Ejecución de operaciones de un esquema conceptual de forma persistente y consistente

Doctorando: Xavier Oriol.
Director: Ernest Teniente [*]

Departamento de Servicios e Ingeniería de Sistemas
Universitat Politècnica de Catalunya – BarcelonaTech
{xoriol,teniente}@essi.upc.edu

Resumen El objetivo de la presente tesis es la ejecución de operaciones de un esquema conceptual de forma *persistente* y *consistente*. Es decir, los efectos de las operaciones son persistidos automáticamente en una base de datos a la vez que se garantiza que su ejecución no conlleva, en ningún caso, la violación de restricciones de integridad definidas en el esquema. De esta forma, se automatiza la construcción de las capas de dominio y persistencia de un Sistema de Información requiriendo únicamente la implementación de un interfaz gráfica para su uso.

1 Introducción

Un esquema conceptual es la definición de un Sistema de Información (SI) en términos de qué datos debe almacenar dicho SI y qué operaciones dispondrán sus usuarios para manipularlos. Siendo los esquemas conceptuales definidos con lenguajes formales tales como UML/OCL, se erige lícitamente la siguiente cuestión: ¿Puede un esquema conceptual ser ejecutado? Es decir, ¿es posible *compilar* un esquema conceptual para obtener código o, alternativamente, es posible *interpretar* un esquema conceptual como si fuera código?

Desde los años 60 que se pretende dar respuesta a semejante reto [1]. A tal fin, diferentes propuestas han sido realizadas con distinto matiz. Por ejemplo, la propuesta del OMG, el Model Driven Architecture (MDA), se basa en compilar un esquema conceptual a código a través de una serie de transformaciones automáticas [2]. Por otro lado, el Conceptual Schema-Centric Development [3] contempla la opción de interpretar, directamente, un esquema conceptual como si fuera código.

El objetivo de la presente tesis se enmarca en la segunda vertiente: la *interpretación* de esquemas conceptuales. Dicho de otra forma, nuestra propuesta pretende, dado un esquema conceptual, ejecutar directamente sus operaciones sin necesidad de compilarlo a un lenguaje máquina.

[*] Este trabajo ha sido parcialmente financiado por el Ministerio de Ciencia e Innovación bajo el proyecto TIN2011-24747 y la beca FI de la Secreteria d' Universitats i Recerca de la Generalitat de Catalunya.

2 Hipótesis de Partida y Objetivos

Nuestra propuesta parte de la idea de que un esquema conceptual definido en un lenguaje formal como UML/OCL es lo suficientemente preciso como para poder ser ejecutado.

Dicho supuesto está corroborado por la existencia de herramientas de animación de esquemas conceptuales como USE [4]. Dichas herramientas son aplicaciones que, dado un esquema conceptual, permiten ejecutar sus operaciones manteniendo el estado de los datos en memoria. Obsérvese que una herramienta de animación no cumple nuestro propósito puesto que los datos no son persistidos en ninguna Base de Datos (BD) y que, además, su eficiencia, en cuanto a la comprobación de restricciones de integridad se refiere, está en entredicho [5].

Sin embargo, comprobar la satisfacción de una restricción de integridad debería poder realizarse de forma eficiente. Primero, por la existencia de trabajos específicamente enfocados a la implementación eficiente de dichas comprobaciones [6]. Segundo, porque se puede reducir el problema de comprobar una restricción de integridad a una consulta SQL [7].

Además, con la existencia de técnicas de persistencia automática (e.g. Hibernate), no parece que se debieran de padecer problemas en persistir los datos.

Todo ello nos lleva a la definición de nuestro objetivo principal:

Objetivo 1 *Proporcionar un entorno en el que un usuario pueda cargar un esquema conceptual UML/OCL y ejecutar sus operaciones persistiendo los cambios en una BD garantizando que ninguna restricción sea violada.*

Este objetivo abstracto se desglosa en los siguientes subobjetivos concretos:

Subobjetivo 1 *El entorno debe asegurar la satisfacción de toda restriccion mediante una política de* checking, maintenance *o* enforcement.

Existen diferentes formas de asegurar la no violación de ninguna restricción [3]. La política de *checking* consiste en comprobar la no violación de ninguna restricción una vez ejecutada la operación. El *maintenance* se caracteriza por, al detectar una violación después de ejecutar una operación, aplicar los cambios necesarios en los datos hasta alcanzar un nuevo estado que sea consistente. Finalmente, el *enforcement* es la reescritura de las operaciones de tal forma que se asegure que, al ser ejecutadas, no se incurre nunca en violación alguna.

Subobjetivo 2 *El entorno debe aceptar operaciones tanto de inserción, como de eliminación y modificación de datos, y debe ser validado con un caso de estudio real.*

Nuestro objetivo es alcanzar un resultado utilizable en la producción real de *software*. Para ello, es necesario tener en cuenta no solo las operaciones de inserción de datos, sino también las de eliminación y modificación de datos. Para validar su correcto funcionamiento, se vuelve indispensable su puesta a prueba con un caso de estudio real.

3 Aportación

Para alcanzar los anteriores objetivos proponemos un tipo de reglas lógicas que denominamos RGDs (Repair-Generating Dependencies). Un RGD es una regla lógica que permite detectar qué cambios en los datos originan una violación de una restricción de integridad, y, en la medida de lo posible, qué cambios adicionales en los datos se pueden aplicar para reparar dicha violación.

Formalmente, un RGD tiene la siguiente forma: $l_1 \wedge ... \wedge l_n \rightarrow r_1 \vee ... \vee r_m$; donde l_i es un literal que representa un dato de la base de información, su inserción o borrado, mientras que r_j representa únicamente la inserción o borrado de datos. Por ejemplo, el siguiente RGD:

$$ιpremiumUser(X) \wedge claims(X, Y) \wedge \neg prioritized(Y) \rightarrow δclaims(X, Y) \vee ιprioritized(Y)$$

Indica que se produce una violación cuando se inserta un nuevo usuario X como *premium*, éste tiene una reclamación Y, e Y no es una reclamación prioritaria. Dicha violación se puede reparar o bien eliminando la reclamación o bien insertándola como prioritaria.

RGDs para codificar restricciones. El anterior ejemplo ilustra la codificación de una restricción como un RGD. Sin embargo, normalmente una restricción se debe traducir a más de un RGD puesto que existe más de una combinación de inserciones/borrados de datos que producen la violación de la misma restricción. En un trabajo previo, hemos definido la forma de automatizar la conversión de una restricción escrita en OCL a sus diferentes RGDs [8].

Ejecutando los RGDs mediante el algoritmo *chase*, se puede obtener, dado un conjunto de inserciones/borrados de datos, un superconjunto de estos mismos inserciones/borrados tal que no viola ninguna restricción de integridad. Es decir, aplicar el *chase* sobre los RGDs equivale a mantener la consistencia de los datos con una política de *maintenance*.

Más aún, los RGDs pueden ser configurados para personalizar la forma de reparación de una restricción. En el anterior ejemplo, podría considerarse el caso que reparar la violación eliminando la reclamación de un usuario *premium* no es conveniente. Siendo así, se puede reescribir el RGD de la siguiente forma:

$$ιpremiumUser(X) \wedge claims(X, Y) \wedge \neg prioritized(Y) \wedge \neg δclaims(X, Y) \rightarrow ιprioritized(Y)$$

El RGD solo permite ahora reparar la restricción añadiendo las reclamaciones del usuario *premium* como prioritarias. Más aún, el RGD puede ser configurado para no forzar reparación alguna:

$$ιpremiumUser(X) \wedge claims(X, Y) \wedge \neg prioritized(Y) \wedge \neg δclaims(X, Y) \wedge \neg ιprioritized(Y) \rightarrow \bot$$

En tal caso, $\bot$ simboliza que, de satisfacerse el antecedente, se produce una violación irreparable y se advierte de ello al usuario. En otras palabras, el RGD ha sido configurado para ejercer la política de *checking*.

RGDs para codificar operaciones. Más interesante aún, los RGDs también pueden codificar las operaciones. Una operación muestra el siguiente comportamiento: si cuando se llama la operación, se satisface la precondición, entonces se ejecutan los efectos establecidos en la postcondición. Este comportamiento se

puede simular escribiendo un RGD que contenga, en su antecedente, la llamada a la operación junto a su precondición, y, en su consecuente, los efectos de la postcondición. E.g:

$$userUpgradeCall(X) \wedge \neg problematicUser(X) \rightarrow premiumUser(X)$$

El anterior RGD codifica que, si al llamarse la operación *userUpgrade* con un usuario *X* tal que no es problemático (precondición), se inserta *X* como usuario *premium* (postcondición).

Al ejecutarse este RGD, se puede violar la restricción anteriormente presentada acerca de la no existencia de reclamaciones no prioritarias para un usuario *premium*. En dicho caso, dependiendo de si la restricción ha sido configurada para la política de *checking* o *maintenance*, se advertirá al usuario de una violación o se añadirán las cambios necesarios para evitarla.

Sin embargo, existe la posibilidad de configurar la operación para ejercer la política de *enforcement*, es decir, se puede modificar de tal forma que se asegure la no violación de la restricción. Esto se puede conseguir a través de la modificación del antecedente (la precondición) o del consecuente (la postcondición). E.g:

$$userUpgradeCall(X) \wedge \neg problematicUser(X) \neg claims(X, Y) \rightarrow premiumUser(X)$$

Ahora la operación no viola la restricción puesto que se ha añadido la precondición que un usuario, para poder ser *premium*, primero tiene que tener cerradas todas sus reclamaciones pendientes.

4 Plan de Trabajo

1. Conseguir un caso de estudio real Nuestro trabajo debe partir de un caso de estudio real, si bien es difícil al ser las empresas reacias a ofrecer sus artefactos de desarrollo. Más aún, estamos en busca de un esquema conceptual con restricciones bien documentadas y no un mero diagrama de clases.

2. Codificar sus restricciones como RGDs Si bien ya hemos definido una traducción de un subconjunto del OCL a RGDs, estaría por ver qué restricciones no pueden ser convertidas automáticamente con esta propuesta. Nuestro objetivo aquí es estudiar como traducir las demás restricciones, qué limitaciones ofrece la traducción, y analizar empíricamente la eficiencia de las políticas de *checking* y *maintenance*. Se espera una eficiencia razonable, al menos, en el caso de *checking*.

3. Codificar sus operaciones como RGDs Seguidamente, se procedería a definir una traducción de las operaciones a RGDs. Se espera poder reaprovechar parte de la actual traducción de restricciones OCL a RGDs. Una vez capturadas las operaciones del caso de estudio como RGDs, se analizaría como modificar los RGDs automáticamente para implementar la política de *enforcement*. Esto es, como añadir nuevas condiciones en el antecedente del RGD o como añadir nuevas acciones en al consecuente para asegurar la no violación de ninguna restricción.

4. Implementar una capa de persitencia de datos Finalmente, se procedería a implementar una capa de persistencia automática de datos. Gracias a la existencia hoy en día de técnicas de persistencia automática de datos, no se esperan problemas en esta fase.

5 Relevancia

Importancia a la sociedad No hay duda de que la automatización de la construcción de un *software* a partir de su esquema conceptual es de sumo interés para la industria. Prueba de ello es el impulso que ha recibido en los últimos años el MDA/MDD. Nuestra propuesta, sin embargo, no se basa en la transformación de modelos, sino en la directa interpretación de ellos.

Importancia científica Des de la óptica de la comunidad científica, no sólo se contribuye a la resolución de un problema planteado durante los años 60, sino que nuestra propuesta avanza en consonancia con otros problemas paradigmáticos.

Por ejemplo, el problema de *state reachability*, saber si un esquema conceptual admite un estado que tenga, por lo menos, unas instancias I dadas por el usuario, se puede reducir a *integrity maintenance*. Intuitivamente, solo se deben configuran los RGDs para quitar las reparaciones consistentes en borrar datos dejando únicamente las de inserción, y se considera entonces la inserción de todo I. El problema de *state reachability* es, a su vez, un problema clave para resolver otros como son la redundancia/contradicción de restricciones, entre otros.

6 Conclusiones

Hemos presentado un método para la ejecución consistente y persistente de esquemas conceptuales. Nuestro método se basa en convertir las restricciones y operaciones de un esquema conceptual a unas reglas lógicas que llamamos RGDs. Dichos RGDs pueden ser configurados para implementar las diferentes políticas de consistencia de *checking*, *maintenance* y *enforcement*. Esta solución no solo permite la ejecución consistente de esquemas conceptuales sino que avanza en la línia con otros problemas como el *state reachability*, que a su vez, es clave para resolver la redundancia o contradicción de restricciones, entre otros.

Referencias

1. Teichroew, D., Sayani, H.: Automation of system building. Datamation **17**(16) (1971) 25–30
2. Soley, R., et al.: Model Driven Architecture. OMG white paper **308** (2000) 308
3. Olivé, A.: Conceptual schema-centric development: A grand challenge for information systems research. In: Advanced Information Systems Engineering. Volume 3520 of Lecture Notes in Computer Science. Springer (2005) 1–15
4. Hamann, L., Hofrichter, O., Gogolla, M.: On integrating structure and behavior modeling with OCL. In: International Conference on Model Driven Engineering Languages & Systems (MODELS 2012), Springer (2012) 235–251
5. Clavel, M., Marina, E., Miguel Angel, G.d.D.: Building an efficient component for ocl evaluation. Electronic Communications of the EASST **15** (2008)
6. Cabot, J., Teniente, E.: Incremental integrity checking of UML/OCL conceptual schemas. Journal of Systems and Software **82**(9) (2009) 1459–1478
7. Egea, M., Dania, C., Clavel, M.: MySQL4OCL: A stored procedure-based MySQL code generator for OCL. Electronic Communications of the EASST **36** (2010)
8. Oriol, X., Tort, A., Teniente, E.: Fixing up non-executable operations in UML/OCL conceptual schemas. In: Conceptual Modeling–ER 2014. Springer (to appear) (2014)