Hacia la formalización de la construcción de software
El método RAISE

Paganini, José Humberto; Figueroa Sebastián Marcos; Liberatori, Héctor Pedro

Universidad Nacional de Jujuy, Facultad de Ingeniería

Abstract

Los métodos de especificación formal surgen como una alternativa para la elaboración de software. En este contexto el presente trabajo, recurriendo a contribuciones desde la teoría y apelando a la utilización de un ejemplo de aplicación; relata sumariamente el método RAISE (Rigorous Approach to Industrial Software Engineering) y muestra el empleo del lenguaje RSL (RAISE Specification Language, con el propósito principal de presentar a la especificación formal como una alternativa interesante en el diseño de software. Esta alternativa propone anteponer a la escritura del código en un lenguaje de programación, todo un proceso de especificaciones encadenadas que determinan, sin ambigüedades "de que se habla"; y contribuyen a minimizar los errores lógicos y funcionales del sistema a elaborar.

Este largo proceso parte de una especificación inicial Aplicativa- Abstracta hasta modelos cada vez más concretos que desembocan en el código.

En este trabajo se propone explicar el concepto de especificación y se dan razones sobre lo que se entiende acerca de abstracta. Este fue elaborado por un equipo de investigación que desarrolla sus actividades en esta temática, en Lógica Computacional, análisis y desarrollo de herramientas software para estos fines.

El ejemplo que se muestra en este caso proviene de una aplicación de la logística a través del cual se expone una primaria especificación formal de un aeropuerto; primaria refiere al hecho de sólo se muestran las fases iniciales de todo el complejo proceso antes mencionado. De todo lo enunciado se extraen conclusiones y analizan las ventajas y desventajas de esta técnica.

Palabras Clave

RAISE, RSL, Especificación Formal

Introducción

Los sistemas informáticos tratan de ser, por un lado, un reflejo de hechos, fenómenos, procesos del mundo real, y son construidos para calcular, registrar, ordenar, controlar o administrar esos hechos y procesos (su eficiencia, dependerá de cuanto más acertada sea su correspondencia con el mundo real). Por otro lado los sistemas informáticos pueden tener una dependencia más laxa – o ninguna- con el mundo circundante y describir juegos y/o modelos especulativos, modelos para predicciones y en general temas y aspectos vinculados a lo que se conoce como realidad virtual.

Dado que los procesos, hechos y fenómenos, tanto vinculados al mundo real como el virtual, devienen cada vez más complejos, en gran medida, debido al aumento constante de la velocidad de cambio de estos mundos; esta circunstancia condiciona que la construcción de software afronte diversos problemas en el diseño y producción y en consecuencia el tratamiento de esos problemas lleve al estudio y la generación de nuevas técnicas, métodos y paradigmas. [3] [6]

En el contexto de tecnologías de diseño y producción de software se expone la idea de la formalización del software y el método de especificación formal RAISE.

Esta manera de elaboración de software, antepone a la concreción del código final, escrito en un lenguaje de programación, una sucesión de especificaciones formales vinculadas unas con las otras de manera tal que esa vinculación puede demostrarse con el auxilio de la lógica y de la matemática. [6] [10] De manera tal que se logra un encadenamiento que vincula las especificaciones estableciendo que una actual es consecuencia lógica de una precedente.[3] [6]

Cuando se especifica formalmente, se quiere significar que se logró una definición estricta y no ambigua, para una cosa, sustentada por fórmulas de la lógica y/o la matemática.

Hay que considerar que para todo estudiante de lógica, es bien conocido que no todo lo expresado en un lenguaje humano puede definirse formalmente [1]; sin embargo la especificación formal se basa en la certeza de que existen grandes porciones de discurso, sobre todo, referidos a aspectos técnicos o específicos, que rigurosamente pueden formalizarse.

La definición estricta, con el auxilio de fórmulas, trae consigo la ventaja de aclarar los conceptos sobre los que se habla; esto no solamente hay que analizarlo desde el punto de vista de la cosa en sí, que refiere a su naturaleza física, biológica o en general existencial, de la cual darán cuenta los técnicos y especialistas de cada disciplina; sino de se habla, en el seno de un sistema informático. Que es esa cosa del mundo real (o virtual) devenida en dato, entidad, proceso etc., y colocada dentro del marco de un sistema informático. O sea, debe ser tratada como una lista, un conjunto, un registro, un arreglo, o alguna otra expresión matemática/ lógica; y si es un hecho o un proceso, es una función, un mapa y respecto a su comportamiento si es secuencial, o concurrente. En general, cómo es más apropiado o conveniente que deba ser tratado.

Contribuye a ayudar a entender el método RAISE, la breve reseña Histórica del mismo que se expone a continuación.

El método RAISE, incluyendo su lenguaje RSL, [7] [5] y las herramientas software para correcciones, chequeos y traducción a lenguajes de programación; fue desarrollado como un esfuerzo colectivo en los proyectos ESPIRIT y LaCoS. Entre los investigadores participantes en esos proyectos se formó el *RAISE group*, liderado por Chris George e integrado por Peter Haff, Klaus Havelund, Anne E. Haxthausen, Søren Prehn, y Kim Ritter Wagner; [4]este grupo editó en Londres en 1992, la primera versión de “The RAISE Specification Language”, y al poco tempo “The RAISE Method”.

Una de las principales aplicaciones de esta tecnología fue la elaboración del Railways System (con el aporte de Dines Bjørner) ,[2] sistema completo que parte de la especificación formal hasta el código de programación de un sistema unificado de ferrocarriles para los países de la Unión Europea.

Otra importante contribución a esa metodología provino de Universidad Tecnológica de Dinamarca DTU (Denmark Tchnologic University), donde Dines Bjørner desarrolló el Tríptico del Software, como un interesante aporte teórico.

La metodología RAISE se instaló con gran fuerza en el UNU-IIST (Instituto Internacional de Tecnología de Software de la Universidad de las 11Naciones Unidas) desde cuya página web [7] iist.unu.edu se puede bajar bibliografía, ejemplos y herramientas de RAISE.

En nuestro país la función pionera de introducción la produjo la Universidad Nacional de San Luis UNSL, debido a sus contactos con el UNU-IIST, y difundida, en gran medida por la maestría en Ingeniería del Software de la Facultad de Ciencias Físicas, Matemáticas y Naturales de la UNSL; como consecuencia de esa difusión se crearon grupos de investigación en otras universidades como la UNJu (Jujuy) y UNCa (Catamarca).

La propuesta que plantea el presente trabajo es la de aportar elementos para facilitar un análisis del método RAISE y además mostrar, como resultado, los primeros pasos de una especificación formal con un ejemplo.

De lo expuesto se sacarán conclusiones y se señalarán ventajas y desventajas de esta tecnología.

Análisis del método RAISE, como Elemento del Trabajo y metodología

Los métodos formales, y en especial el método RAISE, puede ser empleado con éxito, para paliar inconvenientes y problemas que se producen en la elaboración de software, tales como los siguientes: [4] [6]

- Dificultad por parte del diseñador de software para interpretar sin ambigüedades y acabadamente lo

que el interesado (o cliente) solicita. Debido esto, en gran parte, por la falta de un lenguaje común entre ambos.

- Dificultad de detección de errores lógicos o funcionales. Pese a que la totalidad de los lenguajes de programación cuentan con herramientas que detectan y solucionan errores de sintaxis; inclusive algunos semánticos o de contenido; en general los errores lógicos o los funcionales son muy difíciles de localizar y muy costosa su corrección.
- Dificultades en la comunicación entre diseñadores de software y especialistas, técnicos y estudiosos del campo de los procesos físicos; debido al poco conocimiento de algoritmos específicos, técnicas y lenguajes de programación de sistemas informáticos por parte de estos últimos; los que sin embargo, casi seguro, tienen un buen desempeño en matemática y lógica.

Al imponer el método RAISE la especificación formal como etapa previa a la de elaborar el código, logra un efecto beneficioso con esa acción, al producir una disminución de los errores y de los inconvenientes inicialmente planteados. Esto puede verse con el siguiente análisis:[3]

En un software escrito los errores de sintaxis son fáciles de detectar y corregir; ya que sólo basta contrastar lo producido contra patrones y reglas; sin embargo con los errores semánticos (lógicos y/o funcionales) no ocurre lo mismo. Ya que estos últimos, en el caso de software, requieren un diseño especial de pruebas (y toda una teoría); normalmente son necesarias corridas, simulaciones o procedimientos que insumen esfuerzo tiempo y son costosos, o peor aún en el caso que se detecten a través de una falla producida con el software corriendo o *en servicio* suman a su costo los accidentes daños y pérdidas que puedan haberse producido por la falla.

Ante este panorama, la etapa de especificación, en función a los atributos de su formalidad, ataca las fuentes de errores, en sus inicios, en un tiempo donde recién se plantean los primeros requisitos y se empieza a construir un dominio de aplicación. Se fundamenta esto, porque el método en su acción elimina ambigüedades y construye sistemas axiomáticos que interpretan y ajustan los requisitos.

Pero lo realmente importante es que los errores semánticos en el código son, en general, errores de sintaxis en la especificación formal, y por lo tanto, en esa primera etapa son fáciles de detectar y corregir.

A su vez cabe considerar que el RSL, al estar basado en lógica de primer orden, y ser un lenguaje más cercano al discurso humano facilita las acciones tendientes a la detección y corrección de errores.

Otro aporte importante del método RAISE es que aparte de los diagramas funcionales, de entidad relación, operacionales, etc., que puedan elaborarse en la construcción de un sistema informático el hecho de colocar una etapa de especificación formal antes de la concreción del código de programación, enriquece la documentación del sistema, dado que el dominio y los requerimientos están formalizados en gran medida; permitiendo la revisión posterior e incluso la reutilización de partes. [6] [3] [5]

Para avanzar en el análisis de RAISE /RSL se expone el criterio de tratamiento de los conceptos de formalidad de este método.

Considerando que para ser formal un método debe poseer:[6]

- Un lenguaje – símbolos y reglas gramaticales para construir los términos.
- Normalmente, reglas para decidir si un término es bien formado.
- Una semántica- descripción del significado de los términos.
- Una lógica – un conjunto de reglas para determinar si los predicados

acerca de los términos son verdaderos.

Ante el enunciado de propiedades precedente, se infiere que los lenguajes de programación no son formales porque a ellos les falta una lógica. Lo opuesto ocurre con los métodos formales cuyo análisis ontológico descubre que abrevan en la lógica y la matemática, y por esa razón están cercanos al núcleo duro de las Ciencias de la Computación, que constituyen esas dos disciplinas.

Este concepto se refuerza por el hecho de que en la triada *sintaxis*, *semántica* y *pragmática* de la semiótica,[3] los métodos formales se recuestan sobre la semántica; y a su vez los lenguajes de programación lo hacen sobre la sintaxis y la pragmática.[8]

Hasta aquí se han aportado razones para aseverar que usando RAISE, es posible producir software que sea: [6]

- Más probable de ser correcto.
- Más fiable.
- Mejor documentado.
- Más fácil de mantener.

La razón por la cual al método que se está analizando se lo califica como *método riguroso* surge del siguiente análisis.

El concepto de riguroso se establece a través del conocimiento y utilización de los niveles de formalidad que se usan para la escritura de software.

Estos niveles son cuatro, empezando por un primer nivel, o inicial, que es el que sigue:

i) No se generan ocasiones de pruebas y chequeos.

Como ejemplo de esto se toma el código de programación; normalmente un programa no se chequea “per se”, debe generarse una rutina o procedimiento por afuera (cómo lógica de Hoare, u otro) [10] [9] para probarlo.

Siguiendo con la descripción de niveles:

ii) Generación de Ocasiones de prueba, inspeccionadas pero no probadas.

iii) Existencia de Pruebas de manera informal, del tipo de – “Se sigue inmediatamente que…”.

Puede decirse que todos los métodos formales son rigurosos (llegan hasta el tercer nivel), pero estrictamente sólo es riguroso un método con una base formal que permita un nivel adicional como el siguiente:

iv) Ocasiones de pruebas generadas y probadas formalmente.

Nivel que alcanza RAISE.

Del análisis comparativo con otros métodos formales surge otra de las particularidades de RAISE que es la de ofrecer muchas de las prestaciones de otros métodos formales, tales como los *basados en modelo* como Z, VDM o B; o los *algebraicos* como Larch y OBJ y los que consideran *álgebra de procesos* como CSP, CCS y Lotos. Con el adicional que además de esto, el método RAISE utiliza lógica modal y temporal para tratar con fenómenos de concurrencia.[6] [1]

Se puede también decir que:

El método RAISE está basado en dos nociones claves: [6]

Desarrollo separado

- Divide al sistema en subsistemas.
- Las especificaciones del sistema deben ser coherente con las de los subsistemas.
- Desarrolla las partes separadamente, pero teniendo en cuenta que, por más grande que sea la integración; esta no debe contradecir las propiedades originales.

Desarrollo gradual

- Se avanza en el desarrollo logrando nuevas especificaciones que resultan más detalladas que las anteriores.
- Asegura la relación entre una versión y la previa (inclusive en la implementación de código), y justifica que eso se cumple.

Se sigue el paradigma *invente y verifique*.

El desarrollo separado se efectúa por módulos de especificación formal; que en general tienen la forma:[4]

schema (Nombre)

class

Declaración$_1$

.

.
Declaración$_n$
end

Para n ≥ 0 donde una declaración comienza con la palabra reservada (*type*, *value*, *axiom*, *object*, *variable* o *channel*)

El desarrollo gradual, se consigue con refinamientos de especificaciones previas, es decir con avances que hacen a la especificación cada vez más concreta hasta llegar a la traducción a código.

Esto significa que normalmente se inicia el proceso con una especificación Aplicativa Abstracta.[4] [6]

La especificación Aplicativa Abstracta refiere a un *schema* cuyo tipo principal es abstracto. Como anteriormente se dijo, los tipos que se declaran pueden ser reservados (definidos en el lenguaje como Nat, Real, etc.), [4] o declarados mediante una expresión. En el caso de declarar un tipo abstracto este no tiene ninguna declaración. Un tipo (*type*) va asociado a un conjunto de operadores; por ejemplo si se lo declara como conjunto se pueden utilizar operadores de conjunto como unión, intersección, pertenencia, etc.; en cambio el tipo abstracto no tiene operadores. En rigor tiene sólo dos = y ≠ que determinan si algo es o no de ese tipo.

Esto puede visualizarse con un ejemplo. Supongamos que se declara el tipo abstracto Automóvil, en principio sólo puedo decir que algo es o no un Automóvil luego se va refinando y se dirá que un es un conjunto de partes, o es algo que cumple una función de transporte, o ambas cosas, u otras, que perfeccionarán la definición.

La especificación va sesgando la definición; de tal forma que el propósito por el cual se declaró ese tipo se irá estableciendo con las definiciones. Si se desea realizar un sistema para comercialización de Automóviles un valor importante sobre el que se trabajará será el precio; en cambio si se trata de carreras de fórmula 1, el comportamiento y las restricciones de la especificación serán distintos.

Entonces hay que considerar de que se habla cuando se habla de Automóvil:

De un objeto comercial
De un dispositivo para correr carreras
De un medio de transporte
De un objeto estético
De combinaciones posibles
Etc.

Lo que indica que se debe expresar dentro de un marco y un modelo, dicho esto con el concepto de *Marco* y *Modelo* de la Lógica Modal.[1] [10]. En consecuencia a esto, se construirá el código adecuado.

El proceso aquí sintetizado es el que efectúa el método RAISE.

Ejemplo de Especificación Aplicativa Abstracta de un Aeropuerto

Se exponen como resultados las especificaciones formales efectuadas en RAISE.

Es importante señalar que en las especificaciones formales mostradas a continuación se utilizan los siguientes operadores: [4]

∀ Cuantificador universal indica *Para Todo*
∃ Cuantificador existencial indica *Existe*
∧ Operador lógico *and*
∨ Operador lógico *or*
⇒Operador lógico de implicancia
~ Operador lógico de negación
→ Indica función *da o produce*
≡ Indica equivalencia
= = Símbolo de *variant* se utiliza cuando algo puede tener más de un valor posible
≠ Indica distinto *no igual*

Para reforzar el concepto de desarrollo separado, se declaran, en forma previa, y en un módulo separado los tipos que intervendrán en la especificación de aeropuerto.

La sintaxis empleada en las declaraciones de *value* se leen e interpretan de la siguiente forma:

La función appropriate de tipo par ordenado Plane, Airstrip, produce (→) un Booleano (verdadero o falso).

O en forma abreviada:

appropriate toma Plane por Airstrip y devuelve (o da) Bool (verdadero o falso)

Entonces:

TYPES MODULE

```
scheme TYPES =
class
   type
     Plane, Airstrip, Airside, Airspace,
     Occupancy = = vacant|
              occupied_ by(occupant: Plane),
     Longitude, Latitude, Height: Real
   value
     appropriate: Plane x Airstrip →Bool
     position: Plane →
         (Longitude x Latitude x Height)

   object
     T: TYPES
end
```

En esta especificación los tipos *Plane, Airstrip, Airside* y *Airspace* son abstractos. El tipo ocupación (*Occupancy*) es un *variant,* tipo reservado en RSL que indica que puede tomar dos valores alternativos, o *vacant* (vacía o vacante), o está ocupada por un avión se declara un objeto T de tipo TYPES que contiene todos los tipos y funciones declaradas en el *scheme* TYPES. Fijado esto, se utiliza el objeto T en la especificación formal de Aeropuerto.

Es importante destacar que las razones técnicas por las cuales una pista es apropiada para un tipo de avión, no se consideran en este esquema, razón por la cual la función *appropriate* se deja subespecificada.

Para la especificación Aplicativa Abstracta de Aeropuerto se adopta el nombre AIRPORT0 (AIRPORT cero), para indicar con ese número que es inicial.

```
scheme AIRPORT0 =
class
   type
     Airport
   value

     /*Generators*/
     landing: T. Plane x T. Airstrip x
              Airport→ Airport,
     takeoff: T. Plane x T. Airstrip x
              Airport→ Airport,
     parked: T. Plane x Airport→ Airport,

     /*Observers*/
     is_ flying: T. Plane x Airport→Bool
     sit-in: T. Airstrip x Airport→
          T. Occupancy,

     /*Derived observers*/
     consistent: T. Airstrip x Airport→Bool
     consistent(s, a) ≡
     ∀s1,s2: T. Airstrip, a: Airport,
          p1,p2: T. Plane •
     sit-in(s1,a) = T. occupied_ by (p1) ∧
     sit-in(s2,a) = T. occupied_ by (p1)⇒
     (s1 = s2) ∧ T. appropriate(p1,s1) ∧
     sit-in(s1,a) = T. occupied_ by (p1) ∧
     sit-in(s1,a) = T. occupied_ by (p2)⇒
     (p1 = p2),
     /* --- 1---*/

     is_ parked:  T. Plane x Airport→Bool
     ∀a: Airport, p:T. Plane,
        ~∃s: T. Airstrip •
     is_ parked(p, a) ≡
     sit-in(s, a) = T. occupied_ by (p) ∧
     ~is_ flying(p, a),
     /* --2-- */

     /*Observers guards*/
     can_ landing: T. Plane x T. Airstrip x
                   Airport→ Bool
     ∀a: Airport •
     can_ landing(p, s, a) ≡
     ∃ s: T. Airstrip • sit-in = T. vacant  ∧
                    T. appropriate( p, s) ∧
     is_flying(p,a),

     can_ takeoff: T. Plane x T. Airstrip x
                   Airport→ Bool
     ∀a: Airport •
     can_ takeoff(p, s, a) ≡
     ∃ s: Airstrip • sit-in =
       T. occupied_ by (p)∧
       T. appropriate( p, s)
       pre
         is_ parked(p, a),

axiom
  [is_ flaying- landing]
  ∀a: Airport, p1, p2:T. Plane,
                    s: T. Airstrip •
```

is_ flaying(p1, landing(p2, s, a) ≡
(p1=p2) ∨ (is_ flying(p1) ∧
landing(p2, s, a) ∧(p1≠p2)

[is_ flaying- takeoff]
∀a: Airport, p1,p2:T.Plane,
s: T. Airstrip •
is_ flaying(p1, takeoff(p2, s. a) ≡
(p1≠ p2) ∧
is_ flaying(p1, a) ∧~ is_ flaying(p2, a),

[is_ flaying- parked]
∀a: Airport, p:T. Plane, s:T. Airstrip •
Is_ flaying(p, parked(p, a))≡ **False**,

[sit-in- landing]
∀ a: Airport, p:T. Plane,
s1, s2 :T. Airstrip •
sit-in(s1, landing(p, s2, a)) ≡
(s1 = s2) ∧ sit-in(s1, a) = T. vacant,

[sit-in- takeoff]
∀a: Airport, p:T. Plane, s:T. Airstrip•
sit-in(s, takeoff(p, s, a)) ≡
sit-in(s, a) = T. occupied_ by(p),

[sit-in- parked]
∀a: Airport, p:T. Plane
~∃s: T. Airstrip •
sit-in(s, parked(p, a)) =
T. occupied_ by(p),

[consistent- landing]
∀a: Airport, p:T. Plane, s:T. Airstrip•
consistent(s, landing(p, s, a) ≡
sit-in(s, a) = T. vacant ∧
T. appropriate(p, s),

[consistent- takeoff]
∀a: Airport, p:T. Plane, s:T. Airstrip•
consistent(s, takeoff(p, s, a) ≡
sit-in(s, a) = T. occupied_ by(p) ∧
T. appropriate(p, s),

[consistent- parked]
∀a: Airport, p:T. Plane •
consistent(s, parked(p, a)) ≡
is_ parked(p, a),

[can_ landing- landing]
∀a: Airport, p:T. Plane, s:T. Airstrip•
can_landing(p, s,landing(p, s, a)≡**True**,

[can_ landing- takeoff]
∀a: Airport, p:T. Plane, s:T. Airstrip•
can_landing(p, s,takeoff(p, s, a)≡**False**,

[can_ landing- parked]
∀a: Airport, p:T. Plane •
can_landing(p, s, parked(p, a)) ≡ **False**,

[can_ takeoff- landing]
∀a: Airport, p:T. Plane, s:T. Airstrip•
can_ takeoff(p, s, landing(p, s, a)) ≡
False,

[can_ takeoff- takeoff]
∀a: Airport, p:T. Plane, s:T. Airstrip•
can_ takeoff(p, s, takeoff(p, s, a)) ≡
True,
[can_ takeoff- parked]
∀a: Airport, p:T. Plane, s:T. Airstrip•
can_ takeoff(p, s, parked(p, a)) ≡**True**,

axiom
[Flying consistence]
∀x: T. Longitude, y: T. Latitude,
z: T. Height, p1, p2: T. Plane •
T. position(p1) = T. position(p2)⇒
(p1= p2)
end

Los comentarios en RSL comprenden el texto encerrado entre /* y */.
Se aclaran algunos de ellos:
--1—La función de consistencia, que devuelve un resultado verdadero o falso; establece que un avión ocupa sólo una pista y que una pista puede ser ocupada por un único avión. En el caso de que un avión ocupe una pista esta debe ser apropiada.
--2—Un avión estacionado no está en ninguna pista. Está en zona de embarque *Airside*.

Discusión de la Especificación Aplicativa Abstracta

Acá se nota claramente el empleo de un tipo abstracto *Airport* o sea que no tiene ninguna expresión; los únicos operadores posibles de ser utilizados son = y ≠ que determinan si algo es o no es un aeropuerto.
Sin embargo hay valores particulares del tipo *Airport* ; funciones que devuelven un aeropuerto, funciones que en rigor son aeropuertos, por lo siguiente:
Un *Airport* es un lugar donde un avión aterriza, despega o se estaciona entonces los *valúes*: *landing, takeoff* y *parked* generan un aeropuerto. Por esa razón están agrupados con el comentario *Generators*.
Los valores observadores son funciones que no devuelven resultados del tipo principal, o sea que establecen condiciones sobre un aeropuerto.
La única manera posible de operar con tipos abstractos, se logra mediante axiomas que vinculan un observador con un generador, ya que como se dijo, un observador establece condiciones y un generador es un aeropuerto.
Entonces estos axiomas resuelven casos como que ocurre cuando un avión sobrevuela un aeropuerto donde aterriza un avión [flying- landing] y la solución que ofrece el axioma es que si se trata de un solo avión ese avión que sobrevuela aterriza, o en el caso de ser dos aviones distintos uno aterriza y el otro sigue volando.
Estos axiomas más el de consistencia de vuelo [Flying consistence] y la función de consistencia forman un marco axiomático que contribuye a la definición de aeropuerto.

Avance en la especificación de Aeropuerto

Definida la fase de la especificación Aplicativa Abstracta, donde la consistencia, y existencia de axiomas certifican el cumplimiento de requisitos fundamentales para un aeropuerto. El próximo paso es construir una especificación más concreta introduciendo estructuras para los tipos creados en la especificación abstracta.
Entonces:

scheme B_SET =
class
type
Set, Elem

value
empty: Set,
Add: Elem x Set →Set,
remove: Elem x Set →Set,
is_ in: Elem x Set → **Bool**

axiom

[is_ in_ empty]
∀e: Elem • ∽is_ in(e, empty),

[is_ in_ add]
∀s: Set, e, e': Elem •
is_ in(e', add(e,s) ≡ e=e' ⋁ is_ in(e',s)

[is_in_remove]
∀s: Set, e, e': Elem •
is_ in(e' remove(e,s)) ≡ e ≠ e' ∧
is_ in(e', s)
end

El módulo precedente, es un módulo independiente que especifica un conjunto Set), con sus valores vacío, agregue y saque.
A continuación se especifica un vector o arreglo S_ ARRAY, de la manera siguiente:

scheme S_ARRAY=
class
type
Array,
Selem,
Index, vinit

value
init: Array,
assign: Index x Selem x Array →
Array,
obtain: Index x Array → Selem

axiom
[obtain- init]
∀i: Index • obtain(i, init) ≡ vinit,

[obtain_assing]

∀i, i':Index, e:Selem, v:Array •
assign (i',obtain(i, e), a) ≡
if i=i' **then** e **else** obtain(i',a) **end**
end

En función a las especificaciones precedentes B_SET y S_ARRAY, se instancian para avanzar en la definición de tipo originalmente abstracto Airport.
Esto es:

scheme Airport1 =
class
object
AS:B_SET(T{T. Plane for Elem})
S:S_ARRAY(T{T.Occupancy for Selem, T. vacant for init})
type
Airport = AS.Set x S.Array
end

El schema Airport1, se lo expresa en forma corta, faltan las declaraciones de values y axiom.

Discusión de los avances en la especificación de un aeropuerto

Se arribó al schema Airport1, más concreto, con el auxilio de dos especificaciones una de un conjunto y otra de un arreglo; que se asimilaron Airport1 bajo la circunstancia de que existe una zona, que se denota con espacio aéreo (Airspace); esta zona comprende todo lugar donde el aeropuerto, mediante sus equipos, detecta aviones. En ella los aviones se aproximan al aeropuerto y esperan su turno para utilizar alguna pista para aterrizar. Entonces usando la especificación de conjunto tomando el tipo p: T.Plane en vez de e: Elem tenemos un conjunto de aviones ps: P_SET volando en el espacio aéreo que llegan para pedir el permiso para aterrizar, y además las pistas se pueden describir mediante el arreglo S_ARRAY en donde se debe usar T.Occupancy en vez de Selem y T.vacant en vez de init. En este caso se necesita un índice entero como un atributo estático de las pistas.

Para un análisis integral del método RAISE es necesario puntualizar algunas de sus desventajas.

Que se expresan de la manera siguiente:

- La aplicación de este método requiere estudio formación y, competencia en lógica computacional y seguramente un equipo de desarrolladores dedicados a la tarea.
- Otra desventaja de este método, la constituye el hecho, que por ahora, no condice con las técnicas ágiles nuevas. Ya que el presente método adolece del inconveniente de que emplea un tiempo mayor para concretarse. Hay que considerar todo lo empleado en la especificación formal y sumado a esto, prácticamente es imposible es la elaboración de prototipos, durante el proceso de creación del software.

Estas condiciones puntualizadas lo muestran desventajoso, sobre todo bajo presiones y requerimientos del mundo moderno que imperiosamente busca en los problemas soluciones rápidas. Lo aquí expuesto se ejemplifica con dos paradigmas opuestos, por un lado, la idea de presentar rápidamente un sistema y luego mejorarlo con el uso, que puede sintetizarse con el adagio *Los melones se acomodan conforme el carro avanza*; versus el paradigma que establece que los resultados mejores se logran con planificación.

A su vez no es menor citar que dado que toda la tecnología del método RAISE se consigue bajo la forma GNU o de software libre, y que los principales estudios y avances en el tema se lleven a cabo en el UNU-IIST, [7] que se encuentra en Macao, República Popular China y en la DTU, sita en Dinamarca. Esto indica que está lejos de los centros de poder comercial y por ende de difusión.

Conclusión

En el presente trabajo, se cumplió lo propuesto que es aportar elementos para facilitar un análisis del método RAISE y además mostrar, como resultado, los primeros pasos de una especificación

formal de un aeropuerto, tomado como ejemplo, con las especificaciones AIRPORT0, abstracta y AIRPORT1, más concreta.

Si bien cabe destacar que AIRPORT1 no pudo extenderse por problemas de alcance del presente trabajo.

Se señala también que en el presente escrito se trabajó en la idea de que las palabras no significan por sí mismas, sino dentro de un contexto.

Este concepto proviene de la Lógica que establece un marco axiomático cómo contexto y desde allí lo toma el método RAISE.

A su vez se señalaron las ventajas y aportes del método RAISE y se puntualizaron sus desventajas.

A modo de cierre se puede suponer una situación donde se tenga el código de un software para el control del espacio aéreo de un aeropuerto, que no haya sido elaborado con el método RAISE, la única forma de saber si se contemplaron las condiciones de aterrizaje, despegue y estacionamiento de aviones con la ocupación de pistas, es la realización de simulaciones, corridas y pruebas.

Con este procedimiento se puede encontrar un error lógico en la programación, pero quizá sea más difícil encontrar un mal funcionamiento por omisión de alguna condición. Ni pensar en las consecuencias de detectar el error en servicio.

En el caso de usar RAISE, el hecho de partir de una especificación abstracta hace que necesariamente se chequeen las condiciones, por una razón de sintaxis. A su vez, el RSL es un lenguaje basado en lógica de primer orden, que posee una gran cantidad de formas, herramientas y reglas de comprobación. También al ser un lenguaje más cercano al coloquial facilita la comprobación de condiciones y requisitos. Y, por último, al ser una etapa "papel y lápiz" si hay un error está muy lejos del tiempo de correr el software y se corrige fácilmente.

Agradecimientos

A Chris George, Dines Bjørner y Daniel Riesco por sus enseñanzas en RAISE.

Referencias

[1] Blakurn, P.; de Rijke, M.; Venema, Y.: "Modal Logic" Cambridge University Press. Cambridge UK 2001 ISBN 0 521 80200 8

[2] Bjørner, D.; Braad, J.,;Mogenen K.:"Models of Railway System: Domain" DTU Denmark 2000

[3] Bjørner, Dines: "Software Engineering 3". 2006 ISBN-10 3-540-21151-9 Springer , Berlin Germany.

[4] George, Chris et al. "The RAISE Specification Language" ISBN 0-13-752833-7 Prentice Hall International. London UK 1.992.

[5] Dasso, A.; Funes, A.:"Verification validation and testing in Software Engineering" http://es.scribd.com/doc/66785202/Verification-Validation-and-Testing-in-Software-Engineering

[6] George, Chris et al. "The RAISE Development Method" TERMA A/S Denmark. Prentice Hall (free circulation)

[7] International Institute of Software Technology UNU-IIST iist.unu.edu

[8] Marafioti, Roberto: "Charles S. Pierce: El éxtasis de los signos"ISBN 950-786-412-1 Biblos Buenos Aires Argentina 2004.

[9] Paniagua Arís, E.; Sánchez González, J.L.; Martin Rubio, F.: "Lógica Computacional" Thompson Madrid España ISBN 84-9732-182-0.

[10] Solís, F. S.; Sánchez Ante, G: "Fundamentos de Lógica Computacional" ISBN 968-24-6100-6 Trillas. México DF México 2002.