puntos x e y $\gamma(x, y) = \gamma(y, x)$ y $\delta(x, y) = \delta(y, x)$ debe cumplirse.

Nótese que un escenario VTS extendido es prácticamente idéntico a un escenario VTS, salvo por el agregado del conjunto V de nombre de variables y la consecuente extensión de la definición de etiquetas de puntos y ejes de forma tal de soportar el uso de variables.

Definición 4.1 (Escenario dominado por asignaciones):

Escenario que no incluya restricciones en los ejes que hagan referencia a variables que no estén asociadas a al menos un punto concreto precedente. Formalmente, siendo $S = \langle \Sigma, P, \ell, <, \gamma, \delta, V \rangle$ un escenario VTS extendido:

$$\forall p, q \in S, \forall v \in V, v \in \gamma(p, q), \exists r \in S_{p]} / \forall l \in \ell(r), l = a(v), a \in \Sigma$$

5.5 SEMÁNTICA

Al igual que en VTS, la semántica de VTS extendido asigna a cada escenario un conjunto de trazas que lo satisfacen. Los puntos etiquetados representan eventos en las trazas.

En VTS para asociar una posición de la traza a un punto del escenario basta con que el evento en la traza esté dentro de los permitidos por ese punto en el escenario. En VTS extendido, en cambio, los puntos en la traza están acompañados de un valor; por lo que un punto p del escenario puede ser asociado a una posición en la traza cuyo evento sea a si $a(x) \in \ell(p) \vee a(*) \in \ell(p)$. Además, dado cualquier par de puntos p, q del escenario que predican sobre una misma variable, deben asociarse a posiciones de la traza que presenten el mismo valor.

Los eventos no etiquetados se llaman instantes. Representan momentos en la ejecución que no están necesariamente asociados con algún evento en particular.

Intuitivamente, un apareo es una correspondencia entre puntos en un escenario y posiciones en una traza, con la intención de mostrar cómo la traza satisface al escenario. Dos puntos diferentes deben asociarse a dos posiciones diferentes en la traza.

Definición 5 (Apareo)

Dado un escenario $S = \langle \Sigma, P, \ell, <, \gamma, \delta, V \rangle$, una traza $\sigma =< s, \tau >$ sobre Σ; decimos que $\hat{\cdot}$ (una correspondencia de $\hat{\cdot} : P \to \Pi(\sigma)$) constituye un apareo entre S y σ sii para todo par de puntos $p, q \in P$:

1. $\ell(p) = \emptyset \vee evento\left(s_{\hat{p}}\right) \in Dom(\ell(p))$
2. $\hat{\cdot}$ es inyectiva para $Dom(P)$
3. $p < q \Rightarrow \hat{p} < \hat{q}$
4. $\forall (e, v) \in \gamma(p, q) : \nexists r, (e_2, v) \in \ell(r) \wedge s_{\hat{r}} = e_2 \wedge (e, valor(s_{\hat{r}})) \in \sigma(\hat{p}, \hat{q})$
5. $s_{\hat{p})} \cap \gamma(0, p) = \emptyset \wedge s_{(\hat{p}} \cap \gamma(p, \infty) = \emptyset$
6. $\Delta\left(\tau_{[\hat{p},\hat{q}]}\right) \models \delta(p, q)$
7. $\Delta\left(\tau_{\hat{p}]}\right) \models \delta(0, p)$
8. $variable\left(\ell(p)\left(evento\left(s_{\hat{p}}\right)\right)\right) = variable\left(\ell(q)\left(evento\left(s_{\hat{q}}\right)\right)\right) \Rightarrow valor\left(s_{\hat{p}}\right) = valor\left(s_{\hat{q}}\right)$

Definición 6 (Satisfactibilidad)

Decimos que una traza σ satisface un escenario S (escrito como $\sigma \models S$) sii existe al menos un apareo entre ellos.

5.6 VERIFICACIÓN DE TRAZAS USANDO VTS EXTENDIDO

En esta sección daremos una forma de llegar de un escenario VTS extendido a un autómata temporizado [3] con variables. Dado un escenario mostraremos un *tableau* que reconoce todas las trazas que satisfacen el escenario.

Los autómatas temporizados son un formalismo bien conocido y ampliamente utilizado para el análisis de sistemas temporizados. En el presente trabajo tomaremos la definición básica utilizada en VTS y la dotaremos de variables de modo similar al formalismo empleado por Uppaal, para adecuarla a nuestros propósitos. La semántica de los autómatas temporizados se basa en transiciones de estados determinadas por eventos y corridas sobre ellos. En pocas palabras, un autómata temporizado es una máquina de estados finitos en la que se introducen relojes de modo de poder predicar sobre el tiempo transcurrido entre estados. La composición paralela de autómatas se define utilizando el producto sincrónico de los autómatas. A esta noción le agregaremos el concepto de variable y valor asociado a cada transición, de modo de poder además predicar sobre el valor de las primeras con respecto al observado en cada momento de la corrida.

Dado un modelo de un sistema representado como un autómata A y un escenario S, definimos $A \models S$ sii existe una corrida r de A empezando en el estado inicial de A tal que $r \models S$

Definición 6 (Autómata temporizado con variables).

Un autómata temporizado es una tupla $A = \langle L, X, V, \Sigma, E, I, l_0 \rangle$ donde:

- L es un es un conjunto finito de localizaciones
- X es un conjunto finito de clocks (números reales no negativos).
- V es un conjunto finito de variables.
- Σ es un conjunto finito de eventos.
- E es un conjunto finito de ejes.
- $I : L \to \Psi_X$ es una función total que asocia a cada localización con sus restricciones temporales, llamada el invariante de la localización.
- $l_0 \in L$ es la localización inicial.
- Un eje $E = \langle l, a, \mu, \psi, \kappa, \alpha, l' \rangle$ donde:
  - $l \in L$
  - $a \in \Sigma$
  - μ es una guarda de la forma[1]: $\begin{cases} v = z \\ v! = z \\ true \end{cases}$
  - ψ es el conjunto de restricciones temporales.
  - κ es una asignación de la forma: $\begin{cases} \emptyset \\ v = z \end{cases}$
  - α es el conjunto de clocks a resetear.
  - $l' \in L$

5.6.1 *Transformación del escenario en autómata*

Al igual que en VTS, dado un escenario VTS extendido, se puede construir un tableau (un autómata temporizado no determinístico) de modo de reconocer todas las trazas que coinciden con el escenario en cuestión. La construcción, igual que antes, se basa en la idea de configuraciones -subconjuntos cerrados del orden parcial que conforman los posibles conjuntos de puntos apareados en una corrida-, los cuales son utilizados para etiquetar los estados del autómata. Las transiciones representan extensiones del apareo desde una configuración hacia otra.

Dada la introducción de variables en el escenario y nuestro enfoque para la solución del problema, debemos extender la noción de configuración respecto de la original para mantener el registro de las variables que ya fueron asignadas y sus valores. Esto es necesario para poder determinar qué acción debe realizarse con el valor de la traza al pasar de una configuración a otra (asignación o comparación).

Por motivos de simplicidad algorítmica restringiremos nuestro estudio de la verificación de escenarios de VTS extendido a la subclase de aquellos dominados por asignación (5.4).

Por el resto de la sección, al referirnos al escenario S nos referimos a la tupla $\langle \Sigma, P, l, <, \gamma, \delta, V \rangle$.

1 z representa al valor asociado al punto en la traza

Definición 7 (Configuración)

$\theta = \{P', \$\}$ es una configuración de un escenario S, donde $P' \subseteq P$, $\$: V \to \beta$ es una función parcial que asocia cada variable con el valor que tiene asignado sii:

1. P' es cerrado a izquierda bajo la relación $<^+$
2. $\forall v \in Dom(\$) : \exists p \in P' / v \in \ell(p)$

Dado $p \in P$, se lo considera una extensión de un solo paso por el label $a(z)$ con $z \in \beta$ (denotada $\{P', \$\} \overset{a(z)}{\to} \{P' \cup \{p\}, \$'\}$) sii:

1. $\{P' \cup \{p\}, \$'\}$ es una configuración de S.
2. $a \in Dom(\ell(p)) \wedge \ell(p)(a) = v \Longrightarrow (\$(v) = z \wedge \$ = \$') \vee (v \notin Dom(\$) \wedge \$' = \$ \cup (v, z))$

Decimos que $\theta \overset{\lambda}{\to} \theta \cup \{p\} \iff$ se cumple la condición 1 y $\ell(p) = \emptyset$

Al igual que en VTS, dada una configuración θ decimos que una restricción entre un par de puntos $\langle p, q \rangle$ es activa en $\theta \iff p \in \theta \wedge q \notin \theta$ (por definición, una restricción $\langle 0, p \rangle$ es activa si $p \notin \theta \wedge \langle p, \infty \rangle$ es activa sólo si $p \in \theta$). Llamamos $\Gamma(\theta) = \cup\gamma(p, q)$ para todas las restricciones activas entre p y q en θ. Una restricción por eventos $\langle p, q \rangle$ es estrictamente activa en θ con respecto a un conjunto de puntos F sii es activa y $q \notin F$. Llamamos $\Gamma_{\triangleright F}(\theta) = \cup\gamma(p, q)$ para todas las restricciones estrictamente activas $\langle p, q \rangle$ en θ respecto de un conjunto de puntos F.

Dada una restricción temporal ϕ y un clock x, definimos $\psi_x(\phi)$ como la restricción temporal sobre x tal que para todo número real no negativo t, $\psi_x(\phi)[x \backslash t]$ es verdadero sii $t \models \phi$.

Dado un escenario S y $F \subseteq P$, definimos el conjunto de clocks a resetear dada una extensión por F como $R_F = \{x_p / p \in F \wedge \exists q.\delta(p, q) \neq [0, \infty]\}$. Dada una configuración θ y una extensión F, definimos la restricción temporal que aplica al cambio de configuración como $\psi_\theta^F = \bigwedge_{q \in \theta \cup \{0\}, p \in F} \psi_{xq}(\delta(q, p))$.

Construcción del Tableau

El tableau T_s para el escenario VTS extendido S es un autómata temporizado con variables $\langle L, X, V, \Sigma, E, I, l_0 \rangle$ tal que:

1. $L = \theta / \theta$ es una configuración de $S \cup \{S_{trap}\}$, llamamos S_{accept} a la configuración P que contiene todos los puntos del escenario.
2. $X = \{X_p / p \in P \cup \{0\}\}$

$E = (r_1)\ \{\langle \theta = \{P, \$\}, a, true, \psi_\theta^F, v = z, R_F, \theta' = \{P', \$' = \$ \cup (v, z)\}\rangle / \theta \overset{a(z)}{\to} \theta' \wedge v \notin Dom(\$), a(v) \notin \Gamma_{\triangleright F}(\theta) \wedge a(*) \notin \Gamma_{\triangleright F}(\theta)\}$

$\cup\ (r_2)\ \{\langle \theta = \{P, \$\}, a, v = z, \psi_\theta^F, \emptyset, R_F, \theta' = \{P', \$\}\rangle / \theta \overset{a(z)}{\to} \theta' \wedge v \in Dom(\$), a(v) \notin \Gamma_{\triangleright F}(\theta) \wedge a(*) \notin \Gamma_{\triangleright F}(\theta)\}$

$\cup\ (r_3)\ \{\langle \theta = \{P, \$\}, a, true, \psi_\theta^F, \emptyset, R_F, \theta' = \{P', \$\}\rangle / \theta \overset{a(z)}{\to} \theta' \wedge a(*) \notin \Gamma_{\triangleright F}(\theta) \wedge a(v) \notin \Gamma_{\triangleright F}(\theta)\}$

$\cup\ (r_4)\ \{\langle \theta, \lambda, true, \psi_\theta^F, \emptyset, R_F, \theta' \rangle / \theta \overset{\lambda}{\to} \theta'\}$

$\cup\ (r_5)\ \left\{\langle \theta, a, v \neq z, \psi_\theta^F, \emptyset, R_F, \theta' \rangle / \theta \overset{a(z)}{\to} \theta' \wedge v \in Dom(\$) \wedge a(v) \in \Gamma_{\triangleright F}(\theta)\right\}$

$\cup\ (r_6)\ \{\langle \theta, a, v \neq z, true, \emptyset, \emptyset, \theta \rangle / a(*) \notin \Gamma(\theta) \wedge v \in Dom(\$) \wedge a(v) \in \Gamma(\theta)\}$

$\cup\ (r_7)\ \{\langle \theta, a, true, true, \emptyset, \emptyset, \theta \rangle / a \notin \Gamma(\theta)\}$

$$\cup\ (r_8)\ \{\langle \theta, a, v = z, true, \emptyset, \emptyset, trap\rangle / a\ (v) \in \Gamma\ (\theta) \wedge v \in Dom(\$)\}$$
$$\cup\ (r_9)\ \{\langle \theta, a, true, true, \emptyset, \emptyset, trap\rangle / a\ (*) \in \Gamma\ (\theta)\}$$
$$\cup\ (r_{10})\ \{\langle trap, a, true, true, \emptyset, \emptyset, trap\rangle / a \in \Sigma\}$$
$$\cup\ (r_{11})\ \{\langle \theta, a, true, true, v = z, \emptyset, trap\rangle / a\ (v) \in \Gamma\ (\theta) \wedge v \notin Dom(\$)\}$$

5.6.2 *Algoritmo para la construcción del Tableau*

A continuación mostraremos un algoritmo constructivo que implementa las reglas de construcción del Tableau analizadas en la sección anterior partiendo del siguiente autómata temporizado:

$A = \langle L, X, V, \Sigma, E, I, l_0\rangle$ donde:

- $L = \emptyset$
- $X = \emptyset$
- V es un conjunto finito de variables.
- Σ es un conjunto finito de eventos.
- $E = \emptyset$
- $I = \emptyset$

Dado que durante la corrida del autómata generado las variables se asignan en orden de aparición, sólo podemos garantizar la correctitud de la verificación para escenarios dominados por asignación 5.4.

```
for p ∈ P:
    X = X ∪ {x_p}
X = X ∪ {x_0}
L = L ∪ {trap}
for a ∈ Σ:
    E = E ∪ {⟨trap, a, true, ∅, true, trap⟩}
if 0 ∈ P:
    l_0 = ([0], {})
else:
    l_0 = ([], {})

L = L ∪ {l_0}

ConfiguracionesActivas = {l_0}
while ConfiguracionesActivas ≠ ∅:
    c = ConfiguracionesActivas.pop()
    RA = restriccionesActivas(c)
    for a(v) ∈ RA:
        E = E ∪ {⟨c, a, v ≠ z, true, ∅, c⟩}
        E = E ∪ {⟨c, a, v = z, true, ∅, trap⟩}
    for a ∉ RA:
        E = E ∪ {⟨c, a, true, true, ∅, c⟩}
    if a(*) ∈ RA:
        E = E ∪ {⟨c, a, true, true, ∅, trap⟩}
    S = Siguientes(c)

    for t ∈ S:
        if t ≠ ∞:
            ConfiguracionesActivas = ConfiguracionesActivas ∪ {(c.t)}

        L = L ∪ (c.t)
        REA = RestriccionesEstrictamenteActivas(c, t)
```

```
        RT = RestriccionesTemporales(c, t)
        R = ClocksAResetear(t)
        W = Labels(t)

        for w ∈ W:
            if w = a(v) ∧ a(v) ∉ REA ∧ a(∗) ∉ REA
                ∧a ∈ Dom(ℓ(punto(t))):
                if variable(t) = v:
                    E = E ∪ {⟨c, a, true, RT, v := z, R, c.t⟩}
                if variable(t) = λ ∧ $(v) ≠ ⊥:
                    E = E ∪ {⟨c, a, v = z, RT, ∅, R, c.t⟩}
            if w = a(∗) ∧ a(∗) ∉ REA ∧ a ∈ Dom(ℓ(punto(t))):
                E = E ∪ {⟨c, a, true, RT, R, c.t⟩}
            if w = λ:
                if punto(t) ≠ ∞:
                    E = E ∪ {⟨c, λ, true, RT, R, c.t⟩}
                else:
                    for a(v) ∈ RA:
                        if v ≠ λ:
                            E = E ∪ {⟨c, a, z ≠ v, RT, R, c.t⟩}
                        else:
                            E = E ∪ {⟨c, a, true, RT, R, trap⟩}
                    for a ∉ RA:
                        E = E ∪ {⟨c, a, true, RT, R, c.t⟩}
```

Función Siguientes

Toma como entrada: una configuración y un escenario (S) y devuelve un conjunto de tuplas (Punto, Variable a asignar).

```
Dev = ∅
for p ∈ P/predecesores(p) = ∅ ∧ p ∉ c
    V = Variables(p)
    for v ∈ V/v ∉ Variables(c):
        Dev = Dev ∪ {(p, v)}
    for v ∈ V/v ∈ Variables(c):
        Dev = Dev ∪ {(p, λ)}
    for p ∈ Puntos(c):
        N = SiguientesInmediatos(p, S)
        for q ∈ N:
            V = Variables(q)
            for v ∈ V/v ∉ Variables(c):
                Dev = Dev ∪ {(q, v)}
            for v ∈ V/v ∈ Variables(c):
                Dev = Dev ∪ {(q, λ)}
return Dev
```

5.6.3 *Teorema de verificación de escenarios VTS*

En esta sección demostraremos que el Tableau construido siguiendo las reglas que enunciamos previamente reconoce las trazas que satisfacen el escenario y sólo éstas.

Para realizar esta demostración probaremos que las siguientes condiciones son imposibles por construcción del autómata:

1. Existe una traza t que satisface S y no es reconocida por el Tableau
2. Existe una traza reconocida por el Tableau que no satisface S

Condición 1

Para realizar esta demostración partiremos del supuesto de que existe una traza t que satisface S y realizaremos una inducción sobre t:

Caso base:

Estando el autómata generado en el estado que contiene la configuración vacía ($\theta = \{[\,], [\,]\}$) se toma del primer elemento de la traza (el par (a, z)) el evento y su valor asociado.

Las reglas que pueden aplicarse a este caso son: r_1, r_3, r_7, r_9, r_{11}.

Existen tres posibles movimientos para el autómata:

1. $a(z) \in \Gamma(\theta)$ por lo tanto, se avanza hacia el estado $trap$ (por r_9, r_{11})
2. $a(z) \notin \Gamma(\theta)$ y no se produce un cambio de configuración (por r_7).
3. $a(z) \notin \Gamma(\theta)$, se consume el evento y se produce un cambio de configuración (por r_1, r_3).

El primer caso no puede suceder por hipótesis.

Los casos 2 y 3 dejan siempre al autómata en un estado válido por hipótesis.

Paso inductivo:

Hipótesis inductiva: El autómata se encuentra en una configuración válida luego de haber consumido $i - 1$ elementos de la traza. En este punto, ya que potencialmente se han definido variables, todas las reglas de construcción son susceptibles de ser aplicadas.

Se procederá a consumir el elemento t_i de la traza ((a, z)) y, al igual que antes, existen tres posibles movimientos para el autómata:

1. $a(z) \in \Gamma(\theta)$ por lo tanto, se avanza hacia el estado $trap$ (por r_8 a r_{11}).
2. $a(z) \notin \Gamma(\theta)$ y no se produce un cambio de configuración (por r_6 y r_7).
3. $a(z) \notin \Gamma(\theta)$, se consume el evento y se produce un cambio de configuración (por r_1, r_2, r_3 , r_4 y r_5).

El caso 1 resulta irrelevante dada la hipótesis inductiva.

En el caso 2 el elemento t_i simplemente será ignorado por el autómata.

En el caso 3, siendo p el punto que se agrega en la extensión, si p es un *instante* entonces se aplica la regla r_4. Caso contrario, se aplican las reglas r_1, r_2, r_3 o r_5. Esto garantiza que el autómata tendrá los ejes necesarios para cambiar de la configuración actual a una nueva configuración válida incorporando a p al apareo existente y por lo tanto, por hipótesis inductiva, eventualmente el estado que contiene todos los puntos de S (o sea $accept$) será alcanzable.

Condición 2

Para que se cumpla esta condición debe suceder que el Tableau dé como resultado un apareo (m) incorrecto.

Por hipótesis existe un primer elemento de la traza incorrectamente apareado (t_i). Es decir, se está realizando una extensión de un solo paso que viola alguna de las reglas.

Para realizar esta demostración realizaremos una inducción sobre m:

Caso Base:

Estando el autómata generado en el estado que contiene la configuración vacía ($\theta = \{[], []\}$) se toma del primer elemento de $m(\{t_x, p\}, \$)$. Dado que no existen variables asignadas en la configuración inicial, las reglas aplicables son:r_1 y r_3, las cuales son inaplicables si no se cumplen las condiciones que permiten realizar la extensión o se violan las restricciones (activas o temporales), por lo tanto, al utilizar estas reglas es imposible aparear incorrectamente t_x.

Paso inductivo:

Estando el autómata generado en el estado que contiene la configuración c_{i-1} vemos que no es posible pasar de la configuración c_{i-1} a c_i (siendo c_i una configuración inválida) aplicando las reglas de construcción del Tableau:

Dado que la inducción realizada es sobre m, no tiene sentido tomar las reglas que no generan cambios de configuración (r_6 y r_7). A su vez, al suponer t es reconocida por el Tableau, no se analizarán las reglas que derivan en $trap$ (r_8 a r_{11}).

r_1y r_3 ya fueron analizadas en el caso base.

En el caso de r_2 se aplican las mismas restricciones que en r_1 sólo que la variable sobre la que se predica (v) debe estar asignada y su valor ($\$(v)$) debe coincidir con el establecido, por lo tanto, no es posible aparear $t_i(a, z)$ si z es diferente de $\$(v)$ utilizando esta regla.

r_4 es la regla utilizada para el apareo de instantes en el escenario, por lo tanto, las únicas restricciones a tener en cuenta son de índole temporal y éstas están expresamente consideradas en la regla, por lo tanto, no es posible aparear incorrectamente t_i utilizando esta regla.

Para poder aparear $t_i(a, z)$ utilizando r_5 debe existir una restricción activa para $a(v)$ respecto a c_{i-1} y la variable v debe estar definida (y $\$(v)! = z$) por lo tanto, no es posible realizar un apareo incorrecto utilizando esta regla.

Dado que no existen más reglas que las analizadas y que todas las reglas de construcción del Tableau que realizan cambios de configuración, lo hacen utilizando cambios válidos el Tableau no puede alcanzar el estado $accept$ cuando la traza no satisface S.

Conclusión

Queda demostrado que ni la condición 1 ni la 2 son posibles y por lo tanto, el Tableau construido siguiendo estas reglas es en efecto válido para ser utilizado como medio de verificación de las trazas del escenario.

6 IMPLEMENTACIÓN DE VTS EXTENDIDO

En este capítulo mostramos las herramientas que desarrollamos como prueba de concepto de nuestra extensión de VTS al uso de variables.

Cabe aclarar que nuestro esfuerzo se enfoca en la extensión del lenguaje VTS, con lo cual sólo reemplazamos las herramientas que son específicas a la versión original de VTS (VTSTranslator y ObsSlice), mientras que conservamos Uppaal para la verificación ya que su soporte de variables es adecuado para nuestro trabajo.

Entre las herramientas que presentamos se encuentran:

- Extensión de la plantilla Visio para la generación de escenarios extendidos con variables
- Vts2Uppaal: una herramienta que a partir de un escenario VTS Extendido genera un autómata temporizado en formato Uppaal implementando el algoritmo presentado en el capítulo 5

6.1 EXTENSIÓN DE LA PLANTILLA VISIO

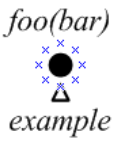
El front-end de VTS está implementado mediante una plantilla MS-Visio. Parte de nuestro trabajo consistió en la adaptación de esta plantilla a las necesidades de nuestras adiciones al lenguaje.

Se introduce el concepto de variable del escenario. Una variable en un escenario es un nombre, el cual pertenece a un espacio de nombres separado de aquel de los eventos. Los eventos y variables pueden combinarse para expresar en forma visual etiquetas de los puntos y ejes de los escenarios extendidos. Dentro de la notación visual, un evento que debe venir acompañado de una variable se expresará utilizando la siguiente sintaxis:

```
<evento>[(<variable>)]
```

Esta notación puede utilizarse tanto en los puntos del escenario (para especificar los eventos que deben coincidir con posiciones de la traza) como en los ejes, para especificar eventos prohibidos entre dos puntos relacionados.

De este modo, un punto con evento asociado se verá de esta manera:



Y una restricción entre dos puntos se verá así:

adaptarla al uso de variablesa la verificación de sistemas críticos a partir de trazas de ejecución en una plataforma de tiempo real (Windows CE). Comenzamos con una introducción al lenguaje VTS y su notación gráfica ilustrándola con escenarios de ejemplo de su aplicación. Continuamos con una breve descripción del conjunto de herramientas que empleamos para la edición de escenarios, traducción

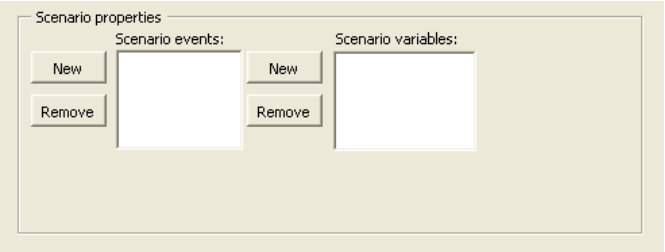


Figura 9: Formulario de edición de propiedades del escenario

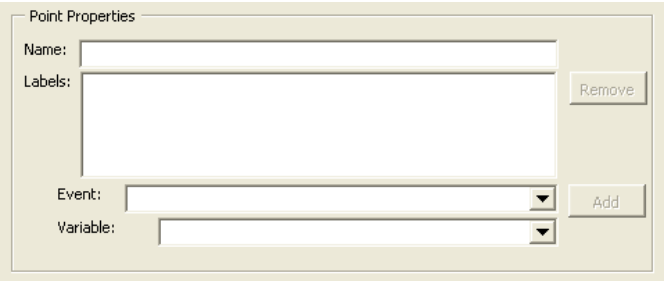
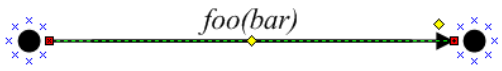


Figura 10: Formulario de edición de propiedades de puntos

de los escenarios en autómatas observadores, composición de dichos autómatas con los generados a partir de trazas de ejecución y verificación de que dichas trazas satisfacen (violan) las propiedades expresadas en los



escenarios

La edición de formularios en VTS++ introduce algunas modificaciones a aquellos que se encuentran en la versión original. Básicamente se trata de tres modos de edición:

- General del escenario: se accede haciendo doble click sobre una parte vacía del escenario
- Edición de puntos: se accede haciendo doble click sobre un punto en particular
- Edición de precedencias: se accede haciendo doble click sobre alguna precedencia en particular

El formulario que se utiliza para los tres casos contiene una sección en común (La superior), en la cual se pueden editar las propiedades generales del escenario (Fig. 9).

En esta sección se puede:

- Editar los eventos que estarán presentes durante todo el escenario
- Editar las variables que se utilizarán en conjunción con los eventos

La Fig. 10 muestra el formulario que se presenta en la parte inferior para el caso de la edición de puntos del escenario. En esta sección se puede configurar el nombre del punto, asociarle eventos y, opcionalmente, una variable a cada uno de ellos.

Finalmente, la Fig. 11 muestra como se verá la parte inferior para el caso de la edición de precedencias.

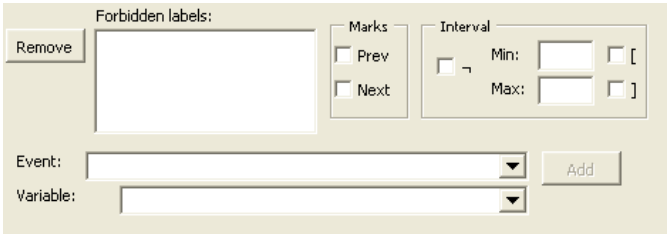


Figura 11: Formulario de edición de propiedades de precedencias

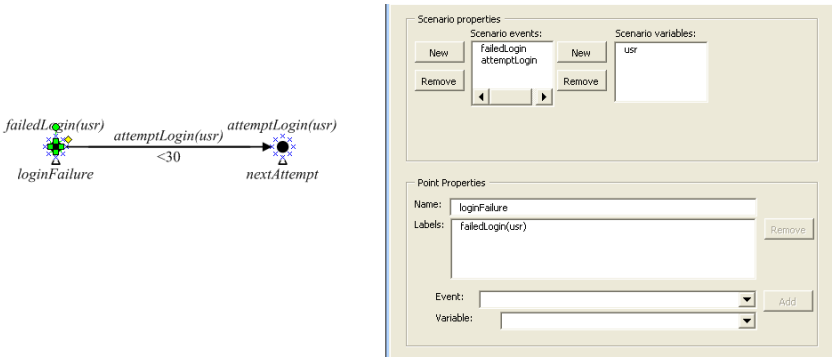


Figura 12: Edición del escenario para el caso: Acceso Inválido

En esta sección se pueden editar los pares <evento, variable> prohibidos para esta precedencia, marcar restricciones temporales y definir si se trata de la próxima ocurrencia (o la previa).

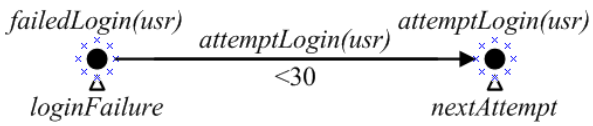
6.1.1 *Ejemplo - Acceso inválido*

En la presente sección veremos un ejemplo práctico de aplicación de la extensión de VTS propuesta. Se trata de un escenario sencillo para especificar ciertas propiedades que debe cumplir un sistema que contenga un módulo de autenticación de usuarios, por ejemplo alguna aplicación web.

La propiedad que queremos verificar puede expresarse informalmente como:

> *Luego de un intento de acceso fallido, un mismo usuario no debería poder reintentar acceder hasta que no haya pasado un intervalo de tiempo determinado*

Esta propiedad puede expresarse en VTS extendido mediante el siguiente escenario:



La Fig. 12 muestra como se vería la edición de un punto de este escenario empleando la herramienta de edición de VTS Extendido. Puede apreciarse que en la parte superior de la pantalla aparecen los eventos (failedLogin y attemptLogin) y las variables (usr) disponibles en el escenario.

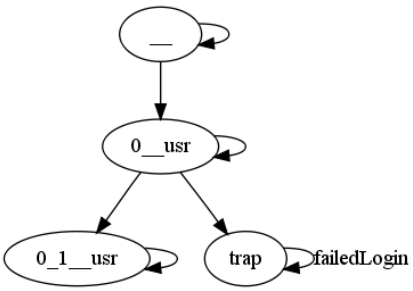


Figura 13: Tableau para escenario Acceso Inválido

El Tableau resultante es el siguiente:

Es importante notar aquí que este tipo de propiedades no son expresables en la versión original de VTS si el universo de valores que puede tomar el nombre de usuario no es conocido de antemano (Y aún bajo este supuesto, si el mismo fuera suficientemente grande, la cantidad de escenarios diferentes a ser verificados haría que el enfoque se torne efectivamente impracticable).

6.1.2 *Implementación del Stencil*

El Stencil está programado utilizando el lenguaje de Macros VBA (Visual Basic for Aplications), el stándard del paquete Office para personalizar la funcionalidad de las aplicaciones que lo componen. Nuestra implementación fue una serie de extensiones al código ya existente y un rediseño de la interfaz para dotarlo de la posibilidad de editar escenarios de VTS con variables.

Para ello, implementamos las siguientes clases para resolver la funcionalidad del núcleo de la aplicación:

- EvSet: Conjunto de eventos propios del escenario.
- VarSet: Conjunto de variables definidas en el escenario.
- Label: Abstracción sobre un par Evento - Variable (Una etiqueta).
- LabelSet: Conjunto de etiquetas del escenario.

Se utiliza a su vez un módulo de abstracción de las herramientas de base que provee MS Visio para almacenar y recuperar la información relevante a VTS en forma de propiedades de los Shapes del dibujo. Cabe destacar que la API provista por MS Visio no está diseñada para este tipo de usos, con lo cual su implementación es bastante engorrosa.

Para resolver los aspectos de interfaz de usuario se utilizan las siguientes clases:

- frmAddEvent: Formulario para agregar eventos al escenario.
- frmAddVar: Formulario para agregar variables al escenario.
- frmProps: Formulario para editar las propiedades de los objetos seleccionados (Puntos o restricciones).

6.2 VTS2UPPAAL

En esta sección se describe otra de las herramientas desarrolladas durante el transcurso de esta tesis. VTS2Uppaal constituye, probablemente,

la herramienta más importante del trabajo, dado que es la que permite completar el paso de la especificación teórica a la verificación práctica de trazas de ejecución, obteniendo un verdadero valor agregado. Básicamente se trata de una herramienta de línea de comandos que toma como entrada una especificación de escenario VTS extendido (en formato XML) y una serie de trazas de ejecución (también escritas en XML) y genera un autómata Uppaal, mediante el cual es posible determinar si la propiedad a verificar es satisfecha por cada una de las trazas a ser evaluadas.

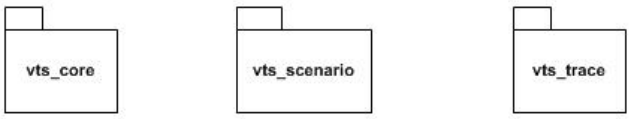
6.2.1 *Lenguaje de implementación*

El lenguaje utilizado para su implementación es Python. Lo elegimos por las siguientes razones:

- Al tratarse de un lenguaje interpretado es posible correrlo en una amplia gama de plataformas
- Se trata de un lenguaje de objetos, lo cual facilita una codificación clara y moderna
- El manejo de conjuntos provisto en forma nativa por el lenguaje se adapta muy bien a las necesidades de nuestro algoritmo

6.2.2 *Diseño de la aplicación*

La aplicación se compone de tres paquetes:



Dentro del paquete vts_core se encuentran las clases que modelan los conceptos principales de VTS extendido, a saber:

- Restricciones temporales
- Etiquetas
- Eventos
- Variables
- Configuraciones

Dentro del paquete vts_scenario encontramos los elementos propios de la definición de un escenario:

- Puntos
- Precedencias
- Escenarios

Por último, dentro de vts_trace se hallan las clases que modelan las trazas de ejecución:

- Marcas de tiempo

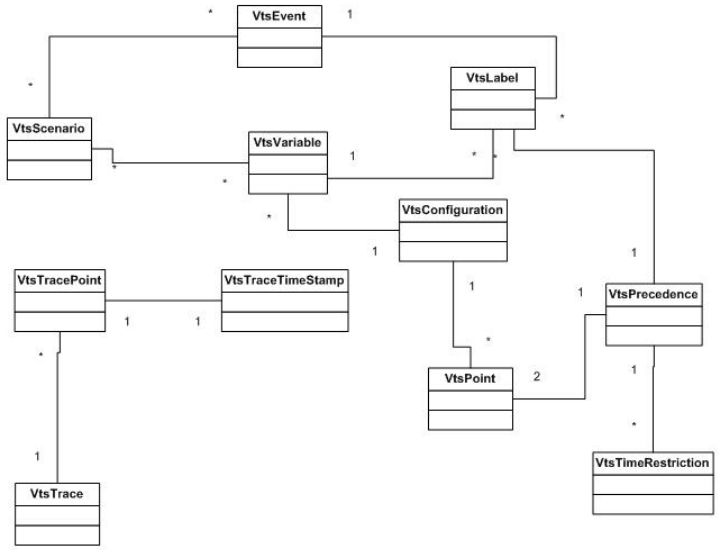


Figura 14: Diagrama de clases de VTS2Uppaal

- Puntos de la traza
- Trazas

La Fig. 14 muestra el diagrama de clases de VTS2Upaal.

La lógica principal se encuentra en el método getAutomata de la clase VtsScenario, el cual implementa el algoritmo de construcción del Tableau analizado previamente.

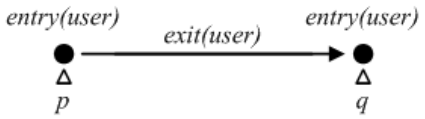
A su vez, VTS2Uppaal traduce las trazas en autómatas Uppaal. Para realizar esta traducción hay que considerar que Uppaal sólo maneja variables con valores enteros de 2 bytes, lo que implica que los valores de las trazas deben ser mapeados a números enteros y que existe un límite a la cantidad máxima de valores diferentes que pueden ser utilizados.

6.2.3 *Ejemplo: Acceso inválido*

Durante esta sección tomaremos un caso práctico a modo de ejemplo de cómo se comporta VTS2Uppaal.

El ejemplo elegido es un sistema de control de acceso a un edificio. Los usuarios tienen una tarjeta identificatoria, la cual deben pasar por el lector de modo de habilitar la entrada al edificio. Al igual, para poder salir del edificio deben pasar la tarjeta por el lector.

Para mostrar la utilización de VTS2Uppaal diseñaremos un escenario sencillo utilizando el Stencil VTS modificado para este efecto, de modo de generar el archivo XML que será utilizado como entrada del aplicativo:



Con este escenario se pretende chequear la siguiente propiedad:

Una vez que un usuario entró no puede volver a entrar *sin haber salido previamente*

A su vez, utilizaremos dos trazas (una que valida el escenario y otra que lo viola) especialmente construidas a los efectos de servir como demostración de las posibilidades de VTS2Uppaal. Contamos con dos archivos de trazas diferentes, uno que valida el escenario y otro que lo viola, de modo de probar ambas posibilidades de VTS2Uppaal:

Archivo válido:

```
<trace>
        <event name="entry" value="mauro" begin="1" end="1"/>
        <event name="entry" value="mauro" begin="5" end="5"/>
        <event name="exit" value="mauro" begin="7" end="7"/>
</trace>
```

Archivo inválido:

```
<trace>
        <event name="entry" value="mauro" begin="3" end="3"/>
        <event name="exit" value="mauro" begin="5" end="5"/>
        <event name="entry" value="mauro" begin="7" end="7"/>
</trace>
```

Debe tomarse en cuenta que dado que los escenarios utilizados son negativos, la propiedad no será satisfecha por el sistema si existe una traza que valide el escenario, de modo que el archivo 1 presenta un caso de ejecución incorrecta del sistema y el 2 uno de ejecución satisfactoria.

Para correr el programa podemos utilizar un comando similar a:

```
vts2uppaal noEntryWithoutExit.vdx noEntryWithoutExit-Valid Valid.
    xml
```

obteniendo como resultado un archivo de autómata que utilizaremos con Uppaal para realizar la verificación de la propiedad requerida.

Como una forma de automatizar la tarea de verificar escenarios y trazas construimos una herramienta que permite realizar todo en un sólo paso aprovechando las capacidades de verificación de modelos de Uppaal a través de la interfaz de línea de comandos:

```
testvts2uppaal noEntryWithoutExit.vdx Valid.xml ./Output/
```

Como salida se obtiene el resultado de la verificación (por salida estándar), el autómata en formato Uppaal (archivo xml), la consulta que se le pasa a Uppaal para verificar la alcanzabilidad del estado S_{accept}, el log de la ejecución de Uppaal, la salida de Uppaal (archivo .res) y, si se ejecuta en un entorno que soporte la biblioteca graphviz (por ejemplo Linux), se obtiene también una representación gráfica del Tableau en formato png.

7 CASO DE ESTUDIO: CONTROL DE ACCESOS A EDIFICIO

Durante este capítulo retomaremos el ejemplo que mostramos en el anterior y lo estudiaremos en mayor profundidad, diseñando diferentes escenarios y trazas para los mismos. Nuestro objetivo es mostrar la eficacia de la extensión realizada a VTS para verificar escenarios reales, eventualmente complejos, de forma sencilla y clara.

7.1 ESCENARIOS

Durante esta sección presentaremos, para cada propiedad que deseamos verificar, una definición informal, un escenario correspondiente escrito en VTS++, un ejemplo de traza que la valide y uno que la viole.

Una característica importante de señalar es que el autómata generado por Vts2Uppaal para las trazas de ejecución fuerza a Uppaal a realizar una verificación secuencial de la traza y, por lo tanto, el orden en que aparecen los eventos significativos para cada escenario influye en el tiempo que demora en dar una respuesta. Por esta razón, las trazas fueron diseñadas de forma tal de que el contraejemplo buscado se encuentre siempre lo más cerca del final posible para permitir analizar la influencia del tamaño de la traza en el tiempo de verificación.

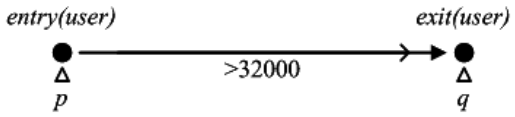
7.1.1 *Escenario 1 - MaxTimeInside*

Con este escenario buscaremos asegurar que el tiempo de permanencia dentro del edificio para un usuario dado es acotado.

La propiedad expresada será:

"Un usuario no puede permanecer en el edificio más de X unidades de tiempo" [1]

El escenario que utilizaremos será el siguiente:



Ejemplo de traza válido:

```
<trace>
        <event name="entry" value="mauro" begin="1" end="1"/>
        <event name="entry" value="pablo" begin="2" end="2"/>
        <event name="exit" value="mauro" begin="18000" end
            ="18000"/>
        <event name="exit" value="pablo" begin="18300" end
            ="18300"/>
</trace>
```

Ejemplo de traza inválido:

1 *Para el caso del escenario VTS++ deberemos definir X como un valor constante arbitrario, queda claro que la elección de un valor en particular no influye en la generalidad del ejemplo utilizado.*

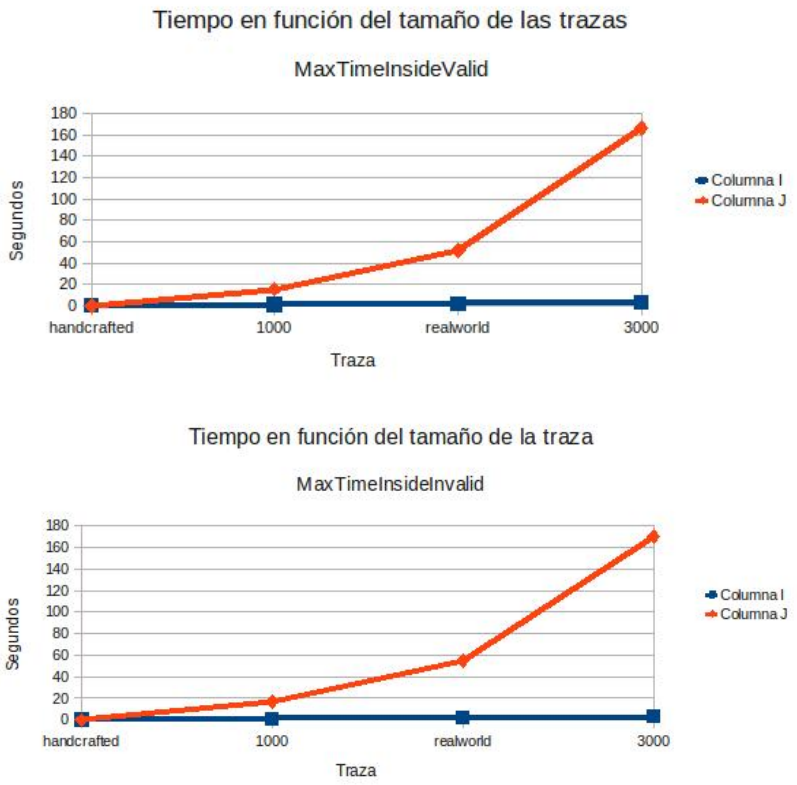


Figura 15: MaxTimeInside

```
<trace>
        <event name="entry" value="mauro" begin="1" end="1"/>
        <event name="entry" value="pablo" begin="2" end="2"/>
        <event name="exit" value="mauro" begin="18000" end
            ="18000"/>
        <event name="exit" value="pablo" begin="35000" end
            ="35000"/>
</trace>
```

7.1.2 *Escenario 2 -MinTimeBetweenEntries*

Con este escenario buscaremos asegurar que el tiempo transcurrido entre una entrada de un usuario y la siguiente entrada del mismo usuario no se produzca en menos tiempo del esperado por el sistema (Si esto sucediera se podría tratar de un ataque informático contra el sistema).

La propiedad expresada será:

"El tiempo transcurrido entre dos entradas de un mismo usuario no podrá ser menor que X unidades de tiempo" [2]

El escenario que utilizaremos será el siguiente:

2 *Para el caso del escenario VTS++ deberemos definir X como un valor constante arbitrario, queda claro que la elección de un valor en particular no influye en la generalidad del ejemplo utilizado.*

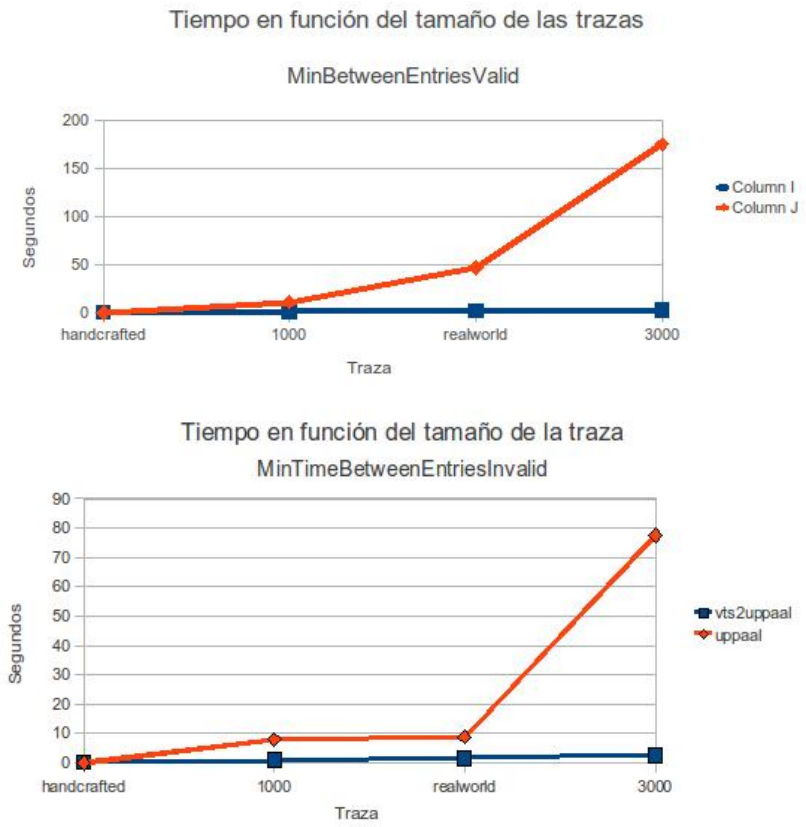
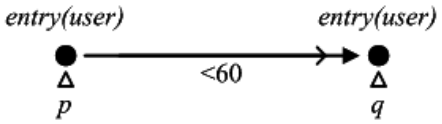


Figura 16: MinTimeBetweenEntries



Ejemplo de traza válido:

```
<trace>
        <event name="entry" value="mauro" begin="1" end="1"/>
        <event name="entry" value="mauro" begin="200" end="200"/>
</trace>
```

Ejemplo de traza inválido:

```
<trace>
        <event name="entry" value="mauro" begin="1" end="1"/>
        <event name="entry" value="mauro" begin="2" end="2"/>
</trace>
```

En la Figura 16 se pueden ver los tiempos de corrida para trazas de diferentes tamaños.

7.1.3 *Escenario 3 -NoEntryInvalidUser*

Con este escenario buscaremos asegurar que un usuario que ha sido desautorizado a ingresar al edificio es efectivamente rechazado por el sistema de control de acceso.

La propiedad expresada será:

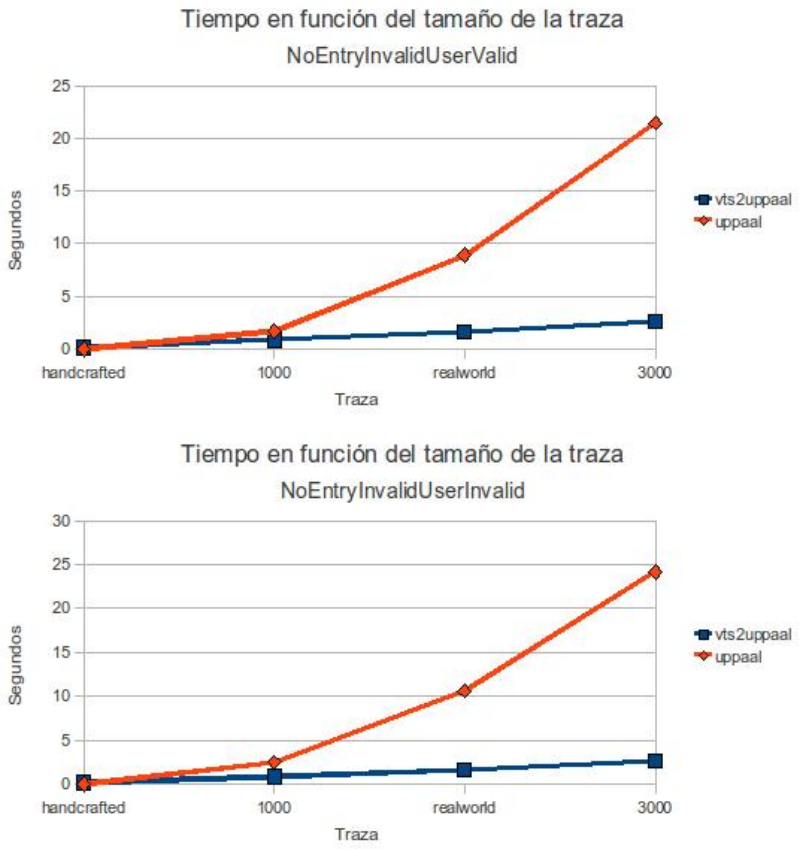
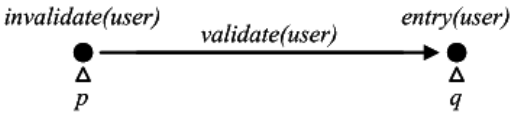


Figura 17: NoEntryInvalidUser

"Luego de que un usuario ha sido invalidado no podrá ingresar al sistema hasta tanto haya sido validado nuevamente"

El escenario que utilizaremos será el siguiente:



Ejemplo de traza válido:

```
<trace>
        <event name="invalidate" value="mauro" begin="1" end="1"/>
        <event name="validate" value="mauro" begin="3" end="3"/>
        <event name="entry" value="mauro" begin="5" end="5"/>
</trace>
```

Ejemplo de traza inválido:

```
<trace>
        <event name="invalidate" value="pablo" begin="1" end="1"/>
        <event name="entry" value="pablo" begin="3" end="3"/>
        <event name="invalidate" value="pablo" begin="4" end="4"/>
</trace>
```

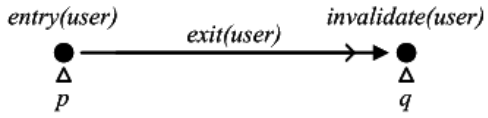
7.1.4 *Escenario 4 -NoInvalidationWithUserInside*

Con este escenario buscaremos asegurar que un usuario no puede ser inhabilitado a usar el sistema mientras aún se encuentra dentro del edificio (Debe permitírsele la salida al menos una vez más).

La propiedad expresada será:

"Una vez que un usuario se encuentra dentro del edificio no puede ser invalidado hasta tanto haya salido"

El escenario que utilizaremos será el siguiente:



Ejemplo de traza válido:

```
<trace>
        <event name="entry" value="mauro" begin="3" end="3"/>
        <event name="exit" value="mauro" begin="3" end="3"/>
        <event name="invalidate" value="mauro" begin="5" end="5"/>
</trace>
```

Ejemplo de traza inválido:

```
<trace>
        <event name="entry" value="pablo" begin="1" end="1"/>
        <event name="invalidate" value="pablo" begin="3" end="3"/>
        <event name="exit" value="pablo" begin="5" end="5"/>
</trace>
```

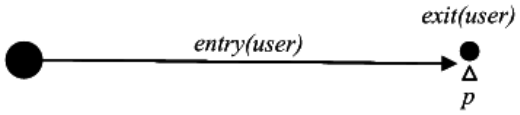
7.1.5 *Escenario 5 -NoExitWithoutEntry*

Con este escenario buscaremos asegurar que un usuario no puede salir del edificio antes de haber entrado.

La propiedad expresada será:

"Un usuario no puede salir sin antes haber entrado"

El escenario que utilizaremos será el siguiente:



Ejemplo de traza válido:

```
<trace>
        <event name="entry" value="mauro" begin="3" end="3"/>
        <event name="exit" value="mauro" begin="5" end="5"/>
</trace>
```

Ejemplo de traza inválido:

```
<trace>
        <event name="entry" value="pablo" begin="1" end="1"/>
        <event name="exit" value="mauro" begin="3" end="3"/>
</trace>
```

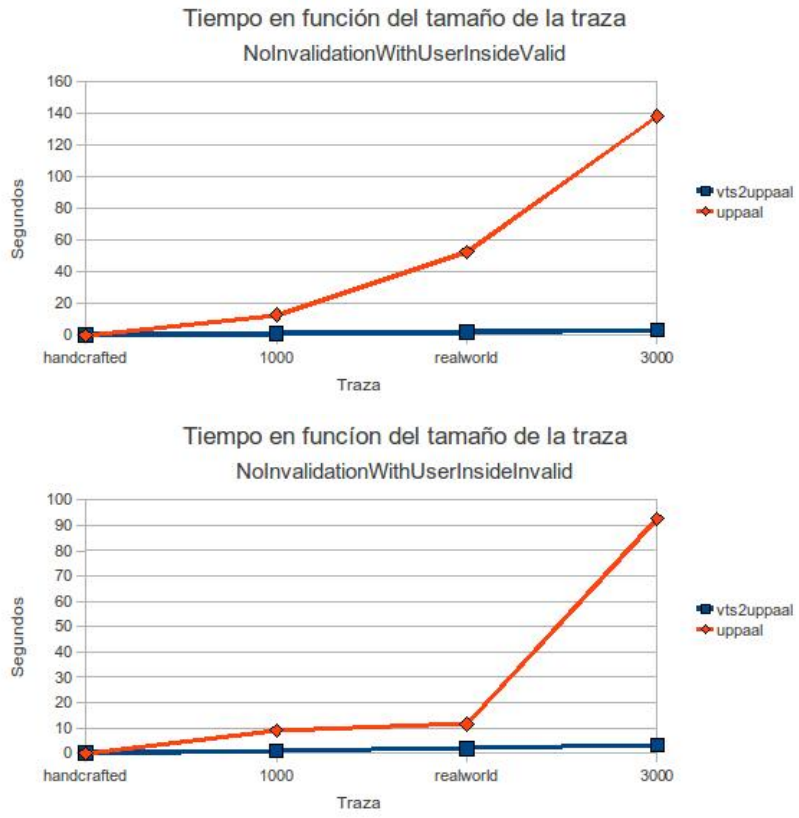


Figura 18: NoInvalidationWithUserInside

Este escenario es un ejemplo sumamente particular, ya que pone en evidencia un problema de posible indeterminación subyacente a la implementación del buscador de apareos, debido a que se estará pidiendo al verificador que compare el valor asociado a un evento con una variable que aún no ha sido definida.

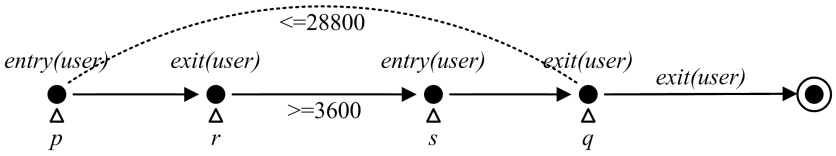
7.1.6 *Escenario 6 -MaxLunchTime*

Con este escenario buscaremos asegurar que un usuario no puede exceder el tiempo del almuerzo permitido.

La propiedad expresada será:

"El tiempo transcurrido entre el momento en que un usuario comienza su almuerzo y lo finaliza no puede exceder X"

El escenario que utilizaremos será el siguiente:



Ejemplo de traza válido:

```
<trace>
        <event name="entry" value="mauro" begin="60" end="60"/>
        <event name="entry" value="pablo" begin="120" end="120"/>
        <event name="exit" value="mauro" begin="1200" end="1200"/>
        <event name="entry" value="mauro" begin="2250" end="2250"/>
```

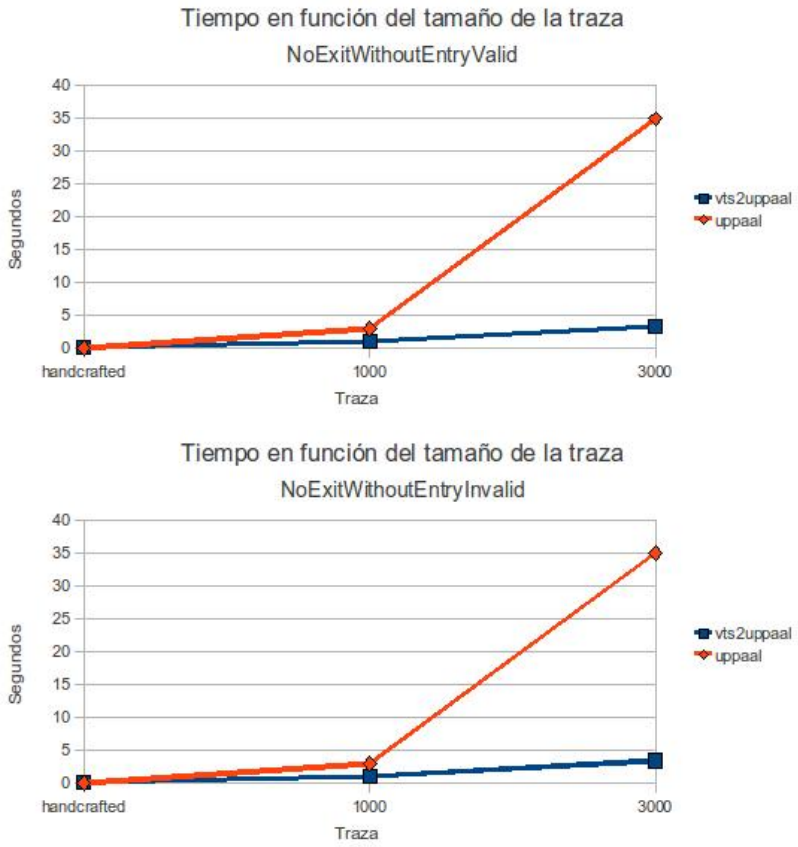


Figura 19: NoExitWithoutEntry

Figura 20: MaxLunchTime

```
        <event name="exit" value="pablo" begin="30000" end="30000"/
            >
        <event name="exit" value="mauro" begin="30001" end="30001"/
            >
</trace>
```

Ejemplo de traza inválido:

```
<trace>
        <event name="entry" value="mauro" begin="60" end="60"/>
        <event name="entry" value="pablo" begin="120" end="120"/>
        <event name="exit" value="mauro" begin="1200" end="1200"/>
        <event name="entry" value="mauro" begin="6250" end="6250"/>
        <event name="exit" value="mauro" begin="20001" end="20001"/
            >
        <event name="exit" value="pablo" begin="30000" end="30000"/
            >
</trace>
```

7.1.7 *Escenario 7 -NoTwoEntriesWithoutExitInBetween*

Con este escenario buscaremos asegurar que un usuario no puede ingresar dos veces consecutivas sin haber salido previamente.

La propiedad expresada será:

"No puede haber dos entradas de un usuario que no estén separadas por una salida"

El escenario que utilizaremos será el siguiente:

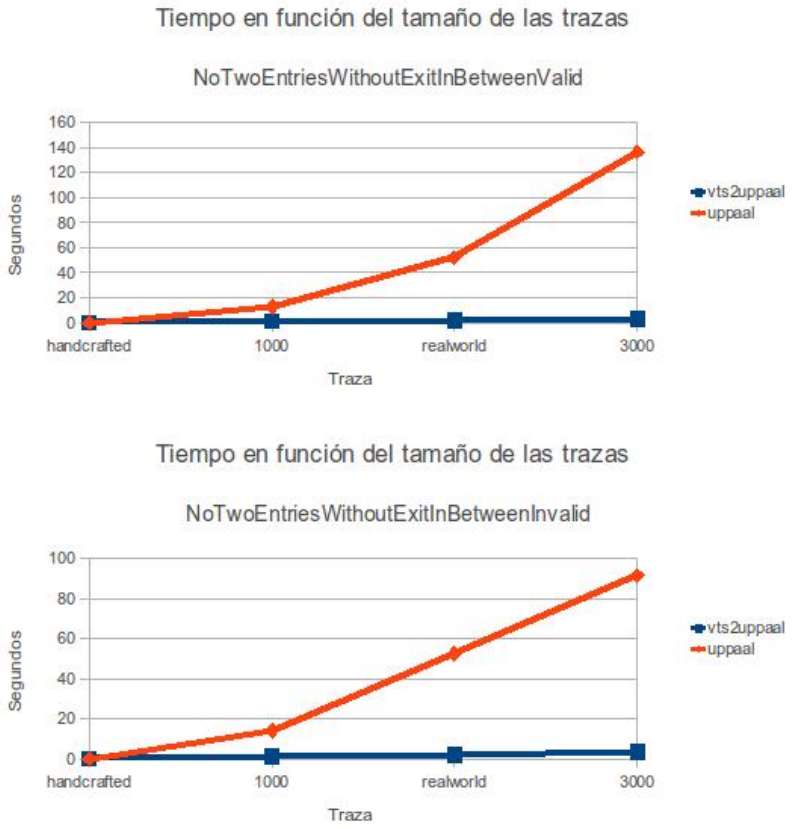
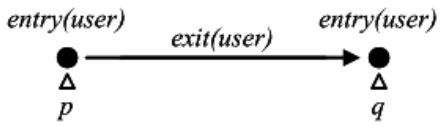


Figura 21: Flo:NoTwoEntriesWithoutExitInBetween



Ejemplo de traza válido:

```
<trace>
        <event name="entry" value="mauro" begin="1" end="1"/>
        <event name="exit" value="mauro" begin="2" end="2"/>
        <event name="entry" value="mauro" begin="3" end="3"/>
</trace>
```

Ejemplo de traza inválido:

```
<trace>
        <event name="entry" value="mauro" begin="1" end="1"/>
        <event name="entry" value="mauro" begin="3" end="3"/>
</trace>
```

7.1.8 *Escenario 8 -NobodyRestsInside*

Con este escenario buscaremos asegurar que ningún usuario quede dentro del edificio.

La propiedad expresada será:

"No puede haber un evento de entrada sin su correspondiente evento de salida"

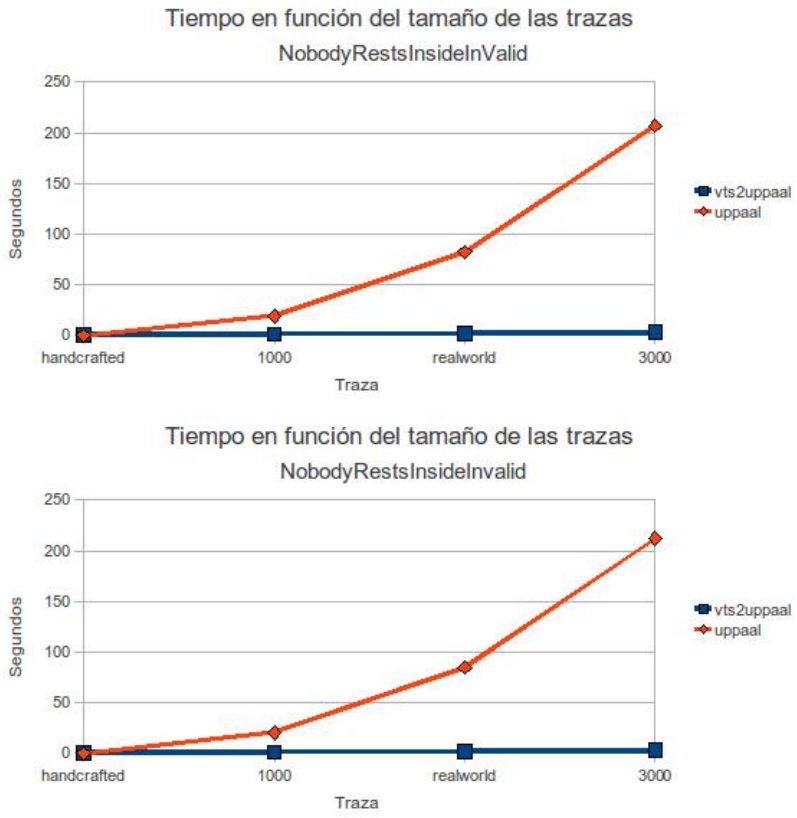
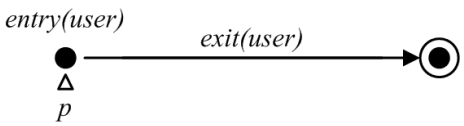


Figura 22: NobodyRestsInside

El escenario que utilizaremos será el siguiente:



Ejemplo de traza válido:

```
<trace>
        <event name="entry" value="mauro" begin="1" end="1"/>
        <event name="entry" value="pablo" begin="3" end="3"/>
        <event name="exit" value="mauro" begin="5" end="5"/>
        <event name="exit" value="pablo" begin="9" end="9"/>
</trace>
```

Ejemplo de traza inválido:

```
<trace>
        <event name="entry" value="mauro" begin="1" end="1"/>
        <event name="entry" value="pablo" begin="3" end="3"/>
</trace>
```

7.2 ANÁLISIS DE LOS RESULTADOS

Para todos los escenarios planteados y con todas las trazas verificadas, los resultados obtenidos concuerdan con lo teóricamente esperado, lo cual indica que el algoritmo presentado y su implementación sirven como prueba de concepto de la solución propuesta al problema estudiado.

CASO	ESCENARIO VTS++	OBSERVADOR
NoEntryInvalidUser	2 puntos, 1 eje	4 puntos, 7 ejes
NoInvalidationWithUserInside	2 puntos, 1 eje	4 puntos, 7 ejes
MaxLunchTime	5 puntos, 4 ejes	7 puntos, 15 ejes
MaxTimeInside	2 puntos, 1 eje	4 puntos, 6 ejes
MinTimeBetweenEntries	2 puntos, 1 eje	4 puntos, 6 ejes
NoTwoEntriesWithoutExitInBetween	2 puntos, 1 eje	4 puntos, 7 ejes
NobodyRestsInside	2 puntos, 1 eje	4 puntos, 6 ejes
NoExitWithoutEntry	2 puntos, 1 eje	3 puntos, 5 ejes

Table 1: Tamaño de los autómatas generados

A su vez puede observarse en las gráficas de resultados la notoria diferencia de escalabilidad de los dos componentes principales de la solución utilizada (Vts2Uppaal y Uppaal). Mientras que el crecimiento del tamaño de las trazas provoca en el tiempo ejecución de Vts2Uppaal un impacto practicamente despreciable, para Uppaal esta variación produce un aumento exponencial.

Este fenómeno se produce debido al modo en que Uppaal realiza la exploración de los caminos posibles antes de emitir una respuesta.

Por ser una solución diseñada para resolver un problema diferente del que nos ocupa, no tiene en cuenta características específicas del análisis de trazas de ejecución que podrían reducir el espacio de caminos a explorar y así mejorar la eficiencia de la verificación.

Una diferencia fundamental entre estos dos dominios de aplicación radica en que los modelos tienen un tamaño acotado (generalmente pequeño), en cambio las trazas de ejecución de un sistema productivo suelen ser considerablemente grandes.

El tamaño de los autómatas observadores sólo depende del tamaño de los escenarios y no de las trazas, como se observa en la tabla 1. Es precisamente la generación del autómata observador la que insume la mayor parte del tiempo de ejecución de vts2uppaal, ya que el autómata correspondiente a la traza de ejecución es trivial. Esto explica el bajo impacto del aumento del tamaño de las trazas en el tiempo de ejecución que se observa en los gráficos de la sección anterior.

7.2.1 *Limitaciones*

Existen algunos escenarios que, a pesar de ser interesantes para el caso de estudio, no pudieron ser especificados debido a limitaciones de la extensión realizada. Entre ellos:

- Verificación de accesos distribuída (Ej: validación de que un usuario no ingrese a dos edificios diferentes sin haber salido del primero).
- Autorización de acceso de un usuario a sí mismo (Ej: validación de que el usuario que otorga la autorización sea diferente de quien la recibe).

8 CONCLUSIÓN

La verificación de sistemas a partir de trazas de ejecución requiere contar con una serie herramientas básicas que permitan:

1. Generar las trazas de ejecución.
2. Especificar las propiedades que los sistemas deben cumplir.
3. Verificar que las trazas cumplen las propiedades especificadas.

Durante este trabajo presentamos la API de TraceIt! para registro de trazas de ejecución para aplicaciones embebidas de tiempo real.

Si bien TraceIt! cumple con sus objetivos de diseño y nos permitió realizar el estudio propuesto, consideramos que la necesidad de incluir la API en la aplicación a verificar constituye un punto débil de su diseño ya que, al requerir la modificación del código de la aplicación a testear, hace posible la introducción de nuevos errores.

Este problema podría evitarse empleando un mecanismo de anotaciones sobre el código (a la manera de DocStrings) de forma que el registro de eventos se agregue a la aplicación mediante un análisis automático del código fuente previo a su despliegue.

Si bien TraceIt! es en principio una API exclusiva para WindowsCE, la herramienta de procesamiento de trazas que desarrollamos (ReadLog) puede extenderse a otros formatos de entrada y/o de salida, lo cual permite aprovecharla en contextos diferentes del analizado durante este trabajo.

Utilizando TraceIt! para la generación de trazas de ejecución analizamos la aplicabilidad y limitaciones de VTS para la verificación de sistemas críticos y comprobamos que, si bien podría utilizarse sin extensiones, en la práctica, existe un conjunto de escenarios que escapan a su poder expresivo.

A partir de lo observado mostramos cómo extender VTS con variables de modo de brindar una solución satisfactoria para esta limitación y desarrollamos un conjunto de herramientas que sirven como prueba de concepto de la solución encontrada.

Luego analizamos un caso de estudio y mostramos el comportamiento de las herramientas utilizadas, obteniendo resultados satisfactorios.

Esto nos permite concluir que un lenguaje gráfico, amigable y a la vez formal puede utilizarse para especificar requerimientos de sistemas y verificarlos a partir de sus trazas de ejecución, sin por ello renunciar al poder expresivo de otro tipo de lenguajes basados en fórmulas tales como LTL o TCTL.

Si bien consideramos que la extensión realizada al lenguaje cumple los objetivos propuestos, no resulta suficiente para expresar de una forma práctica ciertos escenarios. Por ejemplo, en nuestro caso de estudio de control de acceso:

- El control de accesos en diferentes edificios requeriría la posibilidad de predicar sobre al menos dos variables a la vez (El edificio y el usuario).

- Limitar cantidad de ocurrencias de un evento (Por ej, ingresos inválidos).
- Los escenarios no dominados por asignación no pueden ser verificados.

Respecto al uso de Uppaal como herramienta de verificación, observamos que presenta ciertas limitaciones:

1. Una característica inherente al problema estudiado es el no-determinismo del proceso de verificación. Uppaal ataca este problema utilizando backtracking para realizar una exploración exhaustiva, por lo que el tiempo de verificación puede aumentar exponencialmente respecto del tamaño de las trazas, lo cual deriva en un límite práctico para su aplicación en sistemas reales.
2. Las variables en Uppaal sólo admiten valores enteros de 2 bytes, lo cual podría ser un impedimento en ciertas aplicaciones.

8.1 TRABAJOS FUTUROS

- Estudiar el uso de otras herramientas de verificación (existentes o ad-hoc) que reemplacen a Uppaal y permitan obtener una mejora en el tiempo de verificación. Por ejemplo un enfoque alternativo podría ser el de realizar una exploración en paralelo en lugar de utilizar backtracking.
- Estudiar la posibilidad de realizar un pre-procesamiento de la traza para reducir el espacio de exploración de la herramienta.
- Extender el lenguaje para permitir más de una variable asociada a cada etiqueta del escenario. 7.2.1
- Extender el lenguaje para agregar comparaciones entre diferentes variables.7.2.1
- Implementar una versión de TraceIt! que no requiera la inclusión de la API en el código de la aplicación.

Part III

APPENDIX

APÉNDICE

A.1 API TRACEIT!

La API de TraceIt! está compuesta por tres clases:
· Tracer
· Event
· Message
Para utilizar la API sólo se necesita tener conocimientos sobre la primera (Tracer), las otras son de uso interno.

A.1.1 *Métodos de Tracer*

Métodos relacionados con manejo de eventos

Estos métodos crean entradas en el log del sistema a modo de marcar puntos específicos de la ejecución de un programa, los cuales están relacionados con la ocurrencia de ciertos eventos.

```
EventId traceBefore( const char * event )
```

Este método marca un punto previo a la ocurrencia de cierto evento. Este evento se describe mediante una cadena de texto.

El valor retornado consiste en un identificador único de evento (Número entero positivo).

```
void traceAfter( EventId id )
```

Este método marca un punto posterior a la ocurrencia del evento. Recibe como parámetro el id de un evento (generado con la llamada a traceBefore).

Estos métodos están pensados para ser utilizados en conjunto, de modo de conseguir un alto nivel de precisión respecto del momento en que el evento sucedió (Al marcar un punto anterior y posterior a su ocurrencia se acota el margen de error).

```
void trace( const char * event )
```

Este método marca un punto en la ejecución del programa en el que el evento sucedió. Esta primitiva se utiliza para logear información acerca de eventos que no necesitan de una alta precisión.

```
void traceAfterLastEvent()
```

Este método marca un punto en la ejecución posteriormente a la ocurrencia del último evento. Se trata de un método abreviado para la llamada a traceAfter pasando como parámetro el id del último evento.

Message related methods

Estos métodos crean entradas en el log del sistema a modo de marcar puntos específicos de la ejecución de un programa, los cuales están relacionados con la transmisión de mensajes.

```
MessageId traceSend( const char * msg, bool generateId )
```

Este método marca el envío de un mensaje de aplicación. Los contenidos del primer argumento contienen el mensaje y el segundo indica si se necesita generar un id de mensaje (En caso contrario se asume que la generación de identificadores se implementa fuera de la API).

```
void traceReceive( MessageId mid, bool isBroadCast )
```

Este método marca la recepción de un mensaje previamente enviado. Toma como argumentos el id del mensaje y un indicador que se utiliza para determinar si el mensaje debe seguir disponible para su recepción posterior o si debe ser liberado en este momento.void traceReceive(const char * msg)

```
void traceReceive( const char * msg )
```

Otra forma de marcar la recepción de un mensaje es la llamada a este método. La diferencia con el anterior es que este sólo toma una cadena de caracteres la cual indica cuál es el mensaje que debe marcarse como recibido. Este método debe usarse conjuntamente con una llamada del tipo traceSend("msg", false) ya que se asume que el mecanismo de apareo se realiza fuera de la API.

```
ChannelId declareChannel( const char * name )
```

Hasta aquí todos los mensajes son transmitidos a través del canal por defecto. La API también ofrece la posibilidad de definir nuevos canales de comunicación. Este método debe ser llamado antes de la ejecución de los siguientes, de modo de declarar todos los canales que serán utilizados durante la ejecución de la aplicación. El parámetro es el nombre a través del cual se hará referencia al canal en el autómata generado y el valor de retorno es el identificador que se utilizará durante las próximas llamadas.

```
MessageId traceSend( const char * msg, ChannelId cid, bool generateId )
```

Este método cumple la misma función que traceSend(const char * msg, bool generateId), sólo que permite la especificación del canal que desea utilizarse para el envío del mensaje.

```
void traceReceive( const char * msg, ChannelId cid )
```

Este método es análogo a traceReceive(const char * msg), sólo que en esta llamada el usuario debe especificar de qué canal se debe obtener el mensaje enviado previamente.

Constructores

Actualmente existe un único constructor para la clase Tracer:

```
Tracer( bool writeText )
```

El argumento determina si la API debe generar, además de la información para el análisis sintáctico, información textual en el log del sistema, de modo de que el mismo sea humanamente legible.

Cómo usar la API

El objetivo de la API de TraceIt! es ser usada como un medio de generación de entradas para herramientas de verificación de modelos como UPPAAL o Kronos para análisis visual.

Para utilizar la API se necesita la inclusión de los siguientes archivos:
Event.h
Message.h
TraceIt.h
Tracer.h
Event.cpp
Message.cpp
Tracer.cpp

A su vez, la plataforma debe tener disponible la aplicación celogflush.

Una vez que se ha generado un log válido (utilizando la API), puede utilizarse la herramienta ReadLog (En la computadora de escritorio) para analizar sintácticamente el log y emitirlo en diferentes formatos, incluyendo:

- Texto plano
- XML
- xta (Formato de UPPAAL)
- Formato de Kronos

Adicionalmente, la herramienta ReadLog puede generar un archivo de eventos para VTS.

Con el archivo de eventos VTS y la plantilla VTS es posible definir escenarios que alimentarán alguna de las herramientas de verificación de modelos para realizar la verificación de las propiedades análisis a través de las trazas de la aplicación.

A.2 MANUAL DEL USUARIO VTS2UPPAAL

Requerimientos:
python >= 2.5
python-gv
python-graph
Uppaal 4.0.6

A.2.1 *Descripción*

vts2uppaal toma un escenario VTS++ y una o más trazas XML obtenidas de la ejecución de un programa que cuente con un sistema de generació de trazas, por ejemplo TraceIt!, y genera un autómata compatible con Uppaal que permite verificar si las trazas satisfacen el escenario.

A.2.2 *Modo de uso*

```
python vts2uppaal <escenario> <automata> <traza1> [<traza2> ... <trazaN>]
```

El primer parámetro corresponde al escenario VTS extendido (en formato .vdx) que será utilizado contra el que se desea verificar las trazas.

El segundo parámetro corresponde al nombre (sin extensión) de los archivos en que se guardarán el autómata generado y la query para ser utilizada con Uppaal.

A partir del tercer parámetro se indican las trazas que se desea verificar.

Una vez que se tiene los archivos generados, la verificación se realiza mediante la utilidad verifyta (incluida en Uppaal): verifyta <automata>.xml <automata>.qry (Archivos generados por vts2uppaal).

La salida debe interpretarse de la siguiente forma:

"Property is satisfied" implica que la traza satisface el escenario, por lo tanto (ya que se trata de un escenario negativo), el sistema viola la restricción especificada en el escenario.

"Property is NOT satisfied" implica que las trazas analizadas no satisfacen el escenario y por lo tanto no es posible afirmar que el sistema viola la restricción especificada en el escenario.

Opciones:

-verbose: permite obtener informacipon acerca de la salida.

BIBLIOGRAFÍA

[1] Alejandra Alfonso, Víctor Braberman, Nicolás Kicillof, and Alfredo Olivero. Visual timed event scenarios. *Proceedings del 26th International Conference on Software Engineering*, pages 167–177, 2004. (Cited on pages 1 and 7.)

[2] Rajeev Alur. *Techniques For Automatic Verification of Real-time Systems*. PhD thesis, Stanford University, August 1991. (Cited on page 5.)

[3] Rajeev Alur and David Dill. A theory of timed automata. *Theoretical Computer Science*, pages 183–235, 1994. (Cited on page 37.)

[4] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-Checking in Dense Real Time. (Cited on page 2.)

[5] Peter C Bates. Debugging heterogeneous distributed systems using event-based models of behaviour. *ACM Transactions on Computer Systems*, pages 1–31, 1995. (Cited on page 5.)

[6] V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. Taxys = esterel + kronos - a tool for verifying real-time properties of embedded systems. UK, 2001. IEEE Control Systems Society. (Cited on page 5.)

[7] Víctor Braverman, Diego Garbervetzky, and Alfredo Olivero. ObsSlice: A Timed Automata Slicer based on Observers. (Cited on page 11.)

[8] Doron Drusinsky. The temporal rover and the atg rover. *Proc. of the 7th Intl. Workshop SPIN*, 1885:323–330, 2000. (Cited on page 17.)

[9] Martin Gogolla and Francesco Parisi Presicce. State diagrams in uml: A formal semantics using graph transformations. 1998. (Cited on page 2.)

[10] Klaus Havelund and Grigore Rosu. Java pathexplorer - a runtime verification tool. Proceedings 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space, ISAIRAS'01, 2001. (Cited on page 17.)

[11] Rajeev Motwani John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Prentice Hall, 2007. (Cited on page 2.)

[12] Kim G Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. (Cited on page 12.)

[13] Insup Lee, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. Runtime assurance based on formal specifications. Proc. of the IEEE Intl. Conf. PDPTA '99, 1999. (Cited on page 17.)

[14] Masoud Mansouri-Samani and Morris Sloman. Gem: A generalized event monitoring language for distributed systems. *Distributed Systems Engineering Journal*, pages 96–108, 1997. (Cited on page 5.)

[15] Grigore Rosu and Havelund Klaus. Synthesizing dynamic programming algorithms from linear temporal logic formulae. *RIACS Technical Report 01.15*, 2001. (Cited on page 2.)

[16] Sergio Yovine. Kronos: A Verificacion Tool for Real-Time Systems. (Cited on page 11.)