Modelo de Soporte de Decisiones de Diseño Arquitectónicas

M. Celeste Carignano, Horacio Leone, y Silvio Gonnet

INGAR, Universidad Tecnológica Nacional, CONICET
Avellaneda 3657, 3000, Santa Fe, Argentina
{celestec, hleone, sgonnet}@santafe-conicet.gov.ar

Resumen. El diseño arquitectónico es la base de todo sistema de software en el cual se definen los lineamientos fundamentales para la construcción del mismo. La definición de la arquitectura es una orquestación de decisiones de diseño arquitectónico que permiten dar respuesta a los requerimientos no funcionales establecidos. Es fundamental capturar durante su especificación el conocimiento involucrado en la toma de decisiones, para así enriquecer los artefactos resultantes y facilitar su revisión, utilización y mantenimiento. El presente trabajo propone un modelo de soporte para documentar el conocimiento aplicado en la toma de decisiones durante el diseño de la arquitectura, con el objetivo de obtener como resultado no sólo los artefactos arquitectónicos sino también el conocimiento aplicado.

Palabras clave: Arquitectura de software, Diseño arquitectónico, Framework de razonamiento, Tácticas arquitectónicas, Atributos de calidad.

1 Introducción

El diseño arquitectónico es un proceso creativo que depende de muchos factores, como ser: los arquitectos involucrados, el contexto del sistema, las restricciones impuestas (tanto por el cliente y sus necesidades, los participantes, entidades internas o externas, etc). Estos factores influyen en las decisiones arquitectónicas tomadas durante la generación de los artefactos, decisiones que impactarán a lo largo del proceso de desarrollo y sobre el sistema resultante. Adicionalmente, es inviable la obtención de una arquitectura que se describa en su totalidad por si misma y que sea comprendida por todo aquel que intente usarla, ya sea para mantenimiento o para consultarla como referencia en generación de nuevos sistemas.

La etapa de mantenimiento de software es la más extensa en el tiempo dentro del ciclo de vida de un proyecto de desarrollo de software. El mantenimiento puede involucrar cambios correctivos o evolutivos, pudiendo impactar en la arquitectura del sistema. En consecuencia, si la arquitectura no se encuentra correctamente documentada es muy difícil su mantenimiento. Una arquitectura debidamente documentada facilita la comprensión de las decisiones que condujeron a su generación. De esta manera en caso de necesitar revisiones o modificaciones se pueden conocer las suposiciones y criterios aplicados en la selección y generación de los artefactos.

XXXIV Conferencia Latinoamericana de Informática (CLEI 2008), pp. 390-399, 2008.

Actualmente existen herramientas que permiten documentar los artefactos de una arquitectura, definiendo vistas, estilos, elementos y relaciones (Acme [1], Wright [2], UniCon [3] entre otros). Sin embargo, tales herramientas no permiten documentar las decisiones involucradas de manera integrada a los artefactos resultantes.

También existen trabajos que se enfocan en las decisiones de diseño arquitectónico desde un punto de vista de análisis ([4], [5], [6], [7] entre otros) pero no obtienen un modelo que pueda ser aplicado en el diseño arquitectónico de distintos sistemas que permita mantener todo el conocimiento involucrado de manera que sea útil al trabajar con una arquitectura definida.

Este trabajo expone un modelo orientado a objetos para el soporte de decisiones arquitectónicas, utilizando: (i) UML [8] como lenguaje de modelado de los conceptos involucrados, y (ii) OCL [9] como lenguaje de restricciones para definir las reglas que las instancias del modelo deberán cumplir para ser válidas. El objetivo de este modelo es proveer un soporte para capturar el conocimiento aplicado en la toma de decisiones al momento de definir una arquitectura. Para ello, en la sección 2 se describe el modelo conceptual definido, identificando conceptos y reglas que los condicionan. En la sección 3 se presenta un caso de estudio, aplicando el modelo propuesto a la especificación arquitectónica de un sistema de “Home Banking” básico. Finalmente, en la sección 4 se exponen las conclusiones del presente trabajo y los pasos a seguir.

2 Modelo Conceptual

Todo sistema de software se construye para dar respuesta a las necesidades de los “stakeholders”, cumpliendo determinadas características. Las funcionalidades requeridas son especificadas utilizando requerimientos funcionales (*FunctionalRequirement* en Fig. 1). Aquellas características que están asociadas a propiedades que debe poseer el sistema son especificadas utilizando requerimientos no funcionales (*NonFunctionalRequirement* en Fig. 1). Todo requerimiento se encuentra respaldado por fuentes de información (*SourceInformation* en Fig. 1) que contienen las especificaciones informales de los “stakeholders”, productos de técnicas de relevamiento. Los requerimientos funcionales son descriptos en casos de uso (*UseCase* en Fig. 1). Cada caso de uso es documentado en especificaciones (*UseCaseSpecification* en Fig. 1) que detallan la funcionalidad representada.

En la Fig. 1 se modela la relación entre los distintos tipos de requerimientos y los elementos que los describen. El modelo parte de la base de que la arquitectura de un sistema se construye para dar solución a los requerimientos no funcionales del mismo. Estos requerimientos pueden tener como ámbito de aplicación el sistema completo (*SystemScope* en Fig. 1) o un caso de uso particular (*FunctionalityScope* en Fig. 1). Un requerimiento no funcional cuyo ámbito es una funcionalidad debe estar asociado al caso de uso que la describe, de manera de poder realizar una trazabilidad entre los casos de uso y los requerimientos no funcionales que los condicionan.

Los requerimientos no funcionales son descritos a través de escenarios. Bass y colab. [10] caracterizan a un escenario mediante: (i) un estímulo (*Stimulus* en Fig. 1), condición que necesita ser considerada cuando arriba al sistema; (ii) una respuesta (*Response* en Fig.1), actividad realizada luego del arribo del estímulo; (iii) una fuente

de estímulo (*StimulusSource* en Fig. 1), entidad que genera el estímulo; (iv) un entorno (*Environment* en Fig. 1), condiciones bajo las cuales el estímulo ocurre; (v) un artefacto estimulado (*StimulatedArtefact* en Fig. 1); y (vi) una medida de respuesta (*ResponseMeasure* en Fig. 1), restricción específica que debe ser satisfecha por la respuesta. Un escenario representa un requerimiento específico de un atributo de calidad (*QualityAttribute* en Fig. 1).

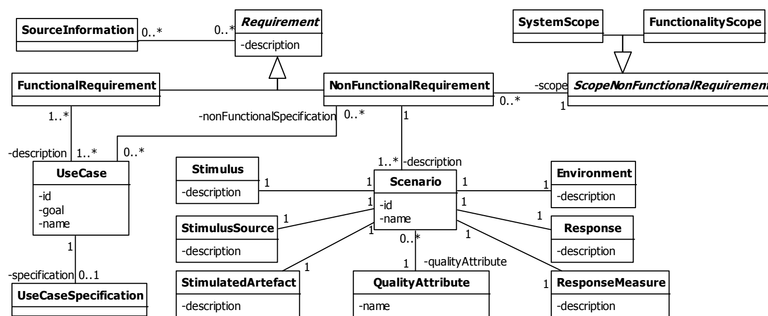

Fig. 1. Relación entre requerimientos, casos de uso y escenarios.

Existen diversos modelos de atributos de calidad que difieren en la calificación y definición de éstos, como ser: el de McCall [11], Boehm [12], e ISO 9126-1 [13]. La Fig. 2 presenta los distintos tipos de atributos de calidad según [13]: i) interno (*InternalQualityAttributeType*); ii) externo (*ExternalQualityAttributeType*); iii) interno y externo (*InternalExternalQualityAttributeType*); o iv) de uso (*OfUseQualityAttributeType*).

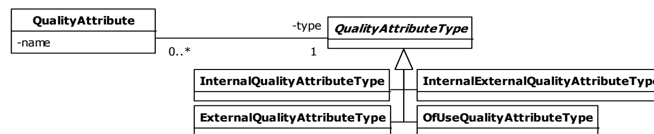

Fig. 2. Tipos de atributos de calidad.

Un atributo de calidad interno (externo) es una característica de un producto de software desde una perspectiva interna (externa). Un atributo de calidad de uso es una característica de un producto de software desde la perspectiva del usuario cuando lo utiliza en un ambiente y contexto específico de uso.

Todo atributo de calidad tiene asociado "frameworks" de razonamiento (*ReasoningFramework* en Fig. 3). Un "framework" de razonamiento provee la capacidad de razonar sobre el comportamiento de un atributo de calidad específico de una arquitectura a través del uso de teorías analíticas [6]. La Fig. 3 presenta los conceptos que participan en la definición de un "framework" de razonamiento. Los "frameworks" de razonamiento tienen un conjunto de parámetros (*Parameter* en Fig. 3) asociados. Éstos pueden ser independientes (*IndependentParameterType* en Fig. 3) o dependientes (*DependentParameterType* en Fig. 3). Los parámetros dependientes son medidas de atributos de calidad calculados usando el "framework" de

razonamiento. Los parámetros independientes tienen asociados un conjunto de tácticas arquitectónicas compatibles que permiten controlarlos.

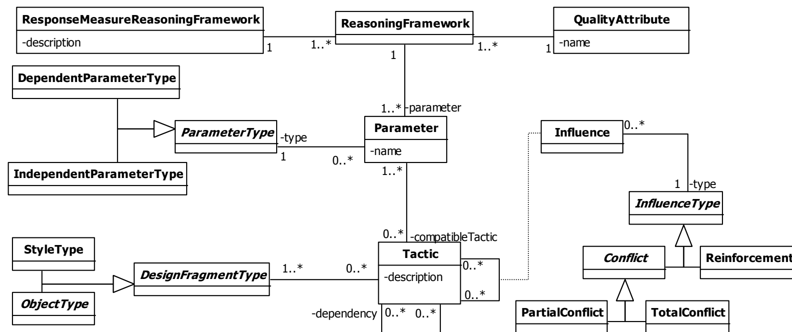

Fig. 3. Framework de razonamiento.

Las tácticas arquitectónicas (*Tactic* en Fig. 3) son una manera de satisfacer una medida de respuesta de un atributo de calidad manipulando algunos aspectos del modelo de atributo de calidad a través de decisiones de diseño arquitectónicas [5].

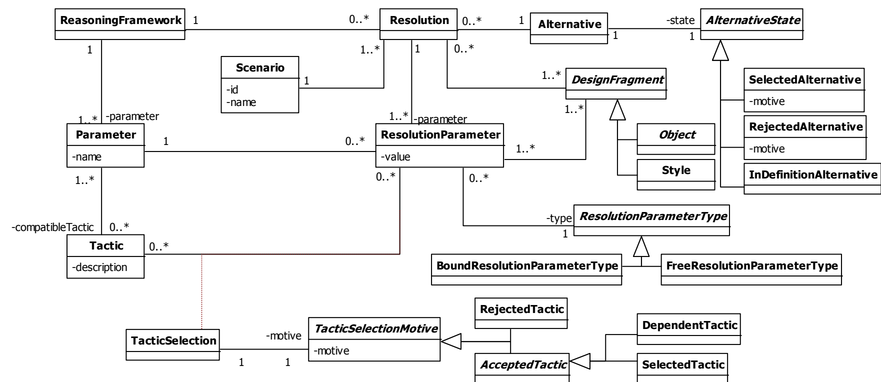

Fig. 4. Alternativas y Resoluciones de Escenarios.

Cuando se define una arquitectura es posible considerar distintas alternativas (*Alternative* en Fig. 4) las cuales pueden ser finalmente aceptadas o rechazadas. Cada una de dichas alternativas está compuesta por un conjunto de resoluciones de escenarios (*Resolution* en Fig. 4) que reúnen las asignaciones a los parámetros del "framework" de razonamiento seleccionado para un escenario dado en dicha alternativa. La Fig. 4 presenta el modelado de alternativas y resoluciones.

Si la resolución del parámetro es de tipo libre, se le debe asignar por lo menos una táctica que permita controlarlo (Restricción 1). Dicha táctica debe pertenecer al grupo de tácticas compatibles del parámetro asociado a la resolución de parámetro

(Restricción 2). Si la resolución del parámetro es de tipo ligada no debe tener tácticas asociadas (Restricción 3).

```
context ResolutionParameter
inv resolutionParameterFree:
      self.type.oclIsKindOf(BoundResolutionParameterType) or          (1)
      (self.type.oclIsKindOf (FreeResolutionParameterType) and
      self.tacticSelection -> select(i |
      i.motive.oclIsKindOf(SelectedTactic)) -> notEmpty())

context ResolutionParameter
inv resolutionParameterTactic:
      self.type.oclIsKindOf(BoundResolutionParameterType) or          (2)
      (self.type.oclIsKindOf (FreeResolutionParameterType) and
      self.tactic -> forAll(st |
      self.parameter.compatibleTactic -> exists (t | t = st)))

context ResolutionParameter
inv resolutionParameterBound:
      self.type.oclIsKindOf(FreeResolutionParameterType) or           (3)
      (self.type.oclIsKindOf(BoundResolutionParameterType) and
      self.tacticSelection -> isEmpty())
```

La aplicación de una táctica puede requerir que otras tácticas también sean aplicadas, lo que genera una relación de dependencia entre ellas (*dependency* en Fig. 3) (Restricción 4). Adicionalmente, la aplicación de una táctica puede influir sobre el logro de otras. Esta influencia puede: (i) ser positiva (*Reinforcement* en Fig. 3), la aplicación de una táctica favorece el logro de otra; (ii) presentar un conflicto parcial (*PartialConflict* en Fig. 3), la aplicación de una táctica puede influir en el logro de otra táctica de manera negativa pero no impide su aplicación; o (iii) representar un conflicto total (*TotalConflict* en Fig. 3), la aplicación de la táctica inhabilita el empleo de otras. En una misma alternativa no se pueden aplicar dos tácticas cuyo conflicto es total (Restricción 5).

```
context TacticSelection
inv dependency: self.motive.oclIsKindOf(RejectedTactic) or
      self.motive.oclIsKindOf(DependentTactic) or
      (self.motive.oclIsKindOf(SelectedTactic) and
      ((self.tactic.dependency -> isEmpty()) or
      (self.tactic.dependency -> forAll(t: Tactic |                   (4)
      t.tacticSelection -> exists(ts|
      ts.motive.oclIsKindOf(AcceptedTactic) and
      ts.resolutionParameter.resolution.alternativeResolution =
self.resolutionParameter.resolution. alternativeResolution)))))

context Influence
inv incompatible: not (self.type.oclIsKindOf(TotalConflict)) or
     (self.type.oclIsKindOf(TotalConflict) and
     not (self.tacticOrigin.tacticSelection -> exists (tsO |
     tsO.motive.oclIsKindOf(AcceptedTactic)  and                      (5)
     self.tacticInfluenced.tacticSelection ->exists (tsTI |
     tsTI.motive.oclIsKindOf(AcceptedTactic) and
     tsO.resolutionParameter.resolution.alternativeResolution =
tsTI.resolutionParameter.resolution.alternativeResolution))))
```

El motivo de la elección de cada táctica debe ser justificado (*SelectedTactic* en Fig. 4). Si la táctica seleccionada depende de otras tácticas, éstas deben seleccionarse en la resolución del parámetro, indicando el motivo de la selección (*DependentTactic* en Fig. 4). En caso de que una táctica no sea seleccionada, opcionalmente se puede especificar el motivo del rechazo de la misma (*RejectedTactic* en Fig. 4).

La aplicación de cada táctica genera fragmentos de diseño (*DesignFragment* en Fig. 4), los cuales pueden ser estilos u objetos arquitectónicos. Cada uno de estos fragmentos de diseño tienen un tipo (*DesignFragmentType* en Fig.3), y cada tipo se encuentra asociado a una o varias tácticas, de manera de que los fragmentos de diseño generados deberán ser de los tipos válidos asociados a la táctica seleccionada (Restricción 6). La utilización de los tipos de fragmentos en la especificación de artefactos de diseño está descripta en [14].

```
context TacticSelection
inv designFragment: self.motive.oclIsKindOf(RejectedTactic) or
     (self.motive.oclIsKindOf(AcceptedTactic) and
     self.resolutionParameter.designFragment -> notEmpty() and      (6)
     self.resolutionParameter.designFragment -> forAll (df |
self.tactic.designFragmentType -> exists(dft| dft = df.type)))
```

Una resolución debe tener tantas resoluciones de parámetro como parámetros independientes tiene el "framework" de razonamiento seleccionado. Un parámetro dependiente no debe tener resoluciones de parámetro asociadas.

3 Caso de Estudio

Para la aplicación del modelo propuesto se utilizó la herramienta USE (UML Specification Environment), la cual permite validar y verificar especificaciones consistentes de UML junto con invariantes, pre- y post-condiciones de OCL [15].

El modelo se aplicó a un caso de estudio en el que se especificaron los requerimientos funcionales de un sistema de "Home Banking". Se relevaron dos requerimientos no funcionales relacionados con el tiempo de ejecución de transferencias (*NFR_TB*) y el esfuerzo de modificación de la interfaz del usuario (*NFR_CH*).

La validación se realizó ingresando como caso de prueba todos los objetos generados a la herramienta USE, verificando que se cumplan las invariantes definidas (Restricciones 1-10). Se describe a continuación cómo se construyó una alternativa de arquitectura para dar respuesta al requerimiento no funcional *NRF_CH* asociado al atributo de calidad *Changeability*.

Como primera medida se generó una estructura base sobre la cuál trabajar, conformada por los objetos del modelo que son independientes de cualquier dominio de problema. Estos objetos permiten utilizar el modelo propuesto para documentar distintos diseños arquitectónicos, sin necesidad de volver a definirlos o modificarlos. Para la generación de tal estructura básica se definieron los atributos de calidad, utilizando los propuestos por ISO 9126-1 [13]. Luego se crearon los objetos correspondientes al "framework" de razonamiento asociado al atributo de calidad *Changeability* propuesto por [5]. En el centro de la Fig. 5 se presenta el "framework" de razonamiento seleccionado (*modifiability*). Para dicho "framework" se explicita el atributo de calidad al cual está asociado (*changeability*), y sus parámetros (instancias de *Parameter*), todos de tipo independientes (*independenParameterType*).

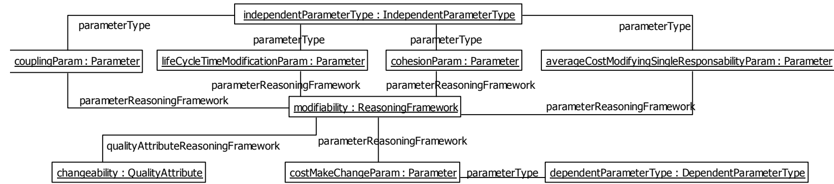

Fig. 5. “Framework” de razonamiento aplicado a caso de estudio.

Una vez definido el “framework” de razonamiento a aplicar se identificaron las tácticas disponibles para controlar los parámetros independientes del mismo utilizando las propuestas en [5]. La Fig. 6 presenta las tácticas seleccionadas para los parámetros *cohesion*, *coupling*, *averageCostModifyingSingleResposability*, y *lifeCycleTimeModification*.

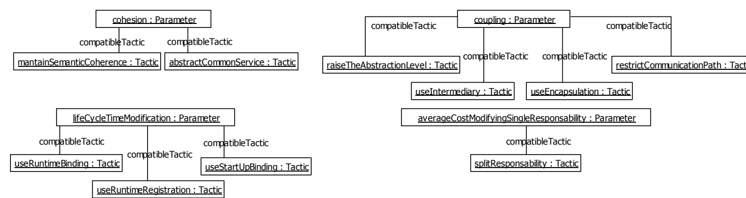

Fig. 6. Tácticas aplicadas a control de parámetros de “framework” de razonamiento.

Para cada táctica se definieron los tipos de fragmentos de diseño asociados en base a [7], definiéndose así los estilos “Pipe and Filter”, “Layered”, “Broker” y “Model-View-Controller”. La Fig. 7 presenta un ejemplo de asociación de tácticas a fragmentos de diseños para el estilo *Model-View-Controller*.

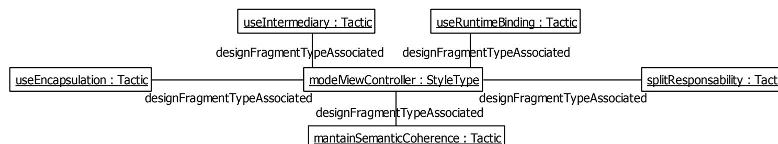

Fig. 7. Ejemplo de fragmento de diseño asociado a tácticas seleccionadas.

Los objetos generados hasta este punto son los que conforman la estructura base del modelo. Una vez definida dicha estructura se comenzó a trabajar sobre el caso de estudio. *NFR_TB* era un requerimiento no funcional de alcance *FunctionalityScope* (Fig. 8). En consecuencia, *NFR_TB* se asoció al caso de uso que describía la ejecución de transferencias (*transferRealize*, Fig. 8) y al escenario que detallaba el comportamiento del requerimiento no funcional (*transferPerformance*, Fig. 8). Dicho escenario tenía como atributo de calidad a *timeBehaviour* (Fig. 8). *NFR_CH* era un requerimiento no funcional de alcance *SystemScope* (Fig. 8) por lo cual se asoció únicamente al escenario que detallaba su comportamiento (*uiChangeability*, Fig.8). Dicho escenario tenía como atributo de calidad a *changeability*. La Fig. 8 presenta la aplicación del modelo a estas definiciones.

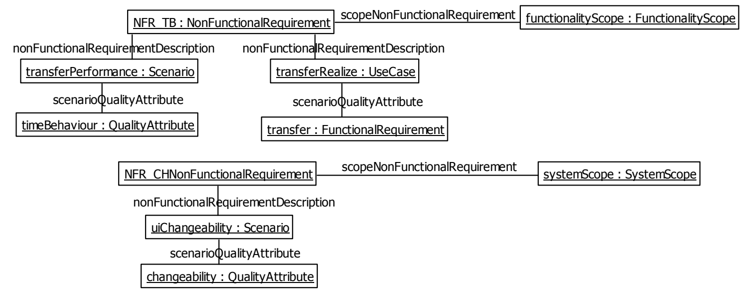

Fig. 8. Requerimientos del Caso de Estudio.

Se trabajó en la propuesta de una arquitectura (*alternativeProposal*) para dar respuesta al requerimiento *NFR_CH*. Dado que dicho requerimiento se encuentra vinculado al atributo de calidad *timeBehaviour* (Fig. 8), un “framework” de razonamiento posible a aplicar es el de *modifiability* (Fig. 5). En consecuencia, se definió una resolución (*resolutionModifiability*, Fig. 9) asociada a dicho “framework” (*modifiability*, Fig. 9).

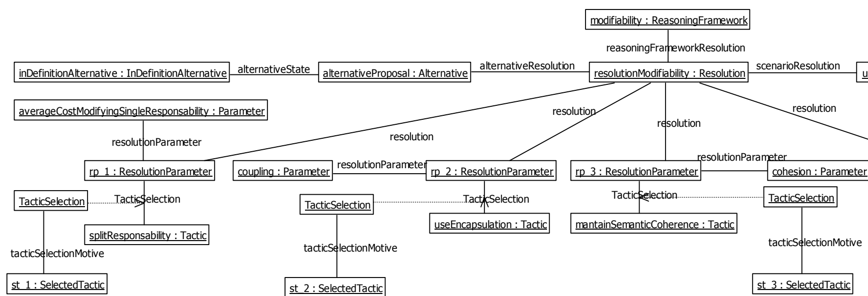

Fig. 9. Selección de tácticas a aplicar.

Luego, se generaron resoluciones de parámetro (*rp_1*, *rp_2*, *rp_3*, *rp_4* en Fig. 9), para cada uno de los parámetro independiente del “framework” de razonamiento (*averageCostModifyingSingleResposability*, *coupling*, *cohesión*, y *lifeCycleTimeModification*, respectivamente) y una táctica para poder controlarlo. Se utilizó la clase del modelo *TacticSelection* para documentar los motivos de selección. El motivo expuesto fue la necesidad de aislar la interfaz de usuario del resto de los componentes de la aplicación, de manera de que en caso de que ocurran cambios en la misma no impacten en los demás componentes. Siguiendo dicho criterio las tácticas seleccionadas fueron: (i) *splitResponsability* para el parámetro *averageCostModifiyngSingleResponsability*; (ii) *useEncapsulation* para el parámetro *coupling*; (iii) *maintainSemanticCoherence* para el parámetro *cohesion*; y (iv) *useRuntimeBinding* para el parámetro *lifeCycleTimeModification*. La Fig. 9 representa parcialmente dicha selección, sólo se ilustran los primeros tres parámetros.

Los fragmentos de diseño generados en la resolución *resolutionModifiability* fueron aquellos correspondientes a un estilo de tipo *Model-View-Controller*. La Fig. 10 representa los artefactos obtenidos en una vista de tipo “Component and Connector”, utilizando el modelo presentado en [14].

Fig. 10. Artefactos obtenidos para dar respuesta a atributo de calidad *changeability*.

De la misma forma se trabajó el requerimiento *NFR_TB*, asociado al tiempo de respuesta (*timeBehaviour*, Fig. 8), completando la alternativa desarrollada. De esta manera, siguiendo el modelo propuesto, se obtuvo la arquitectura necesaria para dar respuesta a los requerimientos no funcionales asociados al caso de estudio presentado. Como resultado se generaron los artefactos arquitectónicos junto a las decisiones arquitectónicas realizadas durante su diseño.

4 Conclusiones y Trabajos Futuros

Este trabajo propone un modelo de soporte para capturar el conocimiento aplicado en la toma de decisiones durante la confección de la arquitectura. El objetivo del mismo es permitir a los arquitectos plasmar el conocimiento y las decisiones tomadas al momento de definir una arquitectura, de manera que en las revisiones, uso y mantenimiento se disponga de la información necesaria.

El modelo planteado permite documentar las arquitecturas alternativas que se especifican durante el diseño arquitectónico, detallando las decisiones tomadas en cada una de ellas al momento de definir las tácticas a aplicar para dar respuesta a los requerimientos no funcionales del sistema. Las decisiones arquitectónicas pueden ser documentadas, de manera que la arquitectura resultante no se conforme únicamente de artefactos, que serán utilizados en etapas posteriores, sino también del conocimiento involucrado, que será utilizado al momento de revisar, modificar y mantener la arquitectura.

Los trabajos futuros refinarán el modelo propuesto con el objetivo de especificar de qué manera la consulta de la arquitectura generada con el modelo presentado puede ser útil en etapas posteriores.

Agradecimientos. Se agradece el apoyo financiero brindado por CONICET, la Agencia Nacional de Promoción Científica y Tecnológica (PICT 12628 y PICT 22118) y la Universidad Tecnológica Nacional.

Referencias

1. Garlan, D., Monroe, R., Wile, D.: Acme: An Architecture Description Interchange Language. CASCON'97, pp 169--183 (1997)
2. Allen, R.: A Formal Approach of Software Architecture. Technical Report, CMU-CS-97-144 (1997)
3. UniCon, http://www.cs.cmu.edu/afs/cs.cmu.edu/Web/People/UniCon
4. Bachmann, F., Bass, L., Klein, M.: Illuminating the Fundamental Contributors to Software Architecture Quality. Technical Report, CMU/SEI-2002-TR-025 (2002)
5. Bachmann, F., Bass, L., Klein, M.: Deriving Architectural Tactics: A Step Toward Methodical Architectural Design. Technical Report, CMU/SEI-2003-TR-004 (2003)
6. Bass, L., Ivers, J., Klein, M., Merson, P.: Reasoning Framework. Technical Report, CMU/SEI-2005-TR-007 (2005)
7. Bachman, F., Bass, L., Nord, R.: Modifiability Tactics- Technical Report, CMU/SEI-2007-TR-002 (2007)
8. Unified Modeling Language: Infrastructure v2.1.1, Object Management Group Specification (2007), http://www.omg.org
9. Object Constraint Language v2.0, Object Management Group Specification (2006), http://www.omg.org
10. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edition. Addison-Wesley (2003)
11. Boehm, B.: Qualitative evaluation of software quality. Proc. 2nd ICSE, pp 592--605 (1976)
12. McCall, J.: Factors in software quality. Rome Air Develop Centre Report, TR-77-369 (1977)
13. ISO/IEC 9126-1:2001 Software engineering - Product quality - Part 1: Quality model, www.iso.org
14. Carignano, M.C., Gonnet, S., Leone, H.: Ontología de Arquitectura de Software: un Modelo Conceptual de Artefactos para el Diseño Arquitectónico. 8th Argentinean Symposium on Software Engineering, pp. 46--60 (2007)
15. Ziemann, P., Gogolla, M.: Validating OCL Specifications with the USE Tool – An Example Based on the BART Case Study. Electronic Notes in Theoretical Computer Science 80, pp. 157--169 (2003)