Captura del razonamiento y evolución de arquitecturas de software mediante la aplicación de operaciones arquitectónicas orientadas a objetivos

María Luciana Roldán, Silvio Gonnet, y Horacio Leone

INGAR (UTN-CONICET), Facultad Regional Santa Fe, UTN, Santa Fe, Argentina
{lroldan, sgonnet, hleone}@santafe-conicet.gov.ar

Resumen. Durante el proceso de diseño de una arquitectura de software, los actores que participan del mismo toman numerosas decisiones de diseño. Para comprender cómo surgió una determinada arquitectura de software, es importante conservar esas decisiones de diseño junto con el razonamiento que las explican. Sin embargo, este conocimiento generalmente se pierde, ya que su representación o documentación es una carga extra de trabajo para quienes lo producen. Para superar este problema, se propone un modelo de operaciones que permite por un lado la definición de operaciones que codifican decisiones de diseño recurrentes, y por otro lado, su aplicación para materializar las decisiones de diseño durante el proceso de construcción de la arquitectura de software. El modelo posibilita definir operaciones orientadas a objetivos, de manera que su ejecución refleje no sólo la decisión sino también el por qué de la misma, permitiendo registrar en forma casi transparente y natural la intención del diseñador, y mantener la trazabilidad entre las decisiones, los elementos arquitectónicos afectados y los objetivos perseguidos o alcanzados.

Keywords: architecture documentation,design rationale, design decisions, goal-oriented operations

1 Introducción

Las arquitecturas de software permiten describir un sistema complejo con alto nivel de abstracción y hace posible a arquitectos y diseñadores razonar sobre las propiedades del sistema [1]. Éstas son los artefactos resultantes de un proceso de diseño, en el cual diferentes actores toman numerosas decisiones que tienen algún impacto en la arquitectura final. Para comprender porqué una arquitectura de software es como es, es importante conservar esas decisiones de diseño [2, 3, 4, 5, 6] junto con el razonamiento en las que se basan. El proceso de diseño de arquitecturas de software (SADP) comprende las primeras decisiones sobre un sistema complejo que contemplan sus requerimientos de calidad, funcionales y sus restricciones. Diseñadores, arquitectos y otros "stakeholders" han reconocido la importancia de documentar decisiones de diseño junto con su razonamiento, y no sólo capturar el artefacto arquitectónico final generado [7]. Sin embargo se reconoce que existen dificultades que inhiben la práctica de documentar las decisiones y el razonamiento de diseño, como la sobrecarga de trabajo que implica

para el diseñador. En la práctica, representar el razonamiento arquitectónico se considera una tarea intensiva en tiempo y esfuerzo, la cual no tiene grandes beneficios en el corto plazo, desde la perspectiva de los productores de este conocimiento. Por esta razón frecuentemente es evitada o realizada inadecuadamente [8, 9]. El razonamiento de diseño, sin embargo, en la práctica es pobremente documentado y frecuentemente se encuentra poco estructurado, lo que significa que reside sólo en la mente de los arquitectos, y finalmente se pierde.

Partiendo de la idea de que cada decisión tomada por el arquitecto siempre persigue la satisfacción de cierto objetivo, en este trabajo se propone un modelo que posibilita la definición y ejecución de operaciones arquitectónicas orientadas a estos objetivos. De esta manera, cada operación define la semántica de una decisión. Esta semántica está dada por: i) el posible contexto al que puede aplicarse la operación, es decir, las precondiciones que deberían cumplirse y los tipos de objetivos que se podrían alcanzar con su aplicación, ii) el impacto esperado sobre la arquitectura de software, lo cual está dado por el efecto de las acciones que forman el cuerpo de la operación sobre el diseño arquitectónico. El uso del modelo de operaciones propuesto facilita la documentación de la evolución de la arquitectura en una forma natural al proceso de diseño, y permite mantener la trazabilidad entre los elementos arquitectónicos afectados y los objetivos perseguidos o alcanzados. En trabajos anteriores [10, 11], hemos propuesto un modelo que captura cada decisión realizada en un SADP junto con los productos resultantes, haciendo posible la trazabilidad entre ellos y la reconstrucción de la historia del proceso de diseño. En el presente trabajo, dicho modelo es redefinido para posibilitar la definición y ejecución de operaciones orientadas a objetivos.

A continuación, en la sección 2, se mencionan trabajos relacionados. En la sección 3, se introduce el enfoque del modelo de representación del proceso de diseño que es la base del presente trabajo. En la sección 4, se aborda el modelo conceptual para definición y ejecución/captura de operaciones orientadas a objetivos. Además, se presenta un dominio para SADP que no sólo incluye tipos de objeto de diseño para representar los principales bloques de construcción de una arquitectura de software, sino también permite definir tipos para representar elementos relativos a razonamiento arquitectónico. Para ilustrar las capacidades de definición de operaciones se presentan algunos ejemplos de operaciones orientadas a objetivos. En la sección 5 se presenta un escenario en el cual se empleó el modelo propuesto para capturar la evolución de una arquitectura de software de un sistema que fue migrado a la *cloud*. Por último, en la sección 6 se presentan las conclusiones del trabajo.

2 Trabajos relacionados

Aportes recientes consideran a la trazabilidad como un método para seguir el proceso de desarrollo de software [12, 13, 14]. Por otro lado, varios autores se han concentrado en el modelado y captura del razonamiento para poder representar y seguir la evolución de los productos del proceso de desarrollo [15, 16]. A pesar de su importancia, existen escasas propuestas que aborden el proceso de desarrollo de forma integrada, y las existentes en su mayoría no permiten la representación del mismo junto al conocimiento empleado. Este problema ha sido reconocido por diversos autores en diferentes dominios de la ingeniería de software [2-6, 8, 9], quienes concuerdan en que las

decisiones tomadas constituyen conocimiento tácito, el cual a pesar de ser esencial para la solución alcanzada, no es explícitamente documentado, ni forma parte del artefacto resultante. Esto conduce a que en futuras revisiones o modificaciones del artefacto a construir (durante su evolución), sea difícil y costoso establecer correspondencias entre las decisiones de diseño y las razones que las respaldaron.

Otra línea vinculada con la temática es la de administración de los productos de los procesos de diseño, entendiendo aquí como productos a los modelos, datos, diagramas, programas, etc. Existen desde hace tiempo, sistemas de administración de productos y de sus versiones [17], los cuales son ampliamente utilizados en la práctica. Estos responden a la necesidad básica que los productos de un proceso de desarrollo deben ser almacenados y organizados. Los sistemas de administración de productos, así como los sistemas de administración de configuraciones de software [18], están enfocados en productos, principalmente en código fuente y software ensamblado, y no contemplan el seguimiento del proceso de diseño. Estos sistemas presentan dificultades para la posterior recuperación de información sobre qué requerimientos fueron contemplados, qué restricciones fueron adoptadas/relajadas, qué decisiones de diseño fueron realizadas, qué alternativas fueron consideradas, y cuál fue la seleccionada, durante el proceso de desarrollo.

3 Un modelo para la representación del proceso de diseño y evolución de arquitecturas de software

El modelo propuesto para definición de *operaciones orientadas a objetivos* (*OpOOs*) está basado en nuestro trabajo previo, que consiste en un modelo para la captura de procesos de diseño ingenieriles y una herramienta que lo implementa parcialmente [10, 11]. Dicho modelo posee un enfoque operacional, en donde las decisiones de diseño son materializadas en la ejecución de operaciones de diseño aplicables a los tipos de productos manejados en el dominio.
A partir de la representación del proceso de evolución mediante las operaciones ejecutadas y los productos generados, se espera poder responder a diferentes preguntas acerca de cómo se llevó a cabo dicho proceso. Por ejemplo:

i) ¿Cuáles fueron los elementos arquitectónicos afectados por la ejecución de una operación o secuencia de operaciones (decisión)?
ii) ¿Qué operaciones se ejecutaron? ¿Qué objetivos perseguían? ¿Qué requerimientos intentaban satisfacer?
iii) ¿Qué productos surgieron como consecuencia de la ejecución de una determinada operación (decisión)?
iv) ¿Qué operaciones se ejecutaron para alcanzar un cierto objetivo?
v) ¿Qué operación dio lugar a la aparición de determinado producto?
vi) ¿Qué objetivos no fueron considerados en el proceso de diseño de la arquitectura?

Para poder responder a tales cuestiones, el modelo se sustenta en un esquema de administración de versiones que permite capturar y representar la evolución de los distintos productos del diseño en forma conjunta con las operaciones aplicadas a los mismos. El modelo considera el proceso de diseño como una secuencia de actividades que operan sobre los productos del proceso de diseño, denominados objetos de diseño. Los

objetos de diseño representan modelos del artefacto que está siendo diseñado, requerimientos a cumplir, restricciones impuestas al modelo. Naturalmente, estos objetos evolucionan durante un proceso de diseño, dando lugar a múltiples versiones de los mismos. Un conjunto de estas versiones constituyen una versión del modelo, la cual describe el estado del proceso de diseño en un determinado instante. En este esquema, cada versión de modelo es generada mediante la aplicación de una secuencia de operaciones en una versión de modelo predecesora. La secuencia de operaciones puede incluir la eliminación, creación, y modificación de versiones que forman la versión de modelo predecesora. El modelo además de proveer un conjunto de operaciones básicas (*agregar*, *eliminar*, *modificar*), posibilita también la definición de operaciones más complejas. En un trabajo previo, las precondiciones y los efectos de estas operaciones fueron formalizadas empleando Cálculo Situacional [11]. En este trabajo se presenta una redefinición del modelo para la definición y ejecución de operaciones. El objetivo es representar operaciones complejas (*refinar*, *abstraer*, *aplicarMVC*, etc.), junto a los objetivos o intenciones que se persiguen con su ejecución, de manera de capturar mayor información durante dicha ejecución. Para la especificación de operaciones del dominio se propuso una gramática libre de contexto, que indica la sintaxis válida para las mismas [19], la cual guarda una estrecha relación con el modelo de definición de operaciones que se verá en la Fig. 1. El esquema de representación de versiones divide la representación de los objetos de diseño en dos niveles: i) nivel repositorio, donde se representa cada objeto de diseño por medio de un *objeto versionable*, y se mantienen las asociaciones con otros *objetos versionables* (que representan a otros objetos de diseño); ii) nivel versiones, donde se representan los distintos estados alcanzados por cada objeto de diseño durante un proyecto. Los objetos que pertenecen a este nivel se denominan *versiones de objeto*. Estas versiones de objetos pertenecen a las diferentes *versiones de modelo* que se obtienen a lo largo del proceso de diseño. Algunos de los conceptos del esquema de versionamiento (como *ObjectVersion* y *ModelVersion*) se incluyen en la Fig. 1, donde se detalla el modelo de *OpOOs*. El enfoque empleado para la administración de versiones, modela el proceso de diseño a partir de una versión de modelo inicial. Aplicando secuencias de operaciones (denominadas ϕ) que, en general, implican la creación, eliminación, o modificación de algunos de los elementos del modelo, se van obteniendo sucesivas versiones de modelo.

El modelo permite la definición de diferentes dominios de diseño. El dominio se especifica en términos de los *tipos de objetos de diseño* que se desean modelar y las *operaciones* que son aplicables a los mismos. Cada tipo de objeto de diseño posee un conjunto de *propiedades*. Dicho modelo ha sido implementado en una herramienta computacional [20] que ha permitido trabajar en diferentes casos de estudios. Antes de iniciar un proceso de diseño, un experto debe definir el dominio o seleccionar un dominio existente. En otras contribuciones hemos definido un dominio genérico para SADP que toma conceptos comunes de los lenguajes de descripción de arquitecturas de software, el método de diseño ADD [1] , de las diferentes vistas con que puede describirse una arquitectura de software [21, 22], y conceptos relativos a razonamiento arquitectónico.

4 Modelo de operaciones orientadas a objetivos

Como se introdujo previamente, una manera de incorporar razonamiento a un modelo de diseño de una arquitectura de software es hacer explícito en el mismo modelo aquellos elementos que permitan registrar el porqué de las decisiones tomadas. Para lograr esto en este trabajo se propone un modelo de operaciones (Fig. 1) que posibilita definir operaciones que al momento de ejecutarlas permitan indicar cuál es el objetivo que se desea alcanzar con dicha ejecución, de esta manera, haciendo explícita la intención del diseñador o el objetivo que desea alcanzar.

4.1. Definición del modelo de operaciones orientadas a objetivo

La clase central en el modelo de operaciones presentado en la Fig. 1 es la clase abstracta *Command*. Una *operación* (*Operation*) es un comando compuesto que posee un cuerpo formado por una lista de comandos que se ejecutan en orden al ejecutarse la misma, la cual captura información como la fecha y hora en que fue aplicada, el actor (arquitecto, diseñador, o *mantainer*) que tomó la decisión. La operación debe ser aplicada con una serie de parámetros de entrada.

La clase abstracta *DataType* generaliza los tipos de parámetros soportados por una operación, abarcando los tipos primitivos (*String*, *Integer*, *Float*, etc., que no se muestran en Fig. 1) y los tipos de objetos de diseño definidos para un dominio. El cuerpo de una operación puede estar formado por operaciones primitivas (*add*, *delete*, *modify*), funciones auxiliares, y otras operaciones del mismo dominio. Las funciones auxiliares son funciones predefinidas por el modelo (y la herramienta que lo implementa [20]), como aquellas que posibilitan la iteración de colecciones, asignación de variables, recuperación o selección de elementos de una versión de modelo o de una colección de objetos, y establecer asociaciones a nivel de repositorio. Una secuencia de operaciones (*SequenceOfOperations*), por otro lado, es una agregación de una o más operaciones ejecutadas en la transformación que da lugar a la aparición de una nueva versión de modelo. En otras palabras una instancia de *SequenceOfOperations* establece una traza entre entre una versión de modelo predecesora y una versión de modelo sucesora, y por ende constituye un elemento esencial para conservar la historia de la evolución del proceso de diseño.

Empleando este modelo y contando con un dominio que incluya los tipos de objeto de diseño apropiados es posible representar intenciones de diseño y objetivos arquitectónicos que se desean alcanzar al ejecutar una operación, estableciendo enlaces que permitan vincular los resultados de la ejecución de una operación (*Outcome*) –versiones de objeto generadas– con los objetivos perseguidos. Considerando el dominio propuesto para SADP, un "*goal*" (intención, objetivo o meta) podría ser por ejemplo un requerimiento a alcanzar, un escenario de calidad que se desea verificar, o una restricción que debe ser respetada, entre otros. El modelo de operaciones permite explicitar ese *goal* como un argumento específico de la operación (*Goal*, subclase de *Parameter* en Fig. 1). Dos son las principales ventajas del modelo de operaciones propuesto:

- Posibilita la *definición* de operaciones orientadas a objetivos. Un experto al definir una operación puede indicar cuál o cuáles son los posibles tipos de objetivos que podrían alcanzarse con la aplicación de la operación. De esta manera, este conocimiento codificado en la definición de la operación, puede ser empleado por una herramienta computacional para sugerir al arquitecto operaciones del dominio que posiblemente

puedan ser aplicadas en un contexto (versión de modelo) en donde estén presentes versiones de objetos de los tipos determinados en su definición. Por ejemplo, en el dominio podrían haberse definido operaciones como *applyMVC*, *applyTwoLayers*, *applyIntermediary*, para las cuales se indica un argumento *goal* cuyo tipo es *ModifiabilityQualityRequirement*. Entonces, en un proyecto de diseño particular, si en la versión de modelo actual se encuentra presente un requerimiento de calidad de *modificabilidad*, tales operaciones podrían ser sugeridas al diseñador para ser ejecutadas. Por otro lado, al momento de la definición de la operación también se definen los resultados esperados (clase *ExpectedOutcome*), lo que posibilita asignar un nombre y tipo a cada resultado, pudiendo éstos ser empleados, en otra operación que conforme su cuerpo (*body*).

• Posibilita la ejecución y captura de operaciones orientadas a objetivos. Bajo este enfoque, el proceso de diseño o evolución de la arquitectura de software tiene lugar a medida que el arquitecto materializa sus decisiones de diseño a través de la aplicación de *OpOOs*. Los objetivos perseguidos efectivamente en la aplicación de esa operación son provistos como un argumento al momento de ejecutar la misma (*ActualGoal*). Esto agrega razonamiento a la decisión y hace posible en una etapa posterior, mantener la traza de la historia del proceso de diseño para evaluar si ciertamente este objetivo se alcanzó y en qué medida. Cabe aclarar que al momento de la ejecución de la operación deben proveerse también el resto de los argumentos (representado en la Fig. 1 por *ArgumentValue*).

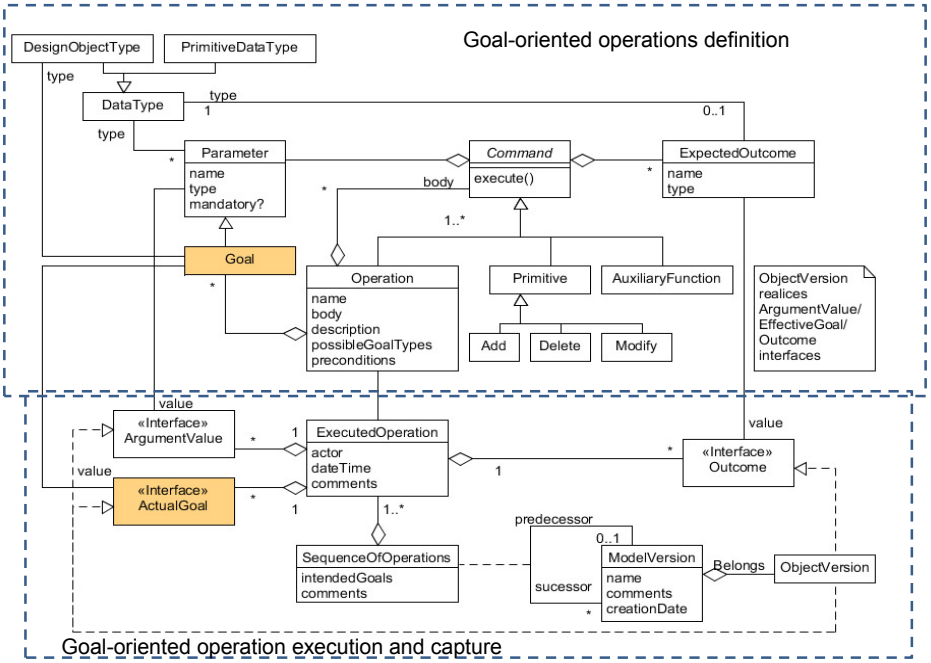

Fig. 1. Modelo para la definición y captura de operaciones orientadas a objetivos.

Por cuestiones de espacio, en el presente trabajo se omiten las restricciones OCL definidas sobre el modelo. Éstas definen que los tipos de valores efectivos de los parámetros de entrada de una operación deben coincidir con la cantidad y tipos de los parámetros

indicados en su definición, y limitan los tipos posibles de resultados (*outcomes*) de una operación ejecutada, debiendo corresponder en tipo de los indicados en su definición (*expected outcomes*).

4.2. Tipos de objeto de diseño de un dominio en cual se definirán operaciones orientadas a objetivo

En otras contribuciones se han detallado los elementos que provee el modelo para definir dominios. Además se ha propuesto un dominio genérico para SADP y su evolución [10, 11, 20, 21]. Algunos de los principales tipos de objetos de diseño y relaciones de dominio son relativos a la vista de *Components-and-Connector* (*C&C*) [11]. Dentro de esta vista se incluyen tipos de objetos de diseño como *Component*, *Connector*, *Port*, *Role* y *Attachment.* Los tipos de objeto de diseño *Characteristic* y *Responsibility* permiten representar propiedades relevantes y comprtamiento de componentes y conectores. *RHasResponsibility* es una relación de dominio que permite representar cuáles son las responsabilidades de un componente en particular. En este caso esta relación se encuentra reificada (transformada en un tipo de objeto de diseño) de manera de poder asignarle propiedades y permitir versionar objetos de tal tipo. En el dominio pueden definirse tipos de objetos de diseño diferentes vistas, como *Allocation* view y *Modules* view [21]. Estos tipos de objeto de diseño son generalizados en un tipo abstracto denominado *ArchitecturalElement* (Fig. 2). Cabe destacar que el modelo es lo suficientemente flexible para permitir a los expertos modelar todo lo que se desee representar explícitamente en un modelo de arquitectura de software. En particular, para poder modelar los "goals" de las decisiones (objetivos que se desean lograr con la ejecución de operaciones) es necesario incluir en el dominio con el que trabajará aquellos tipos de objetos de diseño para representar este tipo de conocimiento arquitectónico relativo al contexto y a razonamiento (Fig.3-a). Estos nuevos tipos están asociados a tipos de objetos de diseño derivados del tipo abstracto *ArchitecturalElement* (Fig. 2) y pueden ser identificados a partir del método de diseño que preferido por el arquitecto, como por ejemplo del método ADD [1]. Las entradas de este método son los requerimientos de calidad (*Quality requirements*) que pueden ser expresados como un conjunto escenarios de calidad más específicos (*Quality scenarios*), requerimientos funcionales (*Functional requirements*) que se traducen en la arquitectura en un conjunto de responsabilidades de sus componentes o módulos (*Responsibilities*), así como restricciones (*Constraints*).

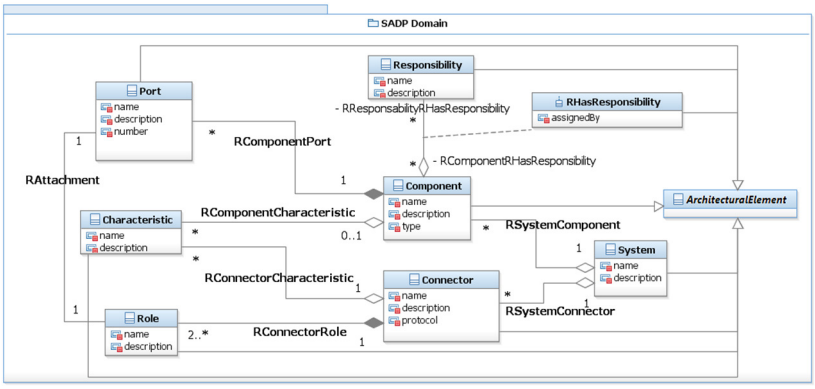

Fig. 2**.** Parte de un posible dominio para SADP que incluye elementos de la vista C&C

Los tipos de objetos de diseño definidos pueden ser especializados según se desee. Por ejemplo, *QualityRequirement* podría ser especializado en requerimientos de calidad específicos como *ModifiabilityQualityRequirement*, *PerformanceQualityRequirement*, *AvailabilityQualityRequirement*, etc. (no incluidos en la figura). Otros tipos de objetos de diseño que se incluyen en este dominio son *Assessment*, para indicar el nivel de aproximación al cumplimiento de un escenario de calidad que se tiene el artefacto arquitectónico alcanzado, y *Constraint* que permite representar explícitamente reglas o condiciones impuestas en el modelo arquitectónico del sistema a construir. *Assumption* permite representar cuestiones que se asume que existen o se verifican al momento de construir la arquitectura de software, y *Alternative* representar posibles caminos de evolución de la arquitectura del sistema. Por otro lado, *Argument* es útil para expresar las razones a favor o en contra de cierta alternativa de diseño. Tipos de objetos de diseño como los recientemente mencionados hacen posible que durante un proceso de diseño puedan agregarse, modificarse o eliminarse versiones de objeto que representan explícitamente en la arquitectura de software misma cierto conocimiento que sirve para explicar el modelo logrado o razonar sobre cómo se obtuvo el mismo. De esta manera, pueden incorporarse elementos (como requerimientos, suposiciones, restricciones, etc.), los cuales pueden ser asociados con otros elementos arquitectónicos (como componentes, conectores, responsabilidades, etc.).

En el dominio que se describió aún es necesario definir ciertas relaciones de dominio que posibiliten vincular elementos arquitectónicos con elementos relativos a razonamiento. Ejemplos de estas relaciones son *RRegardsAssumption* y *RConstraintsTo* (Fig.3-b) para indicar que cierta suposición ha sido tenida en cuenta, y que cierta restricción está limitando las posibilidades sobre un elemento o porción de un modelo de arquitectura de software. *RTradeOff* es otro tipo de objeto de diseño útil para expresar la existencia de un "trade-off" entre dos atributos de calidad. En cuanto a *RSupportsArgument* y *RRejectsArgument* hacen posible vincular un conjunto de elementos arquitectónicos (o fragmento de un modelo de arquitectura de software) con un *argumento*, es decir un motivo que justifica o rechaza la presencia de esos elementos en la arquitectura. Otras relaciones del dominio necesarias para aportar expresividad al modelo de la arquitectura de software son *RSystemAlternative*, *ROppossiteTo*, y *RSimilarTo*, cuyo significado es auto explicativo observando el modelo conceptual de la Fig. 3 (a).

Los tipos de objetos de diseño y relaciones definidos hasta aquí en el dominio no son suficientes para representar explícitamente "intenciones" de diseño u "objetivos" que se desean alcanzar. Para ello, se deben incluir otros tipos como *RPossibleScenarioSolution*, que hace posible asociar un cierto escenario de calidad con los elementos de la arquitectura que surgieron como resultado de una decisión cuyo objetivo perseguido es dicho escenario de calidad. Estas asociaciones permiten representar que existe una alta probabilidad de que determinados elementos que forman parte de la arquitectura sirvan para satisfacer el cumplimiento de un escenario de calidad. En otras palabras, se representa el conocimiento que indica que determinados objetivos han intentando satisfacerse durante el proceso de diseño, aunque aún no se ha llevado a cabo una evaluación que permita comprobar fehacientemente si se ha logrado.

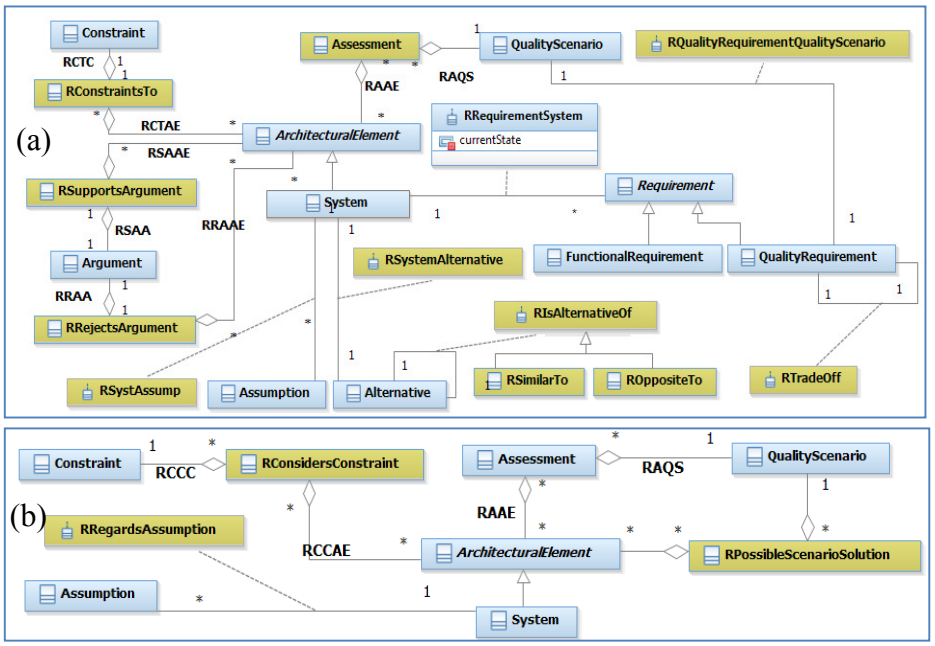

Fig. 3. Design object types to support (a) design rationale and (b) designer intentions

Tal evaluación no es parte de esta propuesta, aunque sí podría representarse el conocimiento obtenido de la aplicación de algún método o herramienta de evaluación. En forma similar, el tipo de objeto de diseño *RConsidersConstraint* (Fig. 3-b) permite expresar que una porción de diseño arquitectónico (es decir, un grupo de versiones de diversos elementos arquitectónicos) se ha incorporado teniendo en cuenta una restricción en particular que fue impuesta previamente sobre el sistema.

4.3. Ejemplos de operaciones orientadas a objetivos

Una vez que se cuenta con los tipos de objeto de diseño definidos en el dominio, tanto para representar aspectos de estructura y comportamiento de la arquitectura, como aspectos de razonamiento, se procede a definir el conjunto de operaciones arquitectónicas. Tales operaciones estarán disponibles para ser utilizadas por los arquitectos para materializar decisiones de diseño e incorporar razonamiento al proceso de diseño. Las operaciones se definen instanciando el modelo de la Fig. 1 (parte superior). Es decir para cada operación se definirá su nombre, argumentos de entrada, posibles tipos de objetivos a alcanzar con la misma, el cuerpo o lista de comandos que lo componen, y los posibles resultados a alcanzar. Cabe aclarar que el argumento "*goal*" de las operaciones puede no ser obligatorio. Dado que en el dominio pueden definirse operaciones de distinta complejidad, en aquellas operaciones más básicas puede omitirse. Esta decisión queda en manos del arquitecto que emplea la herramienta que implementa el modelo propuesto.
Las operaciones menos complejas como *addComponent*, generalmente codifican actividades de diseño simples relativas a la definición de la estructura y comportamiento de la arquitectura de software que se está diseñando. Además, entre las operaciones de menor complejidad se pueden considerar a aquellas que agregan /eliminan en el modelo ciertos

elementos de razonamiento, para documentar el proceso de diseño.

Es posible definir operaciones más complejas que codifican un patrón arquitectónico o táctica, las cuales materializan soluciones arquitectónicas recurrentes y están fuertemente relacionadas al cumplimiento de algún requerimiento de calidad. En trabajos previos, se han especificados diversos patrones arquitectónicos tomados de la bibliografía [10]. En la Fig. 4 puede verse la especificación funcional de una de estas operaciones, en este caso la aplicación del patrón MVC (*applyMVC*). A continuación de dicha especificación, se define la misma operación como operación orientada a objetivo, por lo que se define explícitamente un parámetro "g" que al momento de la ejecución tomará el valor del objetivo que se desea lograr con dicha ejecución. El valor del argumento objetivo será una versión de objeto del tipo adecuado (según su definición) que pertenece a la versión de modelo actual, sobre la cual se está aplicando la operación. Por tal motivo, es necesario que aquellos elementos que serán considerados como objetivos a cumplir, ya hayan sido generados en transformaciones previas empleando las operaciones de diseño adecuadas (por ejemplo, mediante una operación *addQualityRequirement*, *addQualityScenario*, etc.). En cuanto a la especificación de la operación *applyMVC-go* puede observarse, que si bien es una operación compleja, es expresada en términos de las operaciones *applyMVC* y *assignPossibleScenarioSolution*, lo cual simplifica su definición. Sin embargo, la ejecución de la misma abarca la ejecución de numerosos comandos, lo cual no es advertido por el diseñador que ejecuta la operación, lo que refleja la transparencia de la captura de operaciones realizada por el modelo, y la significativa disminución de la carga de trabajo en la documentación de las trasformaciones que tienen lugar durante el proceso de diseño. En cuanto a la operación *assignPossibleScenarioSolution*, éste consiste en el establecimiento de las asociaciones entre los resultados de la operación *applyMVC* (variable *lres*) y una versión de tipo *PossibleScenarioSolution* que es agregada, y entre ésta y un escenario de calidad que está vinculado al requerimiento de calidad que es pasado como objetivo de la operación (*g*).

```
applyMVC(c: Component)
  s := get(System, c)
  view := addComponent(s, 'View', ['V-P1', 'V-P2'], [])
  controller := addComponent (s, 'Controller', ['C-P1', 'C-P2'], [])
  model := addComponent (s, 'Model', ['M-P1', 'M-P2'], [])
  modVista := addConnector(s, 'ConnModView',
['MV-R1', 'MV-R2'], get(Port, model(0)) • get(Port, view(0)), [])
  viewCtrlr := addConnector(s, 'ConViewCtrlr', ['VC-R1', 'VC-R2'],
                get(Port, view(0)) • get(Port, controller(0)), [])
  modCtrlr := addConnector(s, 'ConModCtrlr', ['MC-R1', 'MC-R2'],
                get(Port, model(0)) • get(Port, controller(0)), [])
  delegateResponsibility(c, model(0))
  delegateResponsibility (c, view(0))
  delegateResponsibility (c, controller(0))
```

```
(cont.)
// Connections are reattached with new components
  lpc := get(Port, c) // ports of the refined component
  lps := get(Port, null) // all ports in the system
  for each p in lpc
     np := select(lps)          // ask the user for the new port
     r := get(Rol, p)           // get the role of the former port
     addAssociation(np, r, RConnection)
  end for
  deleteComponent(c) // the original component is deleted
end
```

```
applyMVC-go(c: Component, g: QualityRequirement)
  lres := applyMVC(c)
  ls := select(get(Scenario, g))
  for each s in ls
    assignPossibleScenarioSolution (s, null, lres)
  end for
end
```

```
relaxedMVC-go(c: Connector, g: Constraint)
  s := get(System, c)
  deleteConnector(c)
  considersConstraint(null, g, [s], [])
end
```

```
assignPossibleScenarioSolution(npss: String, sce: QualityScenario, versionsList: [ArchitecturalElement])
  pss := add(npss, RPossibleScenarioSolution)
  addAssociation(sce, pss, RPSSQS)
end
```

Fig. 4**.** Ejemplos de especificación de operaciones orientadas a objetivos

La Fig. 4 también presenta la operación *relaxedMVC-go* cuyo objetivo es cumplir con una restricción impuesta por el entorno en donde se despliegan los componentes del patrón MVC, como puede ser un ambiente que no mantiene estados entre el cliente y el servidor ("*stateless*") como la web. En ese caso, el patrón MVC se relaja eliminando la comunicación que existe entre las vistas y el modelo, y se representa el hecho de que el arquitecto ha considerado esta restricción en su diseño.

En la Fig. 5 se presenta otro grupo de operaciones orientadas a objetivos que codifican algunos patrones arquitectónicos que son específicos arquitecturas en un contexto de *cloud* [21]. Estas operaciones son definidas como orientadas a objetivos, ya que incluyen un argumento extra *goal* para indicar en el momento de su ejecución cuál es el objetivo que busca satisfacer, o que requerimiento es el que conduce la decisión de diseño materializada en dicha ejecución. Por ejemplo, para la operación *replicateDataBase* se define que su aplicación tendrá como objetivo la satisfacción de algún requerimiento de performance que ha sido elicitado para la arquitectura, indicado por el tipo *PerformanceQualityRequirement* del parámetro *goal*, que es un tipo de objeto de diseño definido como subtipo de *QualityRequirement* en el dominio. Algo similar puede definirse para la operación *addWrapper*, que codifica la aplicación de un patrón *wrapper* que posibilita la *modificabilidad* del sistema.

5. Escenario de aplicación

Para ilustrar cómo se emplea el modelo, se presenta un escenario de aplicación que ilustra un proceso de migración de una arquitectura de software de un sistema tradicional a una cloud híbrida. En este caso, el proceso de diseño/evolución parte de la versión de modelo que representa a la arquitectura actual, y a través de la aplicación de operaciones, se va avanzando mientras se capturan las decisiones tomadas, los productos generados/modificados, y los principales objetivos perseguidos por estas decisiones. Algunas operaciones empleadas en este escenario han sido especificadas en un trabajo previo [23], y otras han sido redefinidas como *OpOO* (Fig. 5).

SportsInc es una empresa dedicada a la venta mayorista de ropa deportiva que en los últimos años ha iniciado también su propio web store para venta minorista. Este nuevo modelo de negocio ha sido exitoso y sus proyecciones de ventas online van en ascenso, por lo que han surgido nuevos requerimientos de escalabilidad y disponibilidad para el sistema.

<table>
<tr>
<td>replicateDataBase(s: System, c: Database, rn: String, h: Component, schemaScripts: String, goal: [PerformanceQualityRequirement, AvailabilityQualityRequirement])
dbr := addDatabase(rn, schemaScripts)
setFixedAllocation(dbr, h)
sgoal := select(goal)
ls := select(get(Scenario, sgoal))
for each s in ls
assignPossibleScenarioSolution (s, null, dbr)
end for
end</td>
<td>addWrapper(n: Network, wn: String, lreps: [Responsibility], lports: [Ports], c1: Component, c2: Component, goal: ModifiabilityQualityAttribute)
w := addNewComponent(n, wn, lresps, lports, [])
pw1 := select(get(Port, w)) pc1 := select(get(Port, c1))
res1 := addNewConnector(n, 'Conn1', ['Role11', 'Role12'], [pw1, pc1])
pw2 := select(get(Port, w)) pc2 := select(get(Port, c2))
res2 := addNewConnector(n, 'Conn2', ['Role21', 'Role22'], [pw2, pc2])
ls := select(get(Scenario, goal))
for each s in ls
assignPossibleScenarioSolution (s, null, w • res1 • res2)
end for
end</td>
</tr>
<tr>
<td colspan="2">synchronizeDataBases(s: System, nsync: String, lresps: Collection[Responsibility] , lports: Collection[Ports], mdb: Database, rdb: Database, gatewayComponent: Component, goal: [AvailabilityQualityAttribute, PerformanceQualityAttribute])
(No se muestra la especificación por restricciones de espacio)</td>
</tr>
</table>

Fig. 5**.** Operaciones orientadas a objetivos que definen patrones para la cloud

La empresa desea bajar los costos de infraestructura actuales por lo que se ha decidido migrar parte del sistema de ventas minoristas a la cloud. La empresa además posee un sistema legacy para la gestión de pedidos y ventas mayoristas, que actualmente es operado por los empleados administrativos de la empresa. En la actualidad, los pedidos de los preventistas son comunicados via fax o telefónica, y cargados al sistema por operadores. Se desea liberar a éstos de esta carga de trabajo, por lo que se ha pensado incorporar un nivel de acceso especial para preventistas en el sitio web, para la carga de sus pedidos. La arquitectura deberá satisfacer requerimientos de performance (disminuir la latencia en las respuestas), modificabilidad (posibilitar el reuso de componentes existentes en la nueva arquitectura), y seguridad. Se parte de la arquitectura de software inicial (a la que se denomina mv_0), dada por el sitio de e-commerce actual y los componentes legacy para gestión de ventas mayorista. El arquitecto lleva a cabo el proceso de diseño o evolución de la arquitectura aplicando secuencias de operaciones del dominio sobre las versiones de modelo que se van obteniendo. En primer lugar se identifican los requerimientos de calidad y funcionales que se desean alcanzar en el proceso de evolución a llevar a cabo:

θ_1={*addNewQualityRequirement(SystemSportEv1, 'Scalability')*, a*ddNewQualityRequirement(SystemSportEv1, 'Performance')*, *addNewQualityRequirement(SystemSportEv1, 'Availability')*, *addNewQualityRequirement(SystemSportEv1, 'Modifiability))*, *addNewQualityRequirement(SystemSportEv1, 'Security')*}) .

A partir de estos requerimientos, se toman las primeras decisiones de evolución de la arquitectura de software tendientes a alcanzarlos. Aunque no se muestra en el caso de estudio, es útil definir escenarios de calidad para cada uno de los requerimientos, lo que permitirá especificarlos en forma más concreta. La primera decisión es contratar servicios a un proveedor de servicios de cloud computing (*CSP*), en la modalidad IaaS, y luego migrar los componentes que constituyen la aplicación del web store minorista. Para ello el *CSP* provee una máquina virtual para servidor web sobre la cual se desplegará la capa de presentación del web store, por lo cual será necesario adecuar algunas interfaces a este nuevo entorno. Para la capa de lógica de negocios, y la capa de acceso a datos se procede de la misma manera, reubicándolas en un componente máquina virtual que se encuentra en el servidor de aplicación. También se crea un servidor virtual para la base de datos que se encontrará en la cloud, a fin de evitar la disminución en el rendimiento percibido por los clientes del webstore. Estas decisiones se materializan mediante la ejecución de *OpOOs*, donde el objetivo perseguido (indicado por el valor del último argumento) es la *performance* (representado en la versión de modelo actual por el objeto *Performance* $_{v1}$). Posteriormente mantener la conexión con la red privada virtual de la empresa con el *CSP*, se contrata un servicio VPN Gateway que permite la conexión segura a la infraestructura on-premises.

θ_2={*addCloudProviderNetwork(SystemSport*$_{v1}$*, CSPNetwork', Performance* $_{v1}$*)*}
θ_3={*addVirtualMachine(CSPNetwork*$_{v1}$*, 'VM-WebServer', Performance* $_{v1}$*), addVirtualMachine(CSPNetwork*$_{v1}$*, 'VM-AppServer', Performance* $_{v1}$*) , addVirtualMachine(CSPNetwork*$_{v1}$*, 'VM-DBServer', Performance* $_{v1}$*)*}
θ_4={*reallocateComponent(Presentation*$_{v1}$*, VM-WebServer*$_{v1}$*, Performance* $_{v1}$*)*, *reallocateComponent(BusinessLogic*$_{v1}$*, VM-AppServer*$_{v1}$*, Performance* $_{v1}$*)*, *reallocateComponent(DataAccess*$_{v1}$*, VM-AppServer*$_{v1}$*, Performance* $_{v1}$*)*}
θ_5={*addNewComponent(CSPNetwork*$_{v1}$*, 'VirtualNetworkGateway', [Resp-ConnectVPN],[PortVPNG1, PortVNPG2, PortVPNG3, PortVPNG4, PortVNG5], Security*$_{v1}$*)* }

Se procede a incorporar todas las nuevas responsabilidades arquitectónicas que se deben incorporar al componente de presentación del sistema web, ya que se incorpora el acceso

para preventistas para la carga de pedidos mayoristas.

θ_6={*addNewResponsibility(Presentation$_{v2}$, 'Resp-OrderFormPresentation'), addNewResponsibility(Presentation$_{v2}$, 'Resp-Autentication')*, *addNewResponsibility(Presentation$_{v2}$, 'Resp-OrderFormDefaultValues')*, *addNewResponsibility(Presentation$_{v2}$, 'Resp-OrderFormValidation')*, *addNewResponsibility(Presentation$_{v2}$, 'Resp-Submit')*}

Se decide un incorporar un *wrapper* para encapsular los pedidos y proveer el acceso al módulo legacy de Pedidos, ya que los pedidos mayoristas se continuarán cargando en la base de datos maestra de *SportInc*. Las solicitudes enviadas por el wrapper se encaminarán a través del VPN gateway. El objetivo de esta decisión es reusar un componente existente, y por lo tanto el requerimiento de modificabilidad es indicado como objetivo perseguido por la decisión.

θ_7={*addWrapper(CSPNetwork$_{v1}$, 'Orders-Wrapper', [Resp-ReceiveRequest, Resp-ForwardRequest],[PortOW1, PortOW2]*, *Presentation$_{v2}$*, *VPNGateway$_{v1}$*, *Modifiability $_{v1}$))*}

Para mejorar el rendimiento del sistema es deseable que el componente de acceso a datos del *webstore* minorista tenga un acceso rápido (sin latencia) al catálogo de productos a la venta, junto con los precios y promociones activas. Para ello se requiere replicar el esquema de la base de datos que corresponde a dicha partición, el cual se despliega sobre el componente virtual *VM-DBServer* que ya ha sido creado. Además para mantener esta base de datos actualizada en la *cloud*, se crea un servicio de sincronización que funciona en un servidor de la empresa, y que se encamina a través de la VPN con el gateway.

θ_8={*replicateDataBase(DBSports$_{v1}$, 'DBSports-Catalogue', 'Products catalogue and marketing schemes', VM-DBServer$_{v1}$, Performance$_{v1}$*}
θ_9={*addNewConnector(CSPNetwork$_{v1}$, 'DataAccess-DBCatalogue', ['Role31', 'Role32'],[PortDA2$_{v2}$, PortDBC1$_{v1}$]) }*
θ_{10}={*syncronizeDatabases(SystemSportsEv$_{v1}$, 'DBSynchronizer', [Resp-FindChanges, Resp-UpdateSchema],[PortSync1, PortSync2], DBSportsv1, DBSports-Cataloguev1, VPNGatewayv1, Performance$_{v1}$)*}

La ejecución de esta última operación también es bastante compleja, abarcando la creación y el establecimiento de tres conectores entre la base de datos maestra y el sincronizador, entre el sincronizador y el gateway, y entre el gateway y la base de datos reducida en la cloud. El escenario finaliza con la incorporación de los conectores necesarios entre el componente de acceso a datos del webstore, y el servidor de la base de datos maestra. La arquitectura de software resultante (Model version 10, en Fig. 6) puede continuar evolucionando creándose nuevas instancias de servidores web y de aplicación según demanda, e incorporando componentes dedicados para el balanceo de carga. El modelo permitió capturar las operaciones que dieron origen cada elemento, los valores de los argumentos con que estas operaciones se ejecutaron, y cuáles fueron las intenciones u objetivos que se intentaban alcanzar al momento de ejecutarlas. En una etapa posterior, la arquitectura alcanzada debería ser analizada empleando herramientas o métodos adecuados para verificar en qué grado la arquitectura satisface los objetivos. A partir del proceso de evolución representado mediante la captura de las operaciones ejecutadas y los productos generados, es posible responder a las diferentes preguntas que se plantearon como disparadoras del trabajo de investigación.

Por ejemplo: *¿qué objetivo se espera alcanzar con la aplicación de los patrones de replicación de base de datos y sincronización? Se desea cumplir con los requerimientos relativos a performance y disminución de latencia.* Tal información se deriva de las operaciones capturadas *replicateDataBase* (en θ_8) *y syncronizeDatabases* (*en* θ_{10}) cuyo argumento goal es *Performance$_{v1}$*.

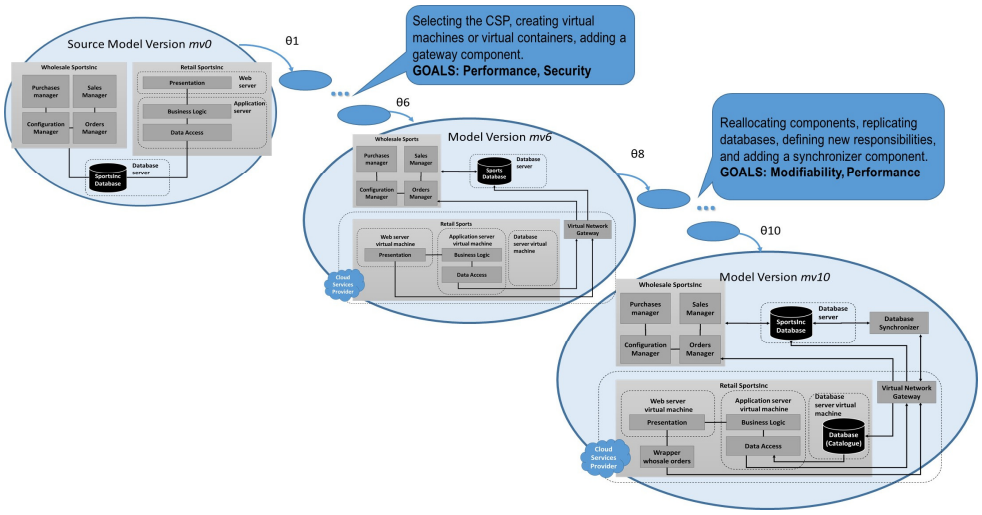

Fig. 6. Proceso de diseño de la arquitectura de SportsInc.

6. Conclusiones

El modelo propuesto favorece y facilita la documentación del SADP, conservando la completa evolución a partir de una arquitectura inicial. Dicha documentación favorece la compresión de los cambios realizados, de sus impactos en la arquitectura original. Permite sistematizar la aplicación de cambios en la arquitectura, ya que éstos se traducen en la aplicación de patrones de que han probado ser soluciones efectivas para problemas recurrentes. Las principales ventajas del modelo de operaciones orientado a objetivos propuesto son: i) posibilita la *definición* de *OpOOs*, de forma que un experto puede indicar cuál o cuáles son los posibles tipos de objetivos que podrían alcanzarse con la aplicación de la operación; ii) posibilita la ejecución y captura de *OpOOs*, lo que permite explicitar qué se desea alcanzar con esa decisión, generando y representando conocimiento que de otra manera se perdería. La propuesta no está limitada a un lenguaje de descripción de arquitecturas particular, ni a un conjunto de vistas específicas, ni está atada a un método de diseño dado. El modelo de dominio es lo suficiente flexible para especificar todos los tipos de objetos de diseño y operaciones que sean necesarias para el dominio del sistema. Como trabajo a futuro se continuará trabajando en la herramienta Trace(aaS) para incorporar la posibilidad de definir operaciones orientadas a objetivos que posibiliten una cierta asistencia al diseñador, sugiriendo las posibles operaciones a ejecutar, dependiendo del contexto de la operación, es decir, de qué requerimientos, escenarios, restricciones, etc. se encuentran presentes en la versión de modelo actual.

Agradecimientos

Este trabajo ha sido financiado en forma conjunta por CONICET, la ANPCyT, y la UTN. Se agradece el apoyo brindado por estas instituciones.

References

1. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, Second Edition. Addison-Wesley, Boston, MA (2003)
2. Tyree, J., Akerman, A.: Architecture Decisions: Demystifying Architecture, IEEE Software

22(2), 19-27 (2005)
3. Jansen, A., Bosch, J.: Software Architecture as a Set of Architectural Design Decisions. In: Proc. of the 5th IEEE/IFIP WICSA, pp. 109-120 (2005)
4. Babar, M. A., Gorton, I.: A Tool for Managing Software Architecture Knowledge. In: Proceedings of the Second Workshop on SHARK-ADI '07, ICSE. IEEE Computer Society, Washington, DC, 11(2007)
5. van Heesch, U., Avgeriou, P., Hilliard, R.: A documentation framework for architecture decisions. Journal of Systems and Software, 85 (4), 795-820 (2012)
6. Kruchten, P., Lago, P., van Vliet, H.: Building up and reasoning about architectural knowledge. In: Proceedings of QoSA'06, Springer-Verlag, Berlin, Heidelberg, 43-58 (2006).
7. Tang, A., Ali Babar, M., Gorton, I., Han, J.: A survey of architecture design rationale. Journal of Systems and Software, 79 (12), 1792-1804 (2006)
8. Falessi, D., Cantone, G., Kruchten, P.: Value-Based Design Decision Rationale Documentation: Principles and Empirical Feasibility Study. In: Proc. of WICSA, 189-198 (2008).
9. Avgeriou, P., Kruchten, P., Lago, P., Grisham, P., Perry, D.: Architectural knowledge and rationale: issues, trends, challenges. SIGSOFT Soft. Eng. Notes, 32 (4), 41-46 (2007)
10. Roldán, M. L., Gonnet, S., Leone, H.: TracED: a tool for capturing and tracing engineering design processes. Advances in Engineering Software, 41 (9), 1087-1109 (2010)
11. Roldán, L., S. Gonnet and H. Leone, Knowledge representation of the software architecture design process based on situation calculus, *Expert Systems,* **30**(1) (2013), 34–53.
12. Gotel, O., Cleland-Huang, J., Hayes, J., Zisman, A., Egyed, A., Grünbacher, P., Dekhtyar, A., Antoniol, G., Maletic, J., Mäder, P., "Traceability Fundamentals", *Software and Systems Traceability*, J. Cleland-Huang, O. Gotel, A. Zisman (Eds.), pp. 3-22, Springer, 2012.
13. Winkler, S., von Pilgrim, J., "A survey of traceability in requirements engineering and model-driven development", *Software System Model*, 9, pp. 529-565, 2010.
14. Murta, L., van der Hoek, A., Werner, C., "ArchTrace: Policy-Based Support for Managing Evolving Architecture-to-Implementation Traceability Links", *21st IEEE/ACM International Conference on Automated Software Engineering (ASE '06)*, pp. 135-144, 2006.
15. Conklin, J., Begeman, M.L., "gIBIS: a hypertext tool for exploratory policy discussion", *ACM Transactions on Information Systems*, 6, 4, pp. 303-331, 1988.
16. Dutoit, A., McCall, R., Mistrik, I., Paech, B. (eds.), *Rationale Management in Software Engineering*, Springer, 2006.
17. Carnduff, T., Goonetillake, J., "Configuration management in evolutionary engineering design using versioning and integrity constraints", *Advances in Engineering Software*, 35, pp. 161-177, 2004.
18. Westfechtel, B., "Models and tools for managing development process", *Lecture notes in computer science*, 1646, 1998.
19. Roldán, M. L., "Un modelo para la representación de conocimiento y razonamiento en el proceso de diseño de arquitecturas de software", UTN, FRSF, 2009.
20. Hernandez, F., Roldán, M.L., Vegetti, M., Gonnet, S., Leone, H. TracED(aaS): Captura y Trazabilidad de Artefactos del Proceso de Diseño, 2° CoNaIISI, San Luis, (996-1007) 2014.
21. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, Paulo, Nord, R., Stafford, J.: Documenting Software Architectures: Views and Beyond, 2nd Ed. Addison-Wesley (2010)
22. IEEE: IEEE 1417, Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Press (2000)
23. Roldán, M.L., Gonnet, S., Leone, H., "Representación de la evolución y refactoring de arquitecturas de software mediante la aplicación y captura de operaciones arquitectónicas", 2° CoNaIISI, San Luis, (1019-1030) 2014.