Evaluation of Markov Models for Architecture Conformance Checking

G. Rodríguez, M. Armentano, Á. Soria, and E. Corengia

***Abstract*—Conformance between architecture and implementation is a key aspect of architecture-centric development. However, the architecture "as documented" and the architecture "as implemented" tend to diverge from each other over time. Thus, conformance checks should be run periodically on the system in order to detect and correct differences. Despite having a structural conformance analysis, assessing whether the main scenarios describing the architectural behavior are faithfully implemented in the code is still challenging. Checking conformance to architectural scenarios is usually a time-consuming and error-prone activity. In this article, we describe *ArchLearner*, a tool to assist architects to bridge the gap between architecture and its implementation. The architecture is specified with Use-Case Maps (UCMs), a notation for modeling both high-level structure and behavior. *ArchLearner* uses Markov Models to detect code deviations with respect to predetermined UCMs, based on the analysis of system execution traces for those UCMs. The results from two case-studies have shown that *ArchLearner* is practical and reduces conformance checking efforts.**

***Index Terms*—Conformance checks, Software architecture, Tool support, Use case maps, Variable order Markov models.**

I. Introducción

La arquitectura de software es un elemento de suma importancia en el proceso de diseño y desarrollo de sistemas que ha sido altamente adoptado en la industria. La visión arquitectónica del sistema es abstracta, deja de lado detalles de implementación, algoritmos o representación de la información y se enfoca en detalles estructurales y de interacción de los elementos del sistema que la componen. Su desarrollo se considera como el primer paso hacia el diseño de un sistema de software y permite capturar decisiones de diseño que servirán para satisfacer sus principales atributos de calidad o drivers como performance, modificabilidad y seguridad [1].

La arquitectura de software abarca tanto aspectos estructurales como comportamentales del sistema, y es comúnmente documentada a través de un conjunto de vistas.

G. Rodríguez, Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil, Buenos Aires, Argentina, (e-mail: guillermo.rodriguez@isistan.unicen.edu.ar).

M. G. Armentano, Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil, Buenos Aires, Argentina, (e-maiul: marcelo.armentano@isistan.unicen.edu.ar).

A. Soria, Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil, Buenos Aires, Argentina, (e-mail: alvaro.soria@isistan.unicen.edu.ar).

E. Corengia, Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil, Buenos Aires, Argentina, (e-mail: emiliocorengia@gmail.com).

Una vista arquitectónica describe el conjunto de componentes del sistema y las relaciones asociadas a ellos, que, a su vez, representan los intereses de uno o más stakeholders. Las vistas suelen ir acompañadas de especificaciones comportamentales en forma de escenarios que detallan los principales drivers arquitectónicos. La correcta documentación de la Arquitectura de Software será de suma importancia para el desarrollo y mantenimiento del sistema. Incluso una arquitectura perfecta resulta inútil si no es comprendida por nadie, o peor aún, si es mal interpretada [2].

Una vez que la arquitectura fue materializada [3], pasa a una etapa de mantenimiento donde el sistema evoluciona. Esta evolución va provocando que la documentación de la arquitectura pierda representatividad sobre su implementación. Una de las principales causas de desvío arquitectónico se debe a que la documentación arquitectónica va quedando, progresivamente, desactualizada. En caso de no tratar este problema adecuadamente, se produce una diferencia entre la arquitectura documentada y la implementada [1], conocida como desvío arquitectónico, y que conduce directamente a problemas de erosión arquitectónica. Comprobar que los escenarios documentados se mantengan representativos a su implementación sigue siendo una tarea compleja y que requiere esfuerzo.

Este trabajo presenta *ArchLearner*, una herramienta para mantener sincronizada la implementación, a medida que evoluciona, con la representación de la arquitectura. El arquitecto es quien está a cargo del mapeo inicial entre los elementos de la arquitectura y los elementos de implementación. Durante esta evolución, los desarrolladores van progresivamente introduciendo cambios en el código, que pueden producir que los mapeos arquitectónicos vayan quedando desactualizados. Esta información es generalmente representada en forma de una distribución de probabilidades, y es utilizada por *ArchLearner* para aprender acerca de las reglas que rigen el comportamiento del sistema.

En particular, los modelos de Markov [4] resultan favorables para representar este tipo de información en forma de trazas de ejecución. Una cadena de Markov es un proceso estocástico que cumple con la propiedad de Markov. Ésta estipula que dado el estado actual, los estados futuros son independientes de los estados pasados. Las cadenas de Markov de orden fijo son una extensión natural en la que el próximo estado depende de un número fijo de estados anteriores. Los modelos de Markov de Orden Variable surgen como una solución al crecimiento exponencial de los estados causado por el incremento del orden en los modelos. En estos modelos el número de variables aleatorias que condicionan la distribución de probabilidades está ligado al contexto que se

está observando. Estos modelos plantean que en una situación real existe cierta relación entre los estados, representada por el contexto, donde algunos de los estados pasados no condicionarán a los estados futuros, obteniendo así una reducción en el número de parámetros que conformarán al modelo.

Para documentar la arquitectura *ArchLearner* utiliza FLABot [5], una herramienta para la depuración y edición de Use Case Maps (UCMs). El arquitecto entrena *ArchLearner* con versiones estables del sistema, y luego en etapas posteriores el revisor utiliza esta información de comportamiento de las versiones para actualizar la documentación. Además, *ArchLearner* es capaz de sugerir actualizaciones en los UCMs que hayan sido diagnosticados con desviaciones al comportamiento esperado para así poder reflejar los cambios de las nuevas revisiones del sistema en la documentación de forma automática.

El resto del trabajo se organiza de la siguiente manera. La sección 2 describe los trabajos relacionados. La sección 3 presenta el enfoque propuesto. La sección 4 reporta los resultados experimentales. La sección 5 discute los resultados obtenidos. Finalmente, la sección 6 concluye nuestro trabajo e identifica futuras líneas de trabajo.

II. Trabajos Relacionados

Verificar la correspondencia entre la arquitectura de software y la implementación para prevenir problemas de erosión ha sido un tema activo de investigación, con enfoques estáticos que a menudo utilizan técnicas de ingeniería reversa [6, 7, 8, 9] para recuperar la arquitectura, y otros que utilizan Lenguajes de Descripción Arquitectónica (ADLs) como soporte [10, 11]. De todas formas, enfoques que hagan uso de la información de ejecución del sistema a nivel arquitectónico, como es el caso de *ArchLearner*, han sido menos explorados. Soluciones basadas en ingeniería reversa buscan reconstruir parcialmente una arquitectura de software en función de su código fuente o implementación para luego poder compararla con la propuesta inicial del arquitecto. Suponiendo que un analizador de código fuente fuese capaz de realizar el proceso inverso de abstraer y especificar una arquitectura que represente fielmente lo que su materialización implica, se estaría solucionando el problema de inconsistencia entre ambos modelos al contar siempre con una arquitectura actualizada. En otras palabras, con la reconstrucción es posible detectar zonas de erosión no deseadas. La calidad de la reconstrucción es clave en este tipo de enfoques.

Lamentablemente las soluciones basadas en ingeniería reversa están limitadas por la inherente diferencia entre estructuras estáticas como clases, paquetes y las estructuras variables en tiempo de ejecución, que son la esencia de la mayoría de las descripciones arquitectónicas. Las estructuras variables pueden ser incluso desconocidas hasta que el programa se ejecuta, por ejemplo: clientes y servidores pueden aparecer y desaparecer dinámicamente; componentes, como los DLLs, pueden cargarse dinámicamente [9]. También existe información semántica que se pierde cuando una arquitectura es implementada mediante clases y métodos, haciendo imposible su recuperación mediante ingeniería reversa. Las representaciones generadas utilizando esta técnica pueden aplicar información adicional provista por estándares como patrones, estilos arquitectónicos o incluso heurísticas para abstraer la información contenida en el código; sin embargo, esta representación puede no ser necesariamente la pretendida por el arquitecto. Por otro lado, las soluciones basadas en ADLs buscan forzar el cumplimiento de las restricciones impuestas por una arquitectura desde su concepción, siendo fundamental un desarrollo centrado en la arquitectura donde toda etapa de implementación sea precedida por otra de diseño arquitectónico [12]. Utilizando un ciclo de desarrollo en cascada, evolutivo o iterativo, se podría impedir que la implementación viole o erosione su arquitectura de base. Para lograr esto, es necesario utilizar un lenguaje de especificación lo suficientemente rico en semántica y por lo tanto, de carácter formal, algo que en muchos proyectos puede resultar impracticable.

En [17], los autores presentan la problemática de mantener información de diferentes fuentes consistente, con coherencia entre los modelos y la documentación. Para ello, los autores abordan la comprobación de la coherencia entre modelos y artefactos de lenguaje natural textual utilizando comprensión del lenguaje natural.

Por otro lado, en [18] los autores han propuesto una evaluación basada en modelos que utiliza vistas arquitectónicas para derivar automáticamente casos de prueba para verificar las restricciones arquitectónicas en el código. Los autores han utilizado un conjunto de herramientas correspondientes para un estudio de caso industrial real utilizando un protocolo de estudio de caso sistemático. Además, han adoptado técnicas exhaustivas de inyección de fallas para detectar las violaciones de restricciones.

En [21] se presenta un framework que soporta el desarrollo de aplicaciones para la conformidad arquitectónica mediante el análisis de reglas y métricas. El framework utiliza grafos agregados y operaciones MapReduce.

Sin embargo, el uso de técnicas de machine Learning ha sido un campo de estudio poco explorado para controlar la erosión arquitectónica.

III. Enfoque Propuesto

El objetivo de *ArchLearner* es encontrar diferencias arquitectónicas entre la documentación en forma de UCM con su correspondiente implementación considerando patrones de comportamiento detallados en la arquitectura, y de ser necesario generar una nueva versión de los UCMs donde estas diferencias han sido resueltas. Por medio del estudio de comportamiento del sistema en ejecución, *ArchLearner* es capaz de encontrar estas relaciones causales entre elementos arquitectónicos para luego utilizarlas en el proceso de análisis y evaluación de las vistas arquitectónicas dadas. Asimismo, cabe destacar que no todos los cambios presentes en la implementación son visibles (o significantes) a nivel de UCM. La Fig. 1 muestra un esquema conceptual del enfoque de *ArchLearner* donde se puede identificar 3 etapas claramente definidas: entrenamiento, generación de conocimiento, y diagnóstico y sincronización de UCMs. Estas 3 etapas están diseñadas para ejecutarse en distintos momentos en el proceso actualización de UCMs.

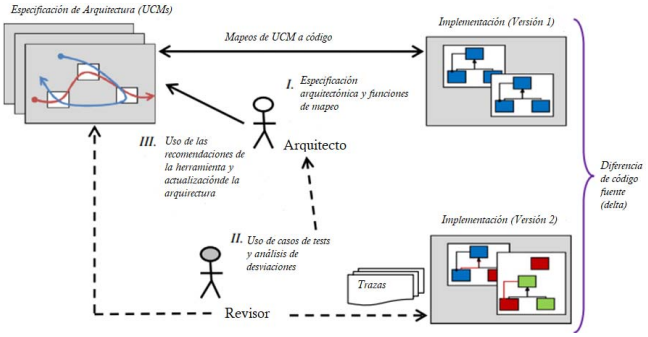


Fig. 1. Etapas de *ArchLearner*.

A. Etapa de Entrenamiento

En nuestro enfoque consideramos dos roles básicos: el arquitecto y el revisor. El arquitecto especifica la arquitectura y la actualiza (si es necesario), mientras el revisor inspecciona el código para adherirse a las reglas arquitectónicas y comunicar los resultados al arquitecto. El proceso comienza con la descripción arquitectónica inicial en forma de UCMs donde el arquitecto describe los principales escenarios del sistema, código instrumentado para poder monitorear la ejecución del sistema, y las correspondientes funciones de mapeo a código en los UCMs que permiten relacionar responsabilidades UCM a métodos en la implementación. Esta información de los mapeos entre la arquitectura y la implementación es clave para que *ArchLearner* pueda realizar las etapas posteriores de entrenamiento, diagnóstico y sincronización. El siguiente paso consiste en capturar los *logs de ejecución* del sistema que sirven para recolectar información acerca de su comportamiento.

La Fig. 2 muestra que esta actividad es realizada por el revisor quien facilita un conjunto de casos de test (i.e. casos de prueba) diseñados especialmente para ejercitar los principales escenarios modelados en los UCMs sobre el sistema instrumentado, permitiendo al componente *ExecutionLogger* capturar los logs de ejecución.

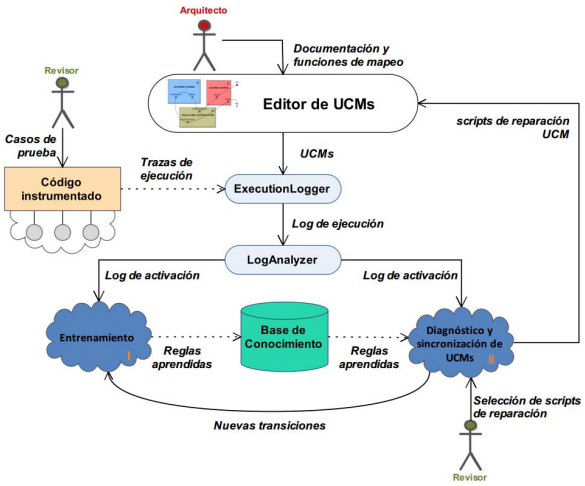


Fig. 2. Enfoque de *ArchLearner*.

El revisor elige los casos de test para un UCM determinado y luego ejercita la implementación actual con esos casos de test. En definitiva, ArchLearner instrumenta el código de la aplicación para que su comportamiento pueda ser monitoreado. Por lo tanto, los casos de test producen un conjunto de *trazas de ejecución* de la aplicación, las cuales serán utilizadas para calcular las diferencias con el UCM analizado [13]. Estas *trazas de ejecución* contienen información vital para el entrenamiento, diagnóstico y sincronización de UCMs tal como llamadas a métodos, creación de objetos, diferentes hilos de ejecución y excepciones. En este punto, la información contenida en estos logs está vinculada con la implementación y ejecución particular de los casos de prueba utilizados.

B. Generación de Conocimiento

Antes de poder diagnosticar los escenarios de UCM, la herramienta necesita generar una base de conocimiento sobre el comportamiento del sistema que está evaluando. Particularmente, en esta etapa de entrenamiento se intenta encontrar relaciones causales dentro de los *logs de activación* entre distintas responsabilidades del sistema. Los *logs de activación* resultan de abstracciones de los logs de ejecución realizadas por *Log Analyzer*. Estas relaciones causales son justamente las que se tratan de modelar en un UCM, por lo tanto, su estudio permite que en etapas futuras de diagnóstico y sincronización de UCMs se pueda evaluar distintas versiones de la implementación del sistema.

En primer lugar, hay que definir cómo representar la información de ejecución en forma de *logs de activación*. Las diferentes secuencias de símbolos que se originan dentro de un *log de activación* pueden ser consideradas como una fuente natural en un problema de Markov, ya que la forma en que son generadas no respeta un orden específico o al menos no se puede asegurar un orden por las características propias de la ejecución en sistemas con alto grado de concurrencia. Por esta razón, se tratan las secuencias de símbolos dentro de un *log de activación* como una serie de variables aleatorias $s = s_1 s_2 ... s_i$, donde cada s_j obtiene un valor discreto del alfabeto finito Σ dentro del ámbito del sistema. De esta forma es posible modelar tales secuencias como cadenas de Markov de orden $L > 1$, donde el orden es la longitud de la memoria del modelo. Alternativamente, como se pone en práctica en [19] tales secuencias pueden modelarse como modelos de Markov de orden variable donde cada sub-secuencia tiene una longitud de a lo sumo L. En este trabajo, se ponen a prueba ambos modelos, y en caso de que en un futuro se agreguen otros modelos, el revisor puede elegir qué estrategia seguir en el proceso de aprendizaje a través de la interfaz de *ArchLearner*. Luego de haber seleccionado el modelo de representación de logs, Markov de orden fijo o variable en nuestro caso, el siguiente paso consiste en construir el modelo en sí por primera vez con el objetivo de poder describir el comportamiento del sistema.

Para poder llevar a cabo esta construcción, es necesario interpretar las secuencias contenidas en los *logs de activación*

como una serie de variables aleatorias, donde el orden ocurrencia de los símbolos está definido en las trayectorias mismas de los UCMs. El pseudocódigo de la Figura 3 describe el proceso de aprendizaje donde se genera información sobre el comportamiento del sistema. Como se interpreta cada nueva observación (paso número 08 en el pseudocódigo) va a depender del modelo de Markov seleccionado, y de los parámetros de configuración utilizados. Esta información creada en esta etapa de entrenamiento luego será utilizada en etapas posteriores de diagnóstico y sincronización de UCMs. En el caso de Markov de orden fijo, la probabilidad de entrar en un estado arbitrario en el momento t+1 sólo depende del estado actual en el momento t pero no en los estados anteriores [20].

```
FUNCTION CrearModelo(Documentación d, Parámetros p)
01.  Modelo m ← ∅
02.  FOR cada UCM u en d DO
03.    FOR cada Trayectoria t en u DO
04.      FOR cada Responsabilidad r en t DO
05.        Lista tags ← ExtraerTags(r)
06.        FOR cada Tag t en tags DO
07.          Observación σ← ComoSímbolo(t)
08.          ActualizarModelo(m, σ, p)
09.        END FOR
10.      END FOR
11.    END FOR
12.  END FOR
RETURN m
```

Fig. 3 PseudoCódigo para la construcción del PSA.

En (1) se describe un proceso estocástico, en el cual se define que la empírica condicional de observar una acción X_{t+1} justo después de una secuencia dada (s_{t+1}) depende únicamente de los n estados anteriores (s_0, s_1,…, s_t).

$$P(X_{t+1}=s_{t+1} \mid X_t=s_t,\ldots, X_n=S_n) = P(X_{t+1}=s_{t+1} \mid X_t=s_t, X_{t-1}=s_{t-1},\ldots, X_0=S_0) \quad (1)$$

Esto significa que las probabilidades de transición entre estados son constantes en el tiempo, lo que facilita la creación de un autómata de probabilidad de sufijo (PSA) a partir de los símbolos contenidos en *logs de activación* al considerar sólo el estado actual (paso número 05 en la Figura 3). En el caso de Markov de orden variable (VOM), la construcción de un PSA que represente la información de los *logs de activación* no es una tarea trivial. En modelos VOM el número de variables aleatorias puede variar según la realización de observaciones específicas. Por lo tanto, para lograr un aprendizaje incremental de las observaciones capturadas en los logs (paso 08 en el pseudocódigo de la Figura 3), en la construcción del PSA es necesario tomar las siguientes medidas adicionales:

1. Para cada símbolo observado añadir un nodo para cada sub-secuencia anterior a en el PSA (es decir, cada posible estado para la acción observada) con una longitud que varía entre 1 y la memoria máxima permitida (L).

2. Mantener registro del número de veces que se observa cada símbolo.

3. Podar los nodos del PSA favoreciendo las secuencias más cortas para obtener buenas predicciones.

4. Eliminar aquellos nodos del PSA que estén predichos por nodos con probabilidad de ocurrencia cero.

Esta etapa de entrenamiento, independientemente del modelo de aprendizaje utilizado, está diseñada para ser utilizada sobre una versión estable del sistema para que pueda servir de ejemplo y así poder generar una base de conocimiento, en forma de modelos de Markov de orden fijo o variable, por ejemplo, sobre los patrones de comportamiento que se suceden en el sistema. Esta base de conocimiento luego puede ser actualizada de forma incremental a lo largo de las sucesivas revisiones que se realicen sobre el sistema durante su evolución, tomando como válidas las reparaciones que seleccione el revisor.

C. Etapa de Diagnóstico

En la etapa de diagnóstico, *ArchLearner* evalúa los UCMs actuales identificando cualquier ausencia o divergencia en las vistas arquitectónicas con respecto al comportamiento esperado. Con el fin de desacoplar la información de estas características meramente de implementación, *ArchLearner* transforma esta información de bajo nivel contenida en los logs de ejecución en secuencias lineales de activación de responsabilidades junto a información adicional como threads, parámetros y métodos ejecutados por componente entre otros, basándose en las funciones de mapeo responsabilidad-código presentes en los UCMs. Esta abstracción que sucede en el componente *LogAnalyzer* permite que *ArchLearner* no dependa directamente de ningún lenguaje de programación en particular, ni de características propias de ejecución.

Como resultado, los *logs de activación* son los utilizados en las etapas de entrenamiento, diagnóstico y sincronización de UCMs. La etapa de entrenamiento consiste en conocer y comprender el sistema por medio del estudio de patrones de comportamiento que se encuentran presentes en los *logs de activación*. Esta información es generalmente representada en forma de una distribución de probabilidades, y es utilizada por la herramienta para aprender acerca de las reglas que rigen el comportamiento del sistema.

De esta manera, el modelo de Markov propuesto permite sugerir acciones correctivas que sirvan para actualizar la documentación actual de UCM. Estas acciones correctivas, en forma de *scripts de reparación*, representan secuencias de operaciones de transformación atómicas que producen nuevos UCMs que se corresponden con la nueva implementación de la revisión actual del sistema.

IV. Resultados Experimentales

Para evaluar el enfoque *ArchLearner* se realizó un estudio retrospectivo sobre 2 proyectos de software que utilizaron diagramas UCM como parte de su documentación y se los ejercitó utilizando la herramienta desarrollada en este trabajo. La Tabla I describe ambos casos de estudio: un proyecto académico denominado Universidad3D, y un proyecto comercial llamado InQuality.

TABLA I
CASOS DE ESTUDIO PARA EVALUAR ARCHLEARNER

Caso de Estudio	Dominio	Implementación	Arquitectura
Universidad 3D	Educación, juego de realidad virtual multijugador	Java, ~370 clases	Cliente-servidor multi-tier, motor 3D
InQuality	Sistema de gestión de la información, procesos de control de calidad	Java, ~1096 clases	Cliente-servidor Web, sistema basado en eventos

El objetivo del experimento es medir la precisión de *ArchLearner* en la detección de cambios arquitectónicos en forma de diagramas UCM y determinar la calidad de las sugerencias de reparación. Para esto se hizo uso de un conjunto de revisiones de software de los sistemas involucrados, y por cada revisión, se evaluó (i) número y calidad de las responsabilidades encontradas durante la revisión de la documentación de UCM como un indicador de precisión del diagnóstico, y (ii) número y calidad de las soluciones propuestas por la herramienta para sincronizar la documentación de UCM actual como indicador de precisión de las sugerencias. Para evaluar el nivel de detección de cambios arquitectónicos, se midió la precisión del diagnóstico realizado por la herramienta con base en la cantidad de responsabilidades UCM presentadas al arquitecto como responsabilidades afectadas en la revisión del sistema estudiado (en contraparte a analizar todas las responsabilidades UCM y mapeos a código manualmente). Asimismo, también se evalúa la calidad de las soluciones propuestas en base al número de sugerencias erróneas que la herramienta no debiera incluir y sugerencias válidas esperadas incluidas en el conjunto solución.

A. Métricas Utilizadas

Para medir el cumplimiento de los objetivos propuestos, definimos la precisión del diagnostico (2) y la precisión de las sugerencias (3):

$$\textit{Precisión del diagnóstico} = RA / (RA + RE) \quad (2)$$

donde RA representa la cantidad de responsabilidades de UCM diagnosticadas correctamente como responsabilidades afectadas, y RE representa la cantidad de responsabilidades de UCM diagnosticadas erróneamente como responsabilidades afectadas.

$$\textit{Precisión de las sugerencias} = SA / (SA + SE) \quad (3)$$

donde SA representa las sugerencias válidas esperadas en el conjunto de scripts de actualización, y SE representa las sugerencias erróneas incluidas en el conjunto de scripts de actualización.

En las métricas de precisión no se incluyen casos negativos, ya que en los experimentos realizados no existen casos donde no haya cambios. En todas las revisiones seleccionadas se tiene la certeza de que hay, al menos, un cambio arquitectónico.

B. Setup del Experimento

El experimento se llevó a cabo según la descripción del enfoque y para cada caso de estudio se siguieron los siguientes pasos. El **paso I** (especificación arquitectónica) consistió en la simulación de la evolución arquitectónica tomando revisiones sucesivas del repositorio de código de cada proyecto. El criterio para elegir una revisión determinada se basó en la cantidad de cambios relevantes a nivel arquitectónico. Se entiende por cambio relevante arquitectónicamente a aquellos cambios que tienen impacto en el diseño arquitectónico, y por lo tanto, en la documentación arquitectónica en forma de diagramas de UCM. Entonces, se diseñó una secuencia de revisiones a estudiar y se reprodujeron los cambios de una versión a la siguiente estudiando estas revisiones. El **paso II** (etapa de entrenamiento) se utilizaron los casos de prueba disponibles por cada revisión del sistema para generar una base de conocimiento de cada revisión. Para revisiones avanzadas del caso de estudio se utiliza información histórica de las revisiones anteriores generando la base de conocimiento de la revisión. El **paso III** (etapa de diagnóstico de la documentación UCM actual) consistió en el uso de la base de conocimiento correspondiente a la revisión para la detección de responsabilidades UCM afectadas. Por último, el **paso IV** (etapa de sincronización de UCMs) involucró la aplicación de los scripts de reparación sugeridos por la herramienta en base al diagnóstico realizado en el **paso III** para re-sincronizar la especificación arquitectónica en forma de diagramas de UCM con los cambios actuales de la revisión.

En cuanto a la granularidad de los mapeos utilizados, cabe destacar que no se realizó ninguna restricción. Es decir, la granularidad del mapeo puede ser 1…1 (granularidad fina) o 1…N (granularidad gruesa). Granularidad fina significa que cada responsabilidad UCM se mapea directamente con sólo un método. De lo contrario, la granularidad gruesa implica que una responsabilidad se mapea con uno o más métodos. Por esta razón, no fue necesario realizar ninguna modificación a la documentación UCM provista en cada revisión de los casos de estudio para la experimentación. A continuación, se describen los casos de estudio que permitieron la evaluación de la herramienta.

C. Caso de Estudio #1: Universidad3D

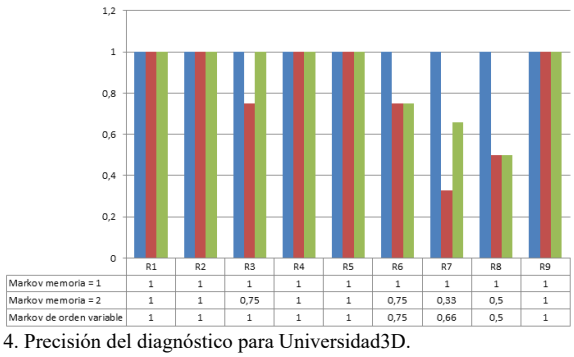
Universidad3D es un juego educativo en tres dimensiones que simula un campus virtual, permitiendo a los estudiantes navegar las instalaciones del campus universitario y de forma interactiva aprender sobre la oferta académica. Este experimento se centró en el análisis del diseño arquitectónico del servidor, cuyas principales funciones son: procesamiento de las solicitudes de los clientes, la ejecución de la lógica del juego, y la gestión de datos.

La Tabla II resume la cronología de revisiones seleccionadas para el proyecto Universidad3D junto a los impactos arquitectónicos que deberían ser detectados por *ArchLearner*.

TABLA II
SECUENCIAS DE VERSIONES PARA UNIVERSIDAD3D

Versión	Características arquitectónicas agregadas o modificadas	#clases afectadas	Elementos arquitectónicos involucrados
R0	Inicio del proyecto.	N/A	N/A
R1	Se completó la codificación de la comunicación entre el cliente y el servidor.	277	4 responsabilidadades 3 componentes
R2	Registración de un jugador y lógica de juego.	277	2 responsabilidadades 2 componentes
R3	Se agregó soporte para nuevos jugadores.	276	4 responsabilidadades 4 componentes
R4	Funcionalidad para cerrar la ventanas de chat.	308	3 responsabilidadades 2 componentes
R5	Se incluyó soporte de estados separando el login del juego.	276	5 responsabilidadades 3 componentes
R6	Funcionalidad para cambiar de avatar por medio del paso de mensajes entre el cliente y el servidor.	277	4 responsabilidadades 3 componentes
R7	Adición de la function de camarin para personalizar el avatar.	392	4 responsabilidadades 4 componentes
R8	Se entra al mundo sin necesidad de estar conectado a Internet	375	4 responsabilidadades 2 componentes
R9	Registración de usuario y comunicación VoIP	371	3 responsabilidadades 3 componentes

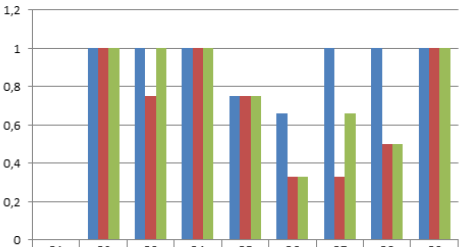
En general, la técnica más sencilla de todas, Markov de orden fijo con memoria 1, dio los mejores resultados en términos de precisión. Esto significa que en casos más complejos, el número de responsabilidades de UCM sigue siendo relativamente bajo lo que no permite sacar provecho realmente de órdenes mayores a 1 (Fig. 4 y Fig. 5). En particular en el caso de Universidad3D, la mayor parte las modificaciones consistieron en la adición de nuevos mapeos a código de responsabilidades existentes.



	R1	R2	R3	R4	R5	R6	R7	R8	R9
Markov memoria = 1	1	1	1	1	1	1	1	1	1
Markov memoria = 2	1	1	0,75	1	1	0,75	0,33	0,5	1
Markov de orden variable	1	1	1	1	1	0,75	0,66	0,5	1

Fig. 4. Precisión del diagnóstico para Universidad3D.

Por esta razón, los resultados en términos de precisión resultaron ser equivalentes utilizando distintas configuraciones de Markov, producto de que al detectar un nuevo símbolo en una responsabilidad existente la estrategia seguida por *ArchLearner* es independiente a la memoria del modelo (Fig. 2). Por otro lado, en este caso se notó la falta de pruebas automatizadas para capturar información de ejecución en el sentido de que en algunas revisiones no fue posible reproducir los escenarios documentados. Si bien esto puede ser el resultado de una implementación incompleta con alguna de las revisiones, esto provocó una caída importante en la precisión de la herramienta siempre que esta información no estuvo disponible. Seguramente, si se hubiera contado con alguna herramienta similar a *ArchLearner* se hubiera evitado marcar alguna de estas revisiones como estable.



	R1	R2	R3	R4	R5	R6	R7	R8	R9
Markov memoria = 1	0	1	1	1	0,75	0,66	1	1	1
Markov memoria = 2	0	1	0,75	1	0,75	0,33	0,33	0,5	1
Markov de orden variable	0	1	1	1	0,75	0,33	0,66	0,5	1

Fig. 5. Precisión de las sugerencias para Universidad3D.

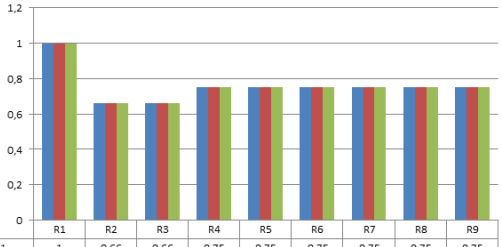
D. Caso de Estudio #2: InQuality

InQuality es una plataforma Web comercial para la gestión de calidad empresarial desarrollado por la empresa Analyte[1]. Básicamente, InQuality es un framework utilizado para mantener y gestionar de forma consistente los documentos de una empresa.

El núcleo incluye funciones para control de la documentación, auditoría, cálculo automático de indicadores, seguimiento y la supervisión de las actividades empresariales. La Tabla III describe los cambios arquitectónicos que se introducen en las revisiones, junto con las clases Java y elementos arquitectónicos involucrados en dichos cambios.

En el caso de InQuality, las Fig. 6 y Fig. 7 ilustran resultados mucho más uniformes en comparación con Universidad3D. Principalmente, porque en InQuality se extendió la funcionalidad del sistema haciendo uso de un framework que dictó las reglas de cómo se debía extender la aplicación. Al estudiar versiones estables de InQuality, los cambios observados fueron equivalentes arquitectónicamente en todas las revisiones.

Las altas de responsabilidades son independientes a la configuración de Markov seleccionada, lo que provocó que no exista variación en términos de precisión entre las distintas configuraciones de Markov.



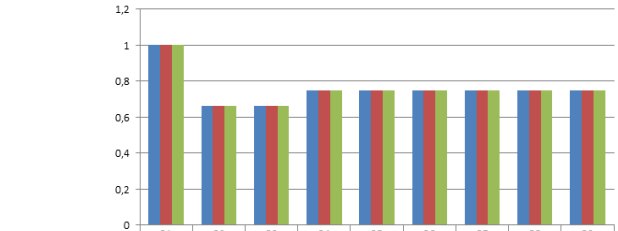
	R1	R2	R3	R4	R5	R6	R7	R8	R9
Markov memoria = 1	1	0,66	0,66	0,75	0,75	0,75	0,75	0,75	0,75
Markov memoria = 2	1	0,66	0,66	0,75	0,75	0,75	0,75	0,75	0,75
Markov de orden variable	1	0,66	0,66	0,75	0,75	0,75	0,75	0,75	0,75

Fig. 6. Precisión del diagnóstico para InQuality.

[1] http://www.analyte.com/

TABLA III
SECUENCIAS DE VERSIONES PARA INQUALITY

Versión	Características arquitectónicas agregadas o modificadas	#clases afectadas	Elementos arquitectónicos involucrados
R0	Inicio del proyecto.	N/A	N/A
R1	Se agregó soporte para la remoción de usuarios del sistema.	36	3 responsabilidades 3 componentes
R2	Se agregó soporte para la remoción de un país del sistema.	43	2 responsabilidades 3 componentes
R3	Se agregó funcionalidad para agregar una ciudad al sistema.	21	2 responsabilidades 3 componentes
R4	Se agregó funcionalidad para modificar y actualizar datos de una provincial al sistema.	40	3 responsabilidades 3 componentes
R5	Se agregó funcionalidad para agregar un nuevo tipo de moneda al sistema.	30	3 responsabilidades 3 componentes
R6	Se agregó funcionalidad para agregar nuevos tipos de documentos al sistema.	45	3 responsabilidades 3 componentes
R7	Se agregó soporte para la remoción de una ciudad del sistema.	7	3 responsabilidades 3 componentes
R8	Se agregó soporte para la remoción de un tipo de documento del sistema.	11	3 responsabilidades 3 componentes
R9	Se agregó funcionalidad para agregar un nuevo tipo de impuesto al sistema.	24	3 responsabilidades 3 componentes



	R1	R2	R3	R4	R5	R6	R7	R8	R9
Markov memoria = 1	1	0,66	0,66	0,75	0,75	0,75	0,75	0,75	0,75
Markov memoria = 2	1	0,66	0,66	0,75	0,75	0,75	0,75	0,75	0,75
Markov de orden variable	1	0,66	0,66	0,75	0,75	0,75	0,75	0,75	0,75

Fig. 7. Precisión de las sugerencias para InQuality.

V. DISCUSIÓN

En los casos de estudio presentados, no se obtuvieron diferencias significativas con respecto a los diferentes modelos de markov analizados. En general, la técnica más sencilla de todas, Markov de orden fijo con memoria uno, dio los mejores resultados en términos de precisión. Esto significa que, en casos más complejos, el número de responsabilidades de UCM sigue siendo relativamente bajo lo que no permite sacar provecho realmente de órdenes mayores a uno.

En particular en el caso de U3D, la mayor parte las modificaciones consistieron en la adición de nuevos mapeos a código de responsabilidades existentes. Por esta razón los resultados en términos de precisión resultaron ser equivalentes utilizando distintas configuraciones de Markov, producto de que al detectar un nuevo símbolo en una responsabilidad existente la estrategia seguida por ArchLearner es independiente a la memoria del modelo.

Al estudiar versiones estables de InQuality, los cambios observados fueron equivalentes arquitectónicamente en todas las revisiones, consisten básicamente en altas de responsabilidades. Las altas de responsabilidades son independientes a la configuración de Markov seleccionada, lo que provocó que no exista variación en términos de precisión entre las distintas configuraciones de Markov, dado que los modelos como Markov no aportan información sobre nuevas observaciones.

VI. CONCLUSIONES

En este trabajo hemos presentado *ArchLearner*, una herramienta basada en modelos de Markov para la sincronización asistida de la documentación en forma de UCMs con su respectiva implementación. El objetivo de *ArchLearner* es ayudar a los arquitectos de software a prevenir problemas de erosión arquitectónica. La evaluación de *ArchLearner* con casos de estudio ha mostrado resultados alentadores. Un beneficio percibido fue su capacidad de identificar cambios relevantes arquitectónicamente de forma automática. De esta manera, el arquitecto no queda abrumado por el número de cambios introducidos en cada revisión del sistema. Asimismo, las sugerencias ayudan a identificar cambios significativos y posiblemente violaciones arquitectónicas sin necesidad de recaer en detalles de implementación.

Se evaluaron dos casos de estudio, utilizando configuraciones de cadenas de Markov de primero y segundo orden y de orden variable. A partir de los proyectos analizados, no se encontraron diferencias significativas que justifiquen el uso de cadenas de Markov de orden variable en lugar de cadenas de Markov de primer orden. Esto es debido, principalmente, a que el bajo número de responsabilidades de UCM no justifica el uso de memorias de órdenes mayores.

Como trabajo futuro, apuntamos a definir un conjunto de anotaciones especiales que sirvan para mapear clases a componentes, métodos a responsabilidades, y otro conjunto de anotaciones que sirvan para definir pre y post condiciones. En este caso, las sugerencias de *ArchLearner* estarían dirigidas a actualizar estas anotaciones en lugar de los propios UCMs. En segundo lugar, para mitigar la falta de completitud en trazas de ejecución se podría incorporar información estructural del sistema. Finalmente, planificamos extender *ArchLearner* con técnicas como redes bayesianas [38], las utilizadas en la corrección de texto corrupto [40] o la detección de relaciones dentro de una cadena de ADN [16]. En general, estas técnicas siguen las etapas de entrenamiento y diagnóstico utilizadas en *ArchLearner* y podrían extender las etapas de entrenamiento

diagnóstico y sincronización de UCMs donde el arquitecto de software, según la complejidad del sistema evaluado, pueda seleccionar la técnica que mejor se adapte a sus necesidades.

REFERENCIAS

[1] L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice. 2ed*. Addison-Wesley, 2003.

[2] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, J., and R. Little, *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.

[3] M. Christeron, I. Jacobson, and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[4] A. Amandi, and M. G. Armentano, "Modeling sequences of user actions for statistical goal recognition". *User Modeling and User-Adapted Interaction*. 2011.

[5] E. Scott, A. Soria, and M. Campo, "A Taxonomy-based Approach For Fault Localization In Service-Oriented Applications". *IEEE Latin America Transactions*, 14(5), 2348-2354. 2016.

[6] T. Richner, "Using recovered views to track architectural evolution". In *ECOOP Workshops*, Lisbon, Portugal, Jun. 14-18. 1999.

[7] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture". In *ACM Sigplan Notices*, 40 (10), 167-176. 2005.

[8] N. Medvidovic, A. Egyed, and P. Gruenbacher, "Stemming Architectural Erosion by Coupling Architectural Discovery and Recovery" In *STRAW* 3, Portland, Oregon, USA, May 9. 2003

[9] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan, "Discovering architectures from running systems using colored petri nets". URL: http://www.cs.cmu.edu/able/publications/discotect-jp05. 2005.

[10] M. Abi-Antoun, J. Aldrich, D. Garlan, B. Schmerl, N. Nahas, and T. Tseng, "Improving system dependability by enforcing architectural intent". In *ACM SIGSOFT Software Engineering Notes* (Vol. 30, No. 4, pp. 1-7). 2005

[11] N. Medvidovic, and V. Jakobac, "Using software evolution to focus architectural recovery". *Automated Software Engineering*, 13(2), 225-256. 2006.

[12] P. Bourque, R. Dupuis, A. Abran, J.W. Moore, L. Tripp. "The guide to the software engineering body of knowledge". *IEEE software*, 16(6), 35-44. 1999.

[13] A. Egyed, "A scenario-driven approach to trace dependency analysis". *IEEE Transactions on Software Engineering*, 29(2), 116-132. 2003.

[14] J. Pearl, "Causality: models, reasoning, and inference." *Econometric Theory*, 19 (46), 675-685. 2003.

[15] D. Ron, Y. Singer, and N. Tishby, "Learning probabilistic automata with variable memory length" In *Proceedings of the seventh annual conference on Computational learning theory*, New Brunswick, USA, July 12-15. 1994.

[16] G. Bejerano, "Automata learning and stochastic modeling for biosequence analysis". *Hebrew University of Jerusalem*. 2003.

[17] J. Keim, and A. Koziolek, "Towards Consistency Checking Between Software Architecture and Informal Documentation". In *2019 IEEE International Conference on Software Architecture Companio,* Hamburg, Germany, Mar. 25-26. 2019.

[18] B. Uzun, and B. Tekinerdogan, "Architecture conformance analysis using model-based testing: A case study approach". *Software: Practice and Experience*, 49(3), 423-448, 2019.

[19] C. E. P. Tenemaza, and E.M.I Ortega, "State of Art, Reliability In Electrical Distribution Systems Based On Markov Stochastic Model". *IEEE Latin America Transactions*, 14(2), 799-804, 2016.

[20] M. A. Alzate, "Exact statistics of a complex Markov chain through state reduction: A satellite on-board switching example". *IEEE Latin America Transactions*, 8(4), 403-409, 2010.

Guillermo Rodríguez es actualmente miembro del Instituto de Ingeniería de Software Tandil, Argentina, Investigador Adjunto de CONICET y profesor de UNCPBA. Se graduó de Ingeniero de Sistemas en 2010 (UNCPBA), y Doctor en ciencias de la Computación en 2014 (UNCPBA). Sus principales áreas de investigación son, Realidad Virtual, Desarrollo de Software Orientado a Servicios y Razonamiento Basado en Casos.

Marcelo Armentano es actualmente miembro del Instituto de Ingeniería de Software Tandil, Argentina, Investigador Adjunto de CONICET y profesor adjunto de UNCPBA. Se graduó de Ingeniero de Sistemas en 2003 (UNCPBA), y Doctor en ciencias de la Computación en 2008 (UNCPBA). Sus intereses de investigación incluyen Sistemas Inteligentes, Sistemas de Recomendación, Filtrado de información y Minería de datos.

Álvaro Soria es investigador de la Facultad de Ciencias Exactas de Universidad Nacional del Centro de la Provincia de Buenos Aires (UNCPBA-Argentina) desde 2001. Obtuvo el título de Ingeniero de Sistemas (UNCPBA) en el año 2001 y el título de Doctor en Ciencias de la Computación (UNCPBA) en el año 2009. Sus principales áreas de investigación son Arquitecturas de Software, Diseño dirigido por la Calidad, Frameworks Orientados a Objetos y Localización de Fallas.

Emilio Corengia obtuvo el título de Ingeniero de Sistemas en el año 2014 por la Facultad de Ciencias Exactas de Universidad Nacional del Centro de la Provincia de Buenos Aires (UNCPBA-Argentina).