UN MÉTODO RIGUROSO PARA LA REUSABILIDAD DE SOFTWARE ORIENTADO A OBJETOS

Favre, Liliana
ISISTAN
Instituto de Sistemas de Tandil
Universidad Nacional del Centro
de la Pcia. de Buenos Aires
San Martín 57
(7000) Tandil
Pcia. de Buenos Aires
Argentina
e-mail:lfavre@tandil.edu.ar

RESUMEN

Se presenta en este trabajo un método riguroso para la reusabilidad de software orientado a objetos, enmarcado en el modelo de ciclo de vida transformacional y el lenguaje de especificación algebraica GSBL.

La estrategia básica para la generación semiatomática de código es la transformación de una biblioteca de componentes reusables mediante operadores de enriquecimiento, especialización, renombre y composición. Una componente reusable es un árbol de especificaciones algebraicas y esquemas de clases concretas: la raíz del árbol es la descripción más abstracta , una relación de implementación vincula nodos en el árbol y las hojas referencian esquemas de clases concretas en un lenguaje orientado a objetos.

El método posibilita la construcción de implementaciones en forma modular a partir de especificaciones abstractas reusando "piezas de implementación" referenciadas en las hojas de las componentes reusables.

Palabras claves: métodos formales; reusabilidad; especificaciones algebraicas; programación transformacional; programación orientada a objetos.

UN MÉTODO RIGUROSO PARA LA REUSABILIDAD DE SOFTWARE ORIENTADO A OBJETOS

1. INTRODUCCIÓN

Una tecnología ideal de reusabilidad de software orientado a objetos debería facilitar la identificación, especialización y composición de especificaciones abstractas que satisfacen un conjunto particular de requerimientos informales y la conversión automática de las mismas a versiones ejecutables.

Para lograr avances en esta dirección deben resolverse dos problemas cruciales:
- ¿cómo construir software para que sea factible su reusabilidad?
- ¿cómo definir y adaptar componentes reusables ?

Se propone en este trabajo un método riguroso de reusabilidad de software orientado a objetos basado en el modelo de ciclo de vida transformacional ([PAR90]) y el formalismo algebraico.

En el ciclo del desarrollo del paradigma transformacional se distinguen dos etapas básicas: análisis de requerimientos y optimización. El análisis de requerimientos provee una especificación formal que alimenta al proceso de optimización (refinamiento de versiones) controlado por el programador quien interviene en las decisiones inteligentes.

Son varias las ventajas que brindan las especificaciones formales frente a las especificaciones semi-formales de ingeniería de requerimientos:

- Tienen una semántica formal, sin ambigüedades, que permite razonar sobre corrección y consistencia.
- Permiten prototipación rápida para validar frente a los requerimientos. Además, en un sistema transformacional un prototipo puede ser reutilizado como especificación formal del sistema y como base para la aplicación de reglas de transformación que conduzcan a versiones ejecutables eficientes.

En el marco de la programación orientada a objetos el proceso de optimización puede verse como un continuo que va desde especificaciones formales de clases de objetos a código en un lenguaje orientado a objetos particular. Una clase es generada a partir de otras existentes mediante la aplicación de reglas de transformación que preservan relaciones semánticas.

Las clases de objetos pueden especificarse en forma independiente de implementaciones particulares a partir de especificaciones algebraicas. Se seleccionó como lenguaje de especificación GSBL ([CLE90]) teniendo en cuenta que permite especificaciones incompletas y por consiguiente construir especificaciones en forma incremental reutilizando componentes diseñadas con anterioridad.

Una componente reusable se define como un árbol de especificaciones algebraicas en GSBL y esquemas de programas en un lenguaje orientado a objetos (por ejemplo C++ o Eiffel) que describe una familia de módulos de software en diferentes niveles de abstracción:

- la raíz del árbol es la especificación más abstracta
- un nodo "padre" está relacionado con sus hijos por una relación de implementación
- las hojas referencian esquemas de clases concretas (C++ o Eiffel) que responden a la misma especificación

La estrategia básica para la generación semiautomática de especificaciones es la transformación de una biblioteca de componentes reusables mediante operadores para enriquecimiento, renombre, especialización y composición. La aplicación de estos operadores hasta las hojas de una componente reusable posibilita reusar clases concretas.

El método requiere de la existencia de un sistema de transformaciones que asista en la administración de componentes reusables y en la conversión de especificaciones a clases. El sistema debe registrar la "historia del diseño" a fin de tener una buena documentación, para modificar decisiones de diseño intermedias y para mantenimiento ante cambios en la especificación inicial.

Existen variados trabajos que prueban que la reusabilidad de componentes de software puede direccionarse a partir de descripciones algebraicas estructuradas ([HEN92]; [PRI91]; [SCH94]).

La línea metodológica y la base formal del enfoque seguido están relacionadas con estos trabajos, si bien en los mismos las componentes reusables vinculan especificaciones algebraicas estructuradas en diferentes niveles de refinamiento y se propone la derivación formal en un estilo funcional, por ejemplo ML([HEN92]). En este trabajo se propone integrar especificaciones algebraicas modulares y esquemas de clases concretas en lenguajes orientados a objetos como "piezas de implementación" a ser reusadas.

Se describen a continuación las características de interés para esta presentación del lenguaje de especificación GSBL, se definen componentes reusables y un método riguroso para la reusabilidad de software orientado a objetos.

2. EL NIVEL DE ESPECIFICACIÓN

Las clases de objetos pueden especificarse en forma abstracta a partir de especificaciones algebraicas estructuradas de tipos de datos. Esencialmente éstas definen un conjunto de álgebras heterogéneas.

Un tipo abstracto de datos es, a nivel sintáctico, una descripción de los sorts (géneros o conjuntos sobre los que se define el álgebra), de las operaciones que los manipulan y de las propiedades que tienen estas operaciones expresadas en forma de ecuaciones. A esta descripción se la llama especificación del tipo de dato abstracto.

Este enfoque nació hace aproximadamente 20 años y es uno de los más beneficiados en cuanto a avances teóricos y metodológicos. Existen por lo tanto diversos lenguajes de especificación algebraica. En este trabajo se seleccionó GSBL ([CLE90]). La característica destacable del lenguaje es soportar especificaciones incompletas que facilitan la descripción de aspectos parciales del problema a resolver. Esto permite la construcción incremental de especificaciones y la reutilización de componentes. Posee una semántica formal que posibilita encontrar y definir las condiciones que garantizan la corrección interna de las especificaciones. Se han incorporado algunos cambios para aumentar la capacidad expresiva del lenguaje y la posibidad de trabajar con funciones parciales.

Tradicionalmente, el proceso de construcción de especificaciones se basó en la descomposición horizontal: la especificación de un problema se resuelve descomponiéndolo en subproblemas que

eventualmente serán nuevamente descompuestos hasta llegar a problemas directamente especificables, luego estas sub-especificaciones se combinan para obtener la especificación final. Este enfoque es rígido e insuficiente para abordar la reusabilidad de clases: las decisiones tomadas en cada fase del diseño no pueden ser alteradas en etapas ulteriores. En general, en las aproximaciones iniciales existen aspectos que no conviene precisar dado que podrían producir inconsistencias con requerimientos aún no considerados.

La idea básica para evitar los inconvenientes anteriormente descriptos es construir especificaciones distinguiendo partes completas e incompletas integrando refinamientos horizontales y verticales. El refinamiento horizontal expresa relaciones de extensión y el vertical relaciones "completamiento". El refinamiento vertical introduce una relación de estructuración de especificaciones que está conceptualmente ligada a la noción de subclase en lenguajes orientados a objetos.

La incompletitud es tratada con semántica laxa ([SAN87]). En este tipo de semántica la clase de modelos no es isomorfa, es decir, una especificación incompleta admite modelos que no son isomorfos (cuando una especificación se completa todos sus modelos son isomorfos).

Se describen a continuación los aspectos de interés para este trabajo de GSBL. Una descripción detallada del mismo puede consultarse en [CLE90].

La definición de clase en GSBL tiene la forma:

CLASS nombre de clase
OVER<overlist>
SUBCLASS-OF <subclasslist>
WITH
  SORTS <sortlist>
  OPS <opslist>
  EQS <varlist> <equationlist>
DEFINE
  SORTS <sortlist>
  OPS <opslist>
  EQS <varlist> <equationlist>
END_CLASS

Los objetos del ambiente de especificación se denominan clases y están organizados en una jerarquía bidimensional de acuerdo a los dos tipos de refinamiento.

Las clases en GSBL son especificaciones incompletas con una estructura de componentes y pueden tener un sort principal que se llama igual que la clase. Las componentes son clases del ambiente o instancias locales de clases del ambiente.

La definición de una clase a partir de otras puede hacerse por extensión de la especificación dada, es decir *sobre* otra clase, o mediante la redefinición de alguna parte incompletamente definida, en este caso se la referencia como *subclase*.

La cláusula OVER corresponde a la construcción por extensión y la especificación se extiende con las componentes declaradas en <overlist>.

La cláusula SUBCLASS referencia especialización o refinamiento de las especificaciones de clases que aparecen en la lista <subclasslist>.

La cláusula WITH agrega sorts, operaciones o ecuaciones no completamente definidos. Este enriquecimiento es incompleto, es decir, faltan ecuaciones para especificar las nuevas operaciones o faltan operaciones para generar todos los valores de un sort. El siguiente ejemplo da la definición de la clase ANY. Esta es una clase predefinida en el lenguaje y es considerada superclase implícita de toda clase con un sort principal.

```
CLASS ANY
    WITH
      SORTS ANY
END_CLASS
```

La cláusula DEFINE agrega sorts, operaciones o ecuaciones completamente definidas o completa la definición de sorts u operaciones no completamente definidas que provienen de alguna superclase. Por ejemplo:

```
CLASS secuencia
OVER elemento :ANY
DEFINE
SORTS secuencia
OPS
funct inic_sec () secuencia
funct agreg_sec( secuencia , elemento) secuencia
funct vacia_sec(secuencia) boolean
funct tope_sec ( secuencia s: not vacia_secuencia(s)) elemento
funct resto_sec( secuencia s: not vacia_sec(s)) secuencia
...
EQS { c: secuencia; e,e1,e2: elemento}
vacia_sec(inic_sec()) ≡ true
vacia_sec(agreg_sec(s,e)) ≡ false
tope_sec(agreg_sec(s,e)) ≡ e,
resto_sec(agreg_sec(s,e)) ≡ s,
...
END-CLASS
```

Nótese que una especificación está implícitamente parametrizada por sus partes incompletas, es una componente genérica, y por lo tanto más fácilmente reutilizable. La especificación previa está implícitamente parametrizada en elemento. Por ejemplo para construir una secuencia de naturales se da la siguiente definición:

```
CLASS natsecuencia
   OVER natural
    SUBCLASS OF secuencia[ natural: elemento]
END-CLASS
```

3. QUÉ ES UNA COMPONENTE REUSABLE?

La reusabilidad de software depende de dos problemas cruciales:

- cómo identificar implementaciones existentes que puedan ser reusadas?
- cómo adaptar e integrar "piezas de implementación" existentes en una implementación consistente del sistema?

La identificación "correcta" de implementaciones existentes y la construcción de software automáticamente a partir de las mismas requiere contar con especificaciones formales de su comportamiento.

Para seleccionar una componente es conveniente contar con descripciones estructuradas en diferentes niveles de abstracción que reflejen las diferentes etapas en el proceso de desarrollo de un módulo de software. Una componente debe contar al menos de dos niveles: el que describe comportamiento en forma abstracta y el que referencia a diferentes implementaciones.

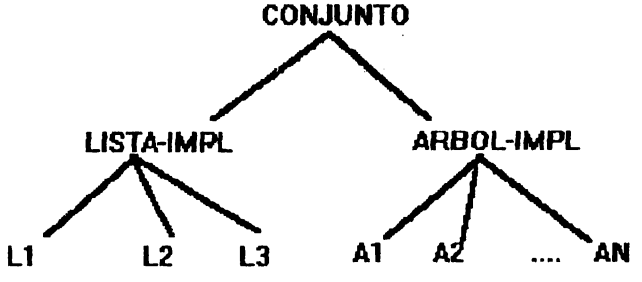
En este trabajo, una componente reusable se define como un árbol de especificaciones algebraicas en GSBL y esquemas de programas en un lenguaje orientado a objetos (por ejemplo C++ o Eiffel) que describe una familia de módulos de software en diferentes niveles de abstracción:

- la raíz del árbol es la especificación más abstracta
- un nodo "padre" está relacionado con sus hijos por una relación de implementación
- las hojas referencian esquemas de clases concretas (C++ o Eiffel) que responden a la misma especificación

Para describir formalmente la transición desde una especificación abstracta a una implementación se introduce formalmente una relación de implementación de especificaciones:

Si E y E1 son especificaciones, luego E es implementada por E1 (denotada por E~~~>E1), si E y E1 tienen la misma signatura y todo modelo de E1 es un modelo de E.

Por ejemplo, la gráfica siguiente muestra una componente llamada CONJUNTO asociada al tipo de dato conjunto finito :



donde CONJUNTO, LISTA-IMPL, ARBOL-IMPL son especificaciones algebraicas de comportamiento del tipo conjunto y L1,L2,L3,A1,A2,...,AN referencian clases concretas por

ejemplo, en C++ o EIFFEL, asociadas a diferentes representaciones para los tipos de datos que intervienen, todas responden a la especificación "padre".

Existen las siguientes relaciones de implementación: CONJUNTO~~~>LISTA-IMPL y CONJUNTO~~~>ARBOL-IMPL .

La relación de implementación permite clasificar componentes en una biblioteca de componentes reusables.

Las componentes reusables pueden expresarse por el operador

implementar(E,{ CR_1,...,CR_n })

donde E es una especificación y CR_1,...,CR_n son componentes reusables tales que sus raíces son implementaciones de E. En particular cada implementación puede ser vista como una componente reusable de la forma implementar(E,{}) y se denota $_{CR}$E.

Por ejemplo para la especificación CONJUNTO se define la siguiente componente reusable:

CR_CONJUNTO= $_{def}$implementar(CONJUNTO,
{implementar(LIST_IMPL,{$_{CR}$ L1,$_{CR}$ L2,$_{CR}$ L3},
implementar(ARBOL_IMPL,{$_{CR}$ A1,.....$_{CR}$ AN}})

La estrategia básica para la generación semiautomática de especificaciones es la transformación de la biblioteca de componentes reusables mediante operadores para enriquecimiento, renombre, especialización y composición. La aplicación de estos operadores hasta las hojas de una componente reusable posibilita reusar clases concretas.

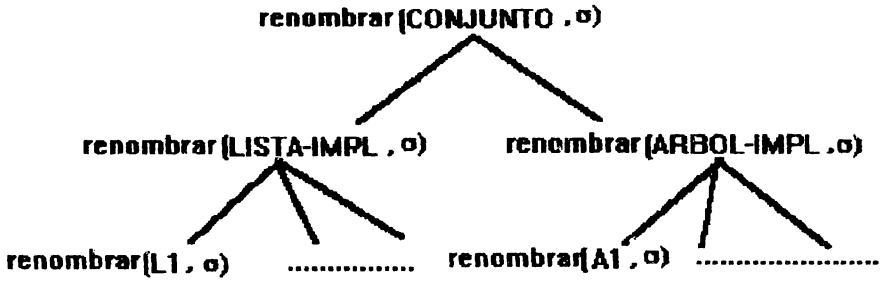
Se prueba en [WIR89] que los operadores de estructuración (composición, renombre, extensión y especialización) preservan relaciones de implementación:

Sean E, E', E1, E1' especificaciones tales que E~~~>E' y E1~~~>E1'. Además sean S un conjunto de sorts, F de operaciones , E un conjunto de axiomas, las siguientes relaciones de implementación se verifican:

- ***componer(E,E1)~~~>componer(E',E1')***
- ***extender(E, S,F,E)~~~>extender(E',S,F,E)***
- ***renombrar (E,σ) ~~~>renombrar(E',σ)***
- ***especializar(E,Σ) ~~~>especializar(E',Σ)***

Los operadores de estructuración de especificaciones se extienden para manipular componentes reusables. Informalmente, implica la aplicación simultánea de los mismos a todos los nodos. Por ejemplo, la composición de dos componentes reusables se define mediante la composición de sus raíces y recursivamente por la composición de todos sus hijos. (si uno de los argumentos componentes no tiene hijos luego, su raíz es compuesta con todos los nodos de los otros argumentos componentes).

La componente resultante de aplicar un renombre sintáctico que preserva la relación de implementación a la componente CONJUNTO se grafica a continuación:



Formalmente ,
renombrar $_{cr}$(implementar(E,{CR1,.....,CRn},ρ)=
$_{def}$ implementar(renombrar(E,ρ),{renombrar($_{CR}$ CR1, ρ),.....,renombrar($_{CR}$ CRn, ρ)})

4.HACIA UN MÉTODO RIGUROSO DE REUSABILIDAD...

Se describe a continuación un método riguroso para el reuso sistemático de componentes en el nivel de especificaciones de diseño. La línea metodológica está basada en el método formal para reusabilidad de componentes propuesto en [HEN92] si bien en los mismos las componentes reusables vinculan especificaciones algebraicas estructuradas en diferentes niveles de refinamiento y se propone la derivación formal en un estilo funcional. En este trabajo se propone integrar especificaciones algebraicas modulares y esquemas de clases concretas en lenguajes orientados a objetos como "piezas de implementación" a ser reusadas a partir de un método riguroso.

El método puede ser particionado en las siguientes etapas:

Descomposición

Dada una especificación E descomponerlas en apropiadas sub-especificaciones $E_1,E_2,.....,E_n$. Sea d una expresión que describe la descomposición.

Matching

Comparar cada sub-especificación E_i con la raíz R_i de una componente reusable CR_i. La identificación es correcta si R_i puede modificarse mediante la aplicación de operadores de forma tal que la versión modificada sea una implementación de E_i. Los géneros y operaciones de R_i deben conectarse con los de E_i por un apropiado renombre. La versión renombrada debe ser extendida con los sorts, operaciones y axiomas que completan E_i.Finalmente debe restringirse la signatura a la signatura visible de E.

Un problema abierto es la identificación de componentes reusables por su funcionalidad : el matching sintáctico puede realizarse comparando signaturas, la correctitud semántica del mismo involucra prueba de teoremas. Si la componente es grande puede existir una cantidad

exponencial de posibilidades de matching. Para que un reuso sea efectivo debe ser menos costoso identificar una componente que construirla.

Construcción de las componentes del matching

Para cada componente reusable CR_i seleccionada para cada sub-especificación E_i seleccionar una implementación cuyos hijos sean hojas y aplicarle los operadores $OP_1, OP_2, ..., OP_n$ que han sido usados en el matching previo, es decir se construye la especificación $OP_1(.....(OP_n(CR_{hoja})....)$.

$OP_1(.....(OP_n(Raiz(CR)).....)\rightsquigarrow OP_1(.....OP_n(R_{hoja})).....)$ es una relación de implementación y en particular $OP_1(.....OP_n(R_{hoja})).....)$ es una especificación correcta. Además teniendo en cuenta que la relación de implementación es transitiva $E\rightsquigarrow OP_1(.....OP_n(R_{hoja})).....)$ es una relación de implementación.

La especificación R_{hoja} tiene asociadas diferentes esquemas de clases en un lenguaje orientado a objetos .

Conversión de especificaciones a lenguajes orientados a objetos

A partir de las especificaciones obtenidas en las etapas previas se genera semiautomáticamente código orientado a objetos. Las transformaciones principales están relacionadas con las transformaciones de relaciones expresadas en las claúsulas OVER y SUBCLASS en el nivel de especificación, que deben traducirse a relaciones cliente y herencia , respectivamente en el nivel orientado a objetos.

A partir de las componentes reusables y mediante la aplicación de reglas de transformación que preservan relaciones de equivalencia operacional y descendencia operacional se construyen las nuevas clases.

Las relaciones expresadas en la cláusula OVER pueden generar clases parametrizadas, si las clases no están asociadas a clases definidas en el ambiente de especificaciones. Si la clase está definida se genera una clase concreta C++ o Eiffel resultado de la interpretación de un esquema asociado que el programador elige de la biblioteca. Las especializaciones y/o renombres se resuelven de acuerdo a lo especificado. Se redefinen los accesos de los métodos heredados y se crea la interfaz para la clase cliente.

Las relaciones expresadas en la cláusula SUBCLASS requieren de la creación de una clase base, resultado de la interpretación de un esquema de clase, tal como en el tratamiento de la cláusula OVER. En la clase asociada al tipo que se especifica se refleja la herencia de esta clase base. La misma es descendiente operacional de la clase de la biblioteca que fue interpretada. Pueden aparecer varias clases en SUBCLASS, esto expresa herencia múltiple.

Las transformaciones de relaciones OVER y SUBCLASS y de operadores de renombre y especialización son automáticas. Sin embargo una extensión que involucra nuevas operaciones requiere de la intervención del programador quien debe analizar diferentes representaciones, y generar la implementación de los métodos. Los axiomas en la especificación permiten validar frente a los requerimientos.

Se han realizado experiencias asociadas a esta etapa con la conversión de especificaciones al lenguaje C++ y actualmente con EIFFEL.

Composición de implementaciones

A partir de la expresión de descomposición d y de las especificaciones obtenidas en las etapas previas se compone la versión final a partir de las clases generadas. Las transformaciones se basan en las ideas descriptas en la etapa previa.

Optimización de versiones orientadas a objetos

Las versiones orientadas a objetos obtenidas pueden reestructurarse a partir de un sistema de transformaciones asociadas a este nivel (folding de clases, unfolding de clases, inmersión, abstracción, etc) . La estrategia es generar jerarquías de clases operacionalmente equivalentes a partir de la aplicación de reglas de transformación que preservan relaciones de equivalencia operacional y descendencia operacional en el nivel orientado a objetos. La "historia de las transformaciones" y la especificación algebraica que proviene de la etapa previa constituyen la especificación del sistema generado.

5.CONCLUSIONES

Se describió en este trabajo un método riguroso de reusabilidad de software enmarcado en el lenguaje de especificación algebraica GSBL y el paradigma transformacional. Se describió un marco formal de diseño de especificaciones algebraicas y la transformación de especificaciones a implementaciones en lenguajes orientados a objetos.

La construcción de software orientado a objetos a partir de la transformación de descripciones algebraicas estructuradas mediante operadores de renombre, extensión, especialización y composición provee los beneficios propios de la reusabilidad: reducción de costos de desarrollo, construcción de software confiable, etc. En el contexto de la programación orientada a objetos ofrece otras ventajas:

- La descripción algebraica de componentes de software brinda una clara descripción de la semántica de una jerarquía de clases que facilita el reuso de la biblioteca de clases y el mantenimiento. En el nivel de código orientado a objetos: la herencia y el binding dinámico dificultan la comprensión de la semántica de una jerarquía de clases en las versiones ejecutables finales y por ende el mantenimiento. Además, las relaciones de subclase surgen, en general, para compartir código y no comportamiento.
- Las decisiones fundamentales de diseño son registradas en la "historia del diseño" y no en el código de la clase generada, esto permite generar código eficiente. Por ejemplo, la secuencia de transformaciones que permite generar una clase concreta por especialización de otra clase se registra en la "historia del diseño" y no en el código. La especialización se logra en el nivel de programación orientada a objetos por el mecanismo de subclase a través de la herencia: el acceso a los objetos que son instancias de una clase puede resultar ineficiente en tiempo debido al binding dinámico y la herencia puede causar ineficiencia en cuanto a uso de espacio ya que los métodos son redefinidos en las subclases y pueden no ser utilizados.

6. REFERENCIAS BIBLIOGRÁFICAS

[CLE90]Clerici, S. A and Orejas, F.(1990); The Specification Language GSBL; Recent Trends in Data Type Specification; April.

[ELL90]Ellis,M.A. y Stroustrup, B., The Annotated C++. Reference Manual. AT&T Bell Laboratories, Murray Hill, New Jersey (1990).

[FAV95] Favre, L. ;Cuvillier, D.; Zivoder, I. Desde especificaciones algebraicas a clases C++. Revista Información Tecnológica. La Serena . Chile (Vol 6, N°6, 1995)

[GUE93]Guerreri, E. editor. Second International Workshop on Software Reusability. Lucca, Italia(1993).

[HEN92]Hennicker, R. , Wirsing, M. , A Formal Method for the Systematic Reuse of Specification Components. Lecture Notes in Computer Science 544, Springer-Verlag, Berlin (1992).

[KRU92]Krueger, C. , Software Reuse. ACM Computing Surveys : 24 (2), 131-183 (1992).

[MEY92] Meyer, B.;Eiffel: The Language; Prentice Hall. (1992)

[MEY94] Meyer, B.;Reusable Software :The Base Object-oriented Libraries; Prentice-Hall. (1994)

[PAR90]Partsch, H. Specification and Transformation of Programs. A Formal Approach to Software Development. Springer-Verlag (1990).

[PRI94]Prieto-Díaz, Schafer,W.,Cramer, J.,Wolf,S., First International Workshop on Software Reusability. Dortmund, Alemania (1991).

[SAN87]Sannella,D.; Tarlecki, A. Toward formal development of programs from specifications: implementations revisited. Springer-Verlag LNCS 249, pp 96-110 (1987)

[SCH94]Schafer, W., Prieto-Díaz, R. , Matsumoto, M., Software Reusability. Ellis Horwood (1994).

[WIR89]Wirsing, M. , Broy , M., A Modular Framework for Specification and Implementation. Lecture Notes in Computer Science 351, Springer-Verlag, Berlin (1989).