Hacia una solución basada en frameworks para la definición de refactorizaciones con independencia del lenguaje

Carlos López Nozal[1], Raúl Marticorena Sánchez[1] y Yania Crespo González-Carvajal[2]

[1] Universidad de Burgos, Dpto. de Ingeniería Civil
Área de Lenguajes y Sistemas Informáticos
{clopezno, rmartico}@ubu.es
[2] Universidad de Valladolid, Dpto. de Informática
{yania}@infor.uva.es

Resumen En este trabajo se presenta el estudio de un conjunto de refactorizaciones desde el punto de vista de un lenguaje modelo. El objetivo es validar la factibilidad de llevar a cabo, sobre un framework que conceptualice las abstracciones del lenguaje modelo, las operaciones de refactorización definidas en base a dichas abstracciones. De esta manera se avanza hacia una solución al desarrollo de herramientas de refactorización con independencia del lenguaje.
El trabajo también presenta el estudio de un lenguaje (GJ, Generic Java) como instancia del lenguaje modelo, de manera que se avanza también en la validación del modelo desde el punto de vista de la factibilidad de instanciar el framework para diferentes lenguajes.

Palabras clave: refactorización, independencia del lenguaje, abstracción de lenguajes, frameworks, genericidad.

1. Introducción

El creciente auge de los trabajos realizados en el área de la refactorización de software está marcando una serie de tendencias entre las que se encuentra, como línea abierta de investigación, la búsqueda de una cierta independencia del lenguaje a la hora de definir, analizar y desarrollar herramientas que asistan al proceso de refactorización. En trabajos realizados previamente se ha propuesto el lenguaje modelo MOON[3] [1] para representar las construcciones abstractas necesarias en la definición y análisis de refactorizaciones.

Dicho lenguaje debe servir como base para una solución a las posibilidades de reutilización en el desarrollo de herramientas que ejecutan refactorizaciones cuando se adapten a nuevos lenguajes y nuevas operaciones de refactorización. Para lograr esto se necesita la validación del lenguaje modelo MOON y avanzar en su soporte para lo que se propone una solución basada en frameworks [2].

[3] MOON es acrónimo de Minimal Object-Oriented Notation.

El objetivo final es llegar a lograr la mayor independencia posible del lenguaje original en la realización de refactorizaciones permitiendo reducir los esfuerzos en su definición, al no tener que particularizar para cada lenguaje.

Existen trabajos previos en la definición de refactorizaciones independientes del lenguaje. En [3] [4] se presenta el modelo FAMIX como un meta-modelo para almacenar información con el objetivo de integrar varios entornos de desarrollo, entre ellos una herramienta para asistir a las refactorizaciones, denominado MOOSE [5]. Como punto de partida para la definición del modelo se centran en el estudio de dos lenguajes de características diferentes como Smalltalk y Java. Al intentar dar respuesta tanto a lenguajes estática como dinámicamente tipados, no toman en cuenta características complejas en los lenguajes fuertemente tipados, objeto de nuestro estudio, como herencia avanzada (herencia múltiple con modificadores en herencia de renombrado, redefinición, resolución de conflictos, etc. . .) y genericidad.

Propuestas similares han sido realizadas en la definición de modelos para la descripción de lenguajes orientados a objetos (LOO). A través de la definición del modelo OFL (Open Flexible Languages) [6], o bien con la propuesta de frameworks basados en la representación en forma de grafo del software, que soporten métricas con independencia del lenguaje [7]. En dichas propuestas no se aborda la definición de refactorizaciones sobre dichos modelos.

En la definición de refactorizaciones nos encontramos con trabajos que definen éstas en base a precondiciones y postcondiciones, sustentadas en predicados [8], centrando sus definiciones en un único lenguaje sin contemplar genericidad ni soporte para las mismas.

Hasta donde conocemos, estos han sido los únicos trabajos que se han publicado con un propósito similar, en mayor o menor medida, al nuestro: avanzar hacia el soporte de refactorizaciones lo más independiente posible del lenguaje.

En lo que sigue, el artículo se estructura de la siguiente forma: la Sección 2 analiza como un framework puede dar soporte tanto al lenguaje modelo como al conjunto de refactorizaciones definidas sobre él, la Sección 3 estudia un conjunto de refactorizaciones definidas sobre el lenguaje modelo soportado en el framework, la Sección 4 continúa con la validación del soporte sobre el framework de nuevos lenguajes, con características específicas, como es el caso de la extensión del lenguaje Java con genericidad (GJ o Generic Java), mientras que en la Sección 5 se concluye y se muestran las líneas de trabajo futuras.

2. Solución basada en Frameworks

En el establecimiento de las bases de una solución a las posibilidades de reutilización en el desarrollo de herramientas que ejecuten refactorizaciones independientemente del lenguaje, es necesario un soporte que aúne las abstracciones de los lenguajes de programación (clases, tipos, parámetros formales, . . .), y el conjunto de refactorizaciones definidas sobre las operaciones de dichas abstracciones.

En un caso ideal, si el lenguaje modelo tiene capacidad para representar todas las características de una familia de lenguajes, al analizar un código escrito en un lenguaje estáticamente tipado con genericidad, se produce un proceso de instanciación de las abstracciones del lenguaje definidas. De esta forma se mantiene un núcleo de operaciones comunes independientes del lenguaje en las distintas refactorizaciones. Bajo esta premisa el soporte utilizado podría corresponderse con una biblioteca de clases.

Debido a que el lenguaje modelo no representa todas las características de las familias de lenguajes a tratar, es necesario un soporte que permita la extensión de las nuevas características de lenguajes. El soporte utilizado para conseguir este objetivo es un framework definido como un diseño reutilizable de todo o parte de un sistema software descrito como un conjunto de clases abstractas y la forma en que esas clases colaboran [9]. El diseño evolutivo del framework se basa en identificar los puntos de extensión, las relaciones y restricciones que representen estas nuevas características. Los puntos de extensión se corresponden con clases que representan características del lenguaje modelo. De esta forma cuando una refactorización definida en el framework, contenga operaciones dependientes del lenguaje, puede hacer uso del punto de extensión definido.

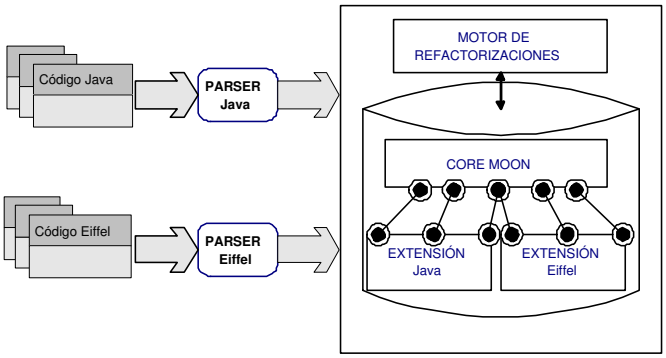

Figura 1. Arquitectura del framework MOON

Para poder abordar el problema de la independencia del lenguaje se necesita extender/instanciar el framework con las nuevas características del lenguaje objeto de estudio. La arquitectura presentada se corresponde con la Figura 1 dándose una descripción más detallada de los módulos CORE MOON y EXTENSIÓN JAVA en [10]. La evolución del framework está determinada a través de las validaciones tanto de otras refactorizaciones como de otros lenguajes.

3. Análisis de Refactorizaciones sobre el Lenguaje Modelo

El estudio de las refactorizaciones se realiza llevando cabo su definición en base al lenguaje modelo, con el fin de evaluar su soporte en una solución basada en frameworks. En dicho estudio se analizan, el conjunto de predicados y acciones

definible sobre el lenguaje, para posteriormente describir la refactorizaciones. Es necesario que la definición de precondiciones, acciones y postcondiciones se pueda expresar en base a la información representada en el lenguaje modelo. En caso de no poder definirse o bien ser dependientes de un lenguaje concreto se analizarán puntos de extensión al framework.

Una vez realizado este proceso, podemos afirmar qué refactorizaciones pueden ser realizadas con independencia del lenguaje origen, siempre que éste haya podido ser transformado a MOON.

3.1. Estudio de Predicados para la Definición de Refactorizaciones

En la definición de refactorizaciones es necesario establecer un conjunto de predicados básicos que recojan la mayoría de conceptos manejados en nuestro lenguaje modelo. Dadas las características de MOON, esto incluye los elementos básicos de LOO estáticamente tipados, con herencia avanzada y soporte a la genericidad con distintas formas de acotación.

Partiendo de definiciones previas [1], se han agregado nuevos predicados a lo largo del proceso que permiten definir relaciones entre los elementos objeto de las refactorizaciones. La definición de estos predicados se realiza de manera semiformal (no es un lenguaje con semántica precisa ejecutable), siendo verificables sobre la sintaxis propia del lenguaje modelo. Los predicados se han clasificado en tres categorías:

Básicos Conceptos fundamentales a la hora de trabajar con lenguajes que soportan la idea de clases (*Classes*), atributos (*Attributes*), métodos (*Methods*), nombres de propiedad (*Name*), signatura de métodos (*Signature*) y comparaciones de nombre y signatura (*EqualName*, *EqualSignature*).

Relaciones de herencia Predicados que expresan las relaciones de herencia entre clases, como ancestros, ancestros directos, descendientes y descendientes directos (*Anc, DAnc, Desc, DDesc*), la distinción entre propiedades heredadas (*IP*) e intrínsecas/esenciales (*EP*) y el aplanamiento de la clase respecto a la jerarquía de herencia ($Nombres_C$), similar a la idea de la forma *flat-short* en [11]. También se definen aquellas características o transformaciones en los métodos, fruto de los modificadores en herencia, como propiedades que se hacen efectivas (*MakeEffective*), propiedades diferidas (*Deferred*) o redefinidas (*Redefine*).

Relaciones de cliente Predicados que expresan las relaciones de cliente, cliente directo, proveedor y proveedor directo (*Client*, *DClient*, *Prov* y *DProv*), así como relaciones de uso a nivel de propiedades, como propiedades que referencian a otra (atributo o método) (*ReferenceProperty, ReferenceAttribute y ReferenceMethod*), bien a través de autoreferencias (*ReferenceSelf*) o incluso, a través de referencias paternas (*ReferenceSuper*).

Mediante este proceso de búsqueda y definición de predicados, se sustenta la base para la comprobación de la definición de refactorizaciones sobre el lenguaje modelo, definiendo un conjunto suficientemente completo. Además mediante este

proceso se ha detectado la necesidad de modificación de la gramática original de MOON, como en el caso de la ausencia de referencia paterna, de cara a definir el predicado *ReferenceSuper*.

3.2. Refactorizaciones de Especialización

Partiendo de la clasificación de refactorizaciones de bajo nivel (*low-level*) y alto nivel (*high-level*) definida en [12] y de la diferenciación entre refactorizaciones (*refactoring*) y refactorizaciones grandes (*big refactoring*) en [13], se han tomado, como caso de estudio, el subconjunto de refactorizaciones de bajo nivel [12] implicadas en refactorizaciones de alto nivel de especialización de clases, jerarquías o bibliotecas, ya que en trabajos anteriores [14] se ha validado el modelo en refactorizaciones de alto nivel de generalización.

En este contexto entendemos como especialización, el proceso contrario a la generalización y abstracción, analizando el conjunto de refactorizaciones implicadas, según estos autores, tomando como base ***Refactoring To Specialize: Subclassing, and Simplifying Conditionals*** [12] y ***Extract Hierarchy (375)*** [13]. El conjunto de refactorizaciones objeto de estudio se enfoca a las siguientes[4] ampliándose dicho estudio a aspectos vinculados a genericidad en [15]:

- Delete Unreferenced Class
- Create Member Method
- Delete Unreferenced Method
- Create Member Attribute
- Delete Unreferenced Attribute
- Moving Member Variable to Subclasses
- Moving Member Method to Subclasses

La definición de refactorizaciones para el modelo se ha realizado siguiendo una plantilla [13, 16]. La plantilla está compuesta por un nombre, una breve descripción de la refactorización, motivación, parámetros de entrada, precondiciones, acciones y postcondiciones. Las precondiciones y postcondiciones son definidas en base al conjunto de predicados analizados en la Sección 3.1. Por motivos de brevedad, en la Tabla 1 podemos ver un ejemplo de una definición de refactorización completa, estando disponible en [17] la definición completa del resto.

El estudio de las precondiciones nos lleva a clasificarlas en tres grupos:

Generales Los predicados básicos sobre los que se define la precondición están definidos por completo en el lenguaje modelo y pueden ser tratados en el núcleo del framework que soporta el lenguaje modelo.

Dependientes del lenguaje Los predicados pueden ser definidos, aunque su implementación tendrá que ser diferida a la instanciación concreta del framework, en el lenguaje de programación utilizado.

No verificables Los predicados no se pueden verificar a través del lenguaje modelo.

Tabla 1. Refactorización ***Move Member Variable to Subclasses***

Descripción	mover el atributo a los descendientes directos
Motivación	un atributo es sólo utilizado por alguno de los descendientes directos
Entrada	atributo (a) y clase (C)
Precondiciones	- El atributo no debe ser referenciado por otras clases ni por la propia clase: $ReferenceAttribute(a, C) = \emptyset$ - Las subclases directas no contienen un atributo con el mismo nombre (en lenguajes con ocultación de atributos): $\forall D \in DDesc(C), \nexists b \in Attributes(D)/EqualName(a, b)$
Acciones	1. $RemoveAttribute(a, C)$ 2. $\forall D \in DDesc(C) \Rightarrow AddAttribute(a, D)$
Postcondiciones	- El atributo no pertenece a la clase C: $a \notin Attributes(C)$ - El atributo es propiedad intrínseca/esencial en todos los descendientes directos de C: $\forall D \in DDesc(C)/a \in EP(D)$

Las acciones son primitivas que operan directamente a nivel sintáctico sobre código MOON, cuya gramática está definida en anteriores trabajos [1, 14]. Por ejemplo, en el caso de una refactorización **Move Member Variable to Subclasses**, dentro de las acciones implicadas tenemos *RemoveAttribute(a,C)* definido como *"Eliminar en la sustitución de la regla de producción ATTRIBUTE_DECS (regla 14) en la gramática definida sobre MOON, la declaración ATT_DEC (regla 16) que contiene un VAR_ID con Name(a) = VAR_ID"*. Todo el conjunto de acciones (*RemoveAttribute, RemoveMethod, RemoveClass, AddAttribute, etc.*) se define en base a esta idea.

3.3. Refactorizaciones con Precondiciones No Generales

Aunque la mayoría de las refactorizaciones estudiadas pueden ser evaluadas y realizadas sobre MOON, dado que las precondiciones y acciones necesarias están reflejadas, surgen problemas en aquellas refactorizaciones en las que las precondiciones no pueden ser evaluadas previamente. A continuación vamos a exponer tanto el caso de refactorizaciones cuya implementación es particular del lenguaje, como refactorizaciones cuyas condiciones iniciales para realizar su análisis no son abordables en el lenguaje modelo MOON.

Precondiciones relativas a los niveles de acceso. En algunas refactorizaciones, se define como predicado a verificar la accesibilidad de alguna propiedad por otra clase (e.g. en ***Move Member Method to Subclasses*** no podemos bajar el método, si accede a alguna propiedad no accesible por los descendientes directos). Dicho concepto, ligado a la idea de encapsulación y ocultación de

[4] Se ha optado por utilizar la nomenclatura definida en [12].

información, está presente en la mayoría de familias de LOO. El predicado que define la accesibilidad de una propiedad de una clase desde otra, ha sido marcado como *IsAccesible(property, provider_class, client_class)*. Sin embargo la información disponible en MOON no incluye modificadores de acceso (e.g. Java, C++, C#, etc.) o listas de exportación (e.g. Eiffel), por lo que la implementación de dicho predicado, debe delegarse en la instanciación concreta del framework para cada uno de estos lenguajes.

En el caso de cada familia de lenguajes, se dará una implementación concreta al predicado en base a la información, ahora sí, disponible. Por ejemplo, en familias de lenguajes que incluyen modificadores, el predicado dependerá del nivel concreto que define el modificador entre la clase proveedora y cliente, dependiendo de la relación de cliente, herencia, agrupación, etc y en familias con listas de exportación dependerá de la participación de la clase cliente, en la lista de clases a las que se exporta la propiedad. La implementación final se fundamenta en la información completa contenida en el código fuente original, aunque la refactorización se pueda definir a través de frameworks.

Refactorizaciones dependientes del control de flujo En refactorizaciones de alto nivel podemos ver cómo la información necesaria para iniciar la refactorización, tampoco está disponible. En concreto la refactorización de especialización ***Subclassing and Simplifying Conditionals*** [12] necesita comprobar los condicionales del código y extraer la lista de instrucciones correspondientes a las ramas **if**, **then** y **else**. Esta refactorización es similar a la planteada en [13] como ***Extract Hierarchy (375)*** cuyos casos particulares más simplificados se tratan en refactorizaciones de nivel medio como ***Replace Type Code with Subclasses (223)***, ***Replace Type Code with State/Strategy (227)*** y ***Replace Conditional with Polymorphism (255)***. Dado que en MOON los condicionales han sido eliminados (simplificados) no podemos tomar decisiones en refactorizaciones de este tipo.

Básicamente todas las refactorizaciones que parten del estudio de sentencias de control de flujo (condicionales e iteraciones), como por ejemplo las refactorizaciones ***Replace Iteration with Recursion*** o ***Replace Recursion with Iteration***[5], no pueden ser llevadas a cabo con la información mínima disponible.

Como conclusión a este apartado, podemos ver como la validación de refactorizaciones realizada, complementa el trabajo previo en [1] donde se inició la definición de predicados y operaciones de refactorización sobre el modelo y se definió una instanciación concreta para Eiffel. A continuación se avanza en dicha línea con el estudio de un nuevo lenguaje.

[5] Aunque dichas refactorizaciones no aparecen en [13], en el catálogo en web (http://www.refactoring.com/catalog/) mantenido por el autor, se añaden la refactorizaciones como contribuciones de Dave Whipp e Ivan Mitrovic, respectivamente.

4. Instanciación de GJ sobre el Framework

Tomando como base el framework que conceptualiza las abstracciones del lenguaje modelo MOON, este apartado pretende ver su capacidad de adaptación con respecto a un nuevo lenguaje que soporte polimorfismo paramétrico de forma explícita. Para poder comprobar la validación de dicha adaptación, se deben presentar las características especificas del lenguaje objeto del estudio e incorporarlas al propio framework, en el caso de no soportarlas directamente, mediante extensiones de las clases ya existentes. El lenguaje elegido para comprobar dicha capacidad de adaptación ha sido GJ (Generic Java). GJ es una de las propuestas para definir tipos genéricos en Java a través de una extensión del lenguaje que se distingue por enfrentarse al problema del código legado [18, 19, 20]. En el estudio presentado sólo se van a tener en consideración las características concretas de GJ en lo referente a genericidad evitando algunas características anexas al lenguaje Java (*inner class, anonymous class, static, ...*) [21].

En la exposición de esta sección primero se presentan las abstracciones del framework del lenguaje modelo, únicamente en lo referente a genericidad, para posteriormente, adaptarlas a las nuevas características del lenguaje GJ.

4.1. Abstracciones referentes a Genericidad en el Framework MOON

Al soportar MOON genericidad, los tipos son clasificados en parámetros formales (FORMAL_PAR) y tipos implementados por las propias definiciones de las clases (CLASS_TYPE). Cuando la definición de una clase no es genérica la asociación entre la clase (CLASS_DEF) y el tipo (CLASS_TYPE) es única, es decir se considera a una clase como una construcción lingüística en un LOO que se usa para implementar tipos. Por tanto, el tipo vendrá dado directamente por la definición de la clase. Cuando la definición de una clase es genérica, se puede decir que es una clase determinante de un conjunto potencialmente infinito de tipos [1]. Cada una de las instanciaciones/derivaciones genéricas realizadas se corresponde con distintos tipos (CLASS_TYPE). La definición de una clase genérica consta de una lista de parámetros formales. El modelo MOON soporta dos variantes en lo referente a la acotación de los parámetros formales: subtipado [22, 23] o conformidad [24] (variante S) y cláusulas **where** [25, 26] (variante W). Ambas persiguen acotar las características de los parámetros formales para garantizar la corrección de tipos en las instanciaciones genéricas, determinando un conjunto de sustituciones válidas para los parámetros formales. La acotación por cláusulas **where** consiste en asociar al parámetro formal, una cláusula que contiene las signaturas de las propiedades que debe tener todo tipo que se quiera utilizar como parámetro real. Si un parámetro formal no está acotado, el tipo del parámetro real en una sustitución, estará acotado por el tipo universal *Object* definido en MOON.

Dentro del conjunto de tipos obtenidos a partir de las instanciaciones genéricas se distinguen dos subconjuntos, tipos completos o completamente instanciados y los tipos no completos o no completamente instanciados:

- Los tipos completos, procedentes de instanciaciones genéricas completas (*completed = true*), son aquellos cuyo conjunto de parámetros reales no contienen ningún parámetro formal (FORMAL_PAR), es decir, el conjunto de parámetros reales permanece fijo.
- Los tipos no completos, procedentes de instanciaciones genéricas no completas, (*completed = false*), cuyo conjunto de parámetros reales contienen al menos un parámetro formal (FORMAL_PAR), es decir, el conjunto de parámetros reales permanece variable dependiendo del contexto.

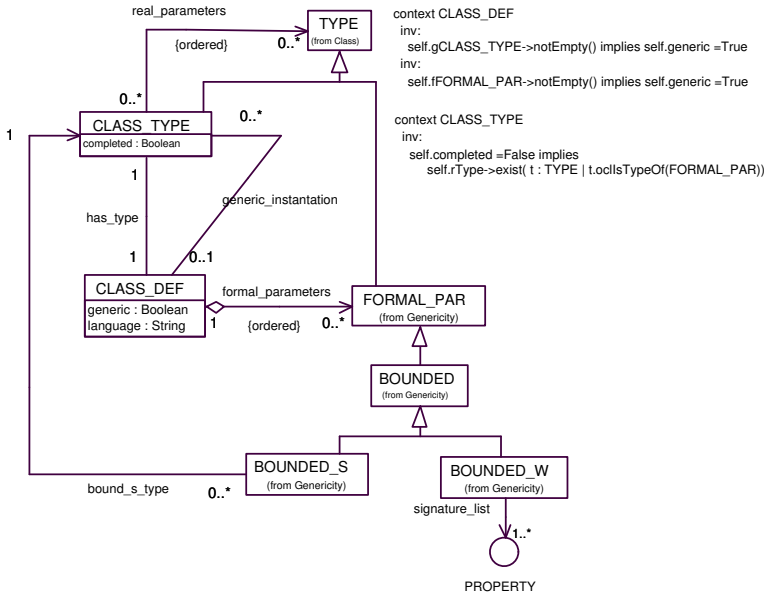

Figura 2. Definición de clases genéricas en MOON soportando acotación por subtipado (o conformidad) y acotación por cláusulas ***where***.

De la definición anterior se deduce que los tipos procedentes de instanciaciones genéricas no completas están contenidos dentro de clases genéricas, ya que utilizan un parámetro formal en la instanciación. Tanto los tipos procedentes de parámetros formales como los procedentes de instanciaciones genéricas no completas tendrán un ámbito local a la definición de la clase que les contiene, mientras que los tipos procedentes de instanciaciones genéricas completas tendrán un ámbito global. En la Figura 2 se muestra un diagrama de clases que representa las abstracciones de MOON, sus relaciones y un conjunto de invariantes definidos sobre la definición de la clase (CLASS_DEF) y sobre los tipos procedentes de las definiciones de las clases (CLASS_TYPE).

4.2. Características del Lenguaje GJ

Algunas de las características relevantes de GJ a tratar en el framework propuesto son: permitir genericidad en los métodos, permitir covarianza en el valor de retorno de los métodos, incorporar nuevos tipos de datos para mantener una coexistencia con el código no genérico legado (tipos *raw* y *erase*) e incorporar acotación F. En este estudio se ilustra la validación del framework respecto a la acotación F.

Un tipo recursivo es aquél cuya especificación está definida en términos del propio tipo. La acotación F es una variante de la acotación por subtipado cuando se trabaja con tipos mutuamente recursivos en el conjunto de parámetros formales de la clase genérica que les contiene. Las reglas sintácticas del lenguaje modelo MOON soportan directamente la acotación F, por abordar la acotación de subtipado y los tipos procedentes de instanciaciones genéricas no completas, como se puede observar en el ejemplo de la Tabla 2. Sin embargo, a la hora de dar soporte sobre el framework, se debe reflejar la semántica de la coexistencia de ambos tipos de acotaciones: subtipado y acotación F. La evolución del framework consiste en:

1. Añadir una clase que represente el parámetro formal acotado con la variante F, BOUNDED_F, que extienda a la clase BOUNDED_S.
2. Añadir la implementación de un método en la clase BOUNDED_F que permita obtener el parámetro formal recursivo que forma parte de la acotación (getRecursiveFormalPar():FORMAL_PAR).
3. Añadir una nueva restricción : "Si el parámetro formal está acotado por la variante F, entonces el CLASS_TYPE obtenido a través de la relación *bound_s_type* no está completamente instanciado (*completed = false*)".

5. Conclusiones y Líneas de Trabajo Futuro

En desarrollo del trabajo se ha mostrado la validación del soporte sobre frameworks, de un conjunto de operaciones de refactorización, así como el soporte a nuevos lenguajes, como el caso de GJ, llevándose a cabo una primera exploración en el diseño del framework.

Dentro de las líneas de trabajo abiertas, se apunta a la validación de un mayor conjunto de refactorizaciones sobre el framework. Por otro lado, se debe profundizar y ampliar el número de casos de estudio de familias de lenguajes (como en el caso de GJ), para su soporte a través de instanciaciones del framework en el proceso de refactorización.

En esta familia de lenguajes, se deja abierto el camino a la inclusión del soporte del Microsoft Intermediate Language (IL), dentro de la plataforma .NET [27], ampliando el número de lenguajes objeto de nuestro estudio.

El propósito final es la construcción de una herramienta que asista al mantenimiento y evolución con una cierta independencia del lenguaje empleado.

Tabla 2. Definición de una clase genérica con acotación F.

Código Java extensión GJ	Código MOON

`interface ConvertibleTo<A> {`
`  A convert();`
`}`
	`deferred class ConvertibleTo[A] signatures`
	`  attributes`
	`  methods`
	`  deferred convert:A;`
	`  body`
	`  ...`
	`end`

`class ReprChange <A implements`
`  ConvertibleTo<B>,`
`  B implements`
`  ConvertibleTo<A>> {`
`  A a;`
`  void set(B x) ...`
`  B get() ...`
`}`
	`class ReprChange[A ->ConvertibleTo[B],`
	`  B ->ConvertibleTo[A]]`
	`  signatures`
	`  atributes`
	`  A a;`
	`  methods`
	`  setB (x:B)`
	`  getB:B;`
	`  body`
	`  ...`
	`end`

Referencias

1. Yania Crespo. *Incremento del potencial de reutilización del software mediante refactorizaciones.* PhD thesis, 2000. Disponible en http://giro.infor.uva.es/docpub/crespo-phd.ps.
2. Mohamed Fayad, Goug Schmidt, and Ralph Johnson. *Building Applications Frameworks: Object-oriented Foundations of Framework Design.* John Wiley & Sons, 1999.
3. Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring.* PhD thesis, 2001.
4. Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings ISPSE 2000*, pages 157–167. IEEE, 2000.
5. Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, 2000.
6. Pierre Crescenzo and Philippe Lahire. Customisation of inheritance. In *Inheritance Workshop at ECOOP 2002*, pages 23–29. Black et al. Eds, Information Technology Research Institute, Jyvaskyla University Press, 2002.
7. Tom Mens and Michele Lanza. A graph-based metamodel for object-oriented software metrics. In Tom Mens, Andy Schrr, and Gabriele Taentzer, editors, *Electronic Notes in Theoretical Computer Science*, volume 72. Elsevier, 2002.
8. Donald Bradley Roberts. *Practical Analysis for Refactoring.* PhD thesis, 1999.
9. Don Roberts and Ralph E. Johnson. Evolving frameworks: A pattern language for developing object-oriented frameworks. In *Pattern Languages of Program Design 3.* Addison Wesley, 1997.

10. Carlos López. Definición de un soporte estructural para abordar el problema de la indepedencia del lenguaje en la definición de refactorizaciones. Technical Report DI-2003-03, Departamento de Informática. Universidad de Valladolid, septiembre 2003. Disponible en http://giro.infor.uva.es/docpub/lopeznozal-tr2003-03.pdf.
11. Bertrand Meyer. Tools for a new culture: Lessons from the design of the eiffel libraries. *CACM*, 33(9):68–88, 1990.
12. William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA, 1992.
13. Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison Wesley, 2000.
14. Y. Crespo, V. Cardeñoso, and J.M. Marqués. Un lenguaje modelo para la definición y análisis de refactorizaciones. In *Actas PROLE'01, Almagro, España*, November 2001. Disponible en http://giro.infor.uva.es/docpub/crespo-prole2001.pdf.
15. Raúl Marticorena, Carlos López, and Yania Crespo. Refactorizaciones de especialización en cuanto a genericidad. Definición para una familia de lenguajes y soporte basado en frameworks. In *Actas PROLE'03, Alicante, España*, November 2003.
16. Lance Tokuda and Don S. Batory. Evolving object-oriented designs with refactorings. In *Automated Software Engineering*, 1999.
17. Raúl Marticorena. Refactorizaciones de especialización sobre el lenguaje modelo MOON. Technical Report DI-2003-02, Departamento de Informática. Universidad de Valladolid, septiembre 2003. Disponible en http://giro.infor.uva.es/docpub/marticorena-tr2003-02.pdf.
18. Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Gj: Specification, 1998.
19. Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. GJ: Extending the java programming language with type parameters. 1998.
20. Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.
21. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.
22. Luca Cardelli. A semantics of multiple inheritance. *Semantics of Data Types*, LNCS(173):51 – 68, 1984.
23. Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4):471 – 523, 1985.
24. Bertrand Meyer. *Object-Oriented Software Construction*. McGraw Hill, 2nd edition, 1997.
25. Barbara Liskov. Programming methodology group progress report. Technical report, Laboratory for Computer Science Progress Report XIV, MIT Laboratory for Computer Science, 1977.
26. Barbara Liskov, Dorothy Curtis, Mark Day, and Sanjay Ghemawat. *Theta Reference Manual*. Programming Methodology group Memo 88. MIT Laboratory for Computer, 1995.
27. Andrew Troelsen. *C# and the .NET Platform*. Apress, 2001.