

# Tema 4: Ingeniería de Requerimiento

# Métodos formales aplicados en la industria del software

## Introduccion

El concepto de métodos formales involucra una serie de técnicas lógicas y matemáticas con las que es posible especificar, diseñar, implementar y verificar los sistemas de información. Hasta hace pocos años el desarrollo industrial de sistemas, usando dichas técnicas, era considerado un complejo ejercicio teórico e inviable en problemas reales. Las notaciones "oscuras" tomadas de la lógica, sin las suficientes herramientas de soporte, no podían competir con los lenguajes de cuarta generación y los ambientes de desarrollo rápido de los años 80 y 90; sin embargo, el mundo industrial cambió su actitud frente a los métodos formales. Por una parte, porque los lenguajes de especificación son cada vez más cercanos a los lenguajes de programación, las técnicas de desarrollo se adaptan mejor a los nuevos paradigmas, y las herramientas comerciales que soportan la calidad del software son métodos que se distribuyen comercialmente. Además, porque a medida que los sistemas informáticos crecen en complejidad las pérdidas causadas por fallas son cada vez mayores. Cuando "corrección certificada" se traduce en dinero, los métodos formales atraen a la industria, ya que su aplicación ayuda a lograr los estándares de calidad que la sociedad exige. La importancia de los métodos formales en la Ingeniería de Software se incrementó en los últimos años: se desarrollan nuevos lenguajes y herramientas para especificar y modelar formalmente, y se diseñan metodologías maduras para verificar y validar. Los modelos que se diseñan y construyen de esta forma, desde las fases iniciales del desarrollo de software, son esenciales para el éxito del futuro proyecto; ya que en la actual Ingeniería de Software constituyen la base que sustenta las subsiguientes fases del ciclo de vida, y porque los errores surgidos en ella tienen gran impacto en los costos del proyecto (Perry, 2006).

Desde hace varias décadas se utilizan técnicas de notación formal para modelar los requisitos, principalmente porque estas notaciones se pueden verificar "fácilmente" y porque, de cierta forma, son más "comprensibles" para el usuario final. Además, el paradigma Orientado por Objetos en la programación parece ser el más utilizado en la industria, y una forma de incrementar la confiabilidad del software y, en general de los sistemas, es utilizar los métodos formales en la ingeniería aplicada. Pero, aunque los métodos formales tienen un amplio recorrido, y su utilidad y eficiencia en desarrollos críticos están demostradas, todavía falta más trabajo para que la mayoría de ingenieros los conozcan y



apliquen. En parte esta labor la deben realizar las facultades que deben incluirlos en sus contenidos académicos; los profesores, quienes se deben formar e investigar mejor esta área del conocimiento; y los estudiantes, quienes deben tener una formación más sólida en matemáticas y lógica. Sólo con un trabajo mancomunado se podrá lograr que la labor del ingeniero de Software sea verdadera Ingeniería, y que el usuario final acepte los productos software como confiables. Los propósitos de los métodos formales son: sistematizar e introducir rigor en todas las fases de desarrollo de software, con lo que es posible evitar que se pasen por alto cuestiones críticas; proporcionar un método estándar de trabajo a lo largo del proyecto; constituir una base de coherencia entre las muchas actividades relacionadas y, al contar con mecanismos de descripción precisos y no ambiguos, proporcionar el conocimiento necesario para realizarlas con éxito. Los lenguajes de programación utilizados para desarrollar software facilitan la sintaxis y la semántica precisas para la fase de implementación, sin embargo, la precisión en las demás fases debe provenir de otras fuentes. Los métodos formales se inscriben en una amplia colección de formalismos y abstracciones, cuyo objetivo es proveer un nivel de precisión comparable para las demás fases. Si bien esto incluye temáticas que actualmente están en desarrollo, algunas metodologías ya alcanzaron un nivel de madurez suficiente para que se utilicen ampliamente. Existe una tendencia discutible a fusionar la matemática discreta y los métodos formales en la Ingeniería de Software. Efectivamente, en muchas temáticas de ésta se apoya la Ingeniería de Software, y no es posible ni deseable evitarlas cuando se trabaja con métodos formales; pero no es posible pretender que por el simple hecho de tomar algunos enfoques de la Matemática Discreta y emplearlos en aquella, se apliquen métodos formales. El principal objetivo de la Ingeniería de Software es desarrollar sistemas de alta calidad y, con los métodos formales, en conjunción con otras aéreas del conocimiento, se puede lograr. En este artículo se describe esa conjunción, y se estructura de la siguiente forma: en la segunda parte se definen los métodos formales; en la tercera se describe su utilidad en la Ingeniería de Software, su base matemática y se presenta un ejemplo ilustrativo; en la cuarta se detallan sus ventajas, y en la quinta se presentan las tendencias de la investigación en esta área del conocimiento.

## Que son los Métodos Formales

Aunque el término se utiliza ampliamente, no parece existir una clara definición acerca de qué son. En muchas ocasiones el término se emplea simplemente para indicar la utilización de un lenguaje de especificación formal, pero no incluye una descripción de cómo se utiliza o la extensión de su uso. Incluso el término "lenguaje de especificación formal" no es preciso, ya que no es claro si los



lenguajes "específicamente" tienen como objetivo diseñar la implementación, en lugar de especificar el problema. En este trabajo se utiliza el término "métodos formales" para describir cualquier enfoque que utilice un lenguaje de especificación formal, pero que describa su función en el proceso del desarrollo de software. Utilizar métodos formales necesariamente implica un proceso formal de refinamiento, razonamiento y prueba. El término "lenguaje de especificación formal" se utiliza aquí para referenciar un lenguaje en el que es posible especificar completamente la funcionalidad de todo o parte de un programa, de tal forma que sea susceptible de razonamiento formal, y que se fundamenta en una sólida base matemática, más en que si es lo suficientemente abstracto. La base teórica de los lenguajes de especificación formal varía considerablemente, la más común es la de los métodos formales. Los lenguajes de este tipo más utilizados en la industria son The Vienna Development Method -VDM- y Zed -Z-, que se basa en la teoría de conjuntos y la lógica de predicados de primer orden de Zermelo Fraenkel. VDM se utilizó primero en proyectos de gran tamaño, pero Z parece ganar popularidad recientemente. Aunque su sintaxis es diferente, ambos lenguajes están estrechamente relacionados y se basan, principalmente, en la teoría matemática de conjuntos. Un sistema matemático formal es "un sistema de símbolos con sus respectivas reglas de uso", todas recursivas. Este último requisito es importante ya que permite desarrollar programas para comprobar si un determinado conjunto de reglas se aplica correctamente; sin embargo, garantizar que se puede lograr utilizando métodos formales tiene un precio, ya que para muchas aplicaciones resulta extremadamente costosos. Las nociones fundamentales de especificación y verificación formal de programas se desarrollaron en los años 70, entre otros por Hoare y Dijkstra: "el primer paso en la construcción de un sistema es determinar un modelo abstracto del problema que se va a resolver sobre el cual razonar". Como no todos los aspectos del problema son relevantes en el sistema que se desarrolla es necesaria una observación cuidadosa para abstraer sus características más importantes, y utilizar un lenguaje formal para describir el modelo -especificación formal-. En las ciencias computacionales los "métodos formales" adquieren un sentido más lineal, y se refieren específicamente al uso de una notación formal para representar modelos de sistemas, y en un sentido aún más estrecho se refieren a la formalización de un método para desarrollar sistemas (Flynn & Hamlet, 2006). El término "métodos formales" se refiere entonces al uso de técnicas de la lógica y de la matemática discreta para especificar, diseñar, verificar, desarrollar y validar programas. La palabra "formal" se deriva de la lógica formal, ciencia que estudia el razonamiento desde el análisis formal -de acuerdo con su validez o no validez-, y omite el contenido empírico del razonamiento para considerar sólo la forma -estructura sin materia-. Los métodos formales más



rigurosos aplican estas técnicas para comprobar los argumentos utilizados para justificar los requisitos, u otros aspectos de diseño e implementación de un sistema complejo. En la lógica formal como en los métodos formales el objetivo es el mismo, "reducir la dependencia de la intuición y el juicio humanos para evaluar argumentos". Los métodos menos rigurosos enfatizan en la formalización y renuncian a la computación, definición que implica un amplio espectro de técnicas, y una gama igualmente amplia de estrategias. La interacción de las técnicas y estrategias de muchos métodos formales, en determinados proyectos, se limita por el papel que interpretan y por los recursos disponibles para su aplicación.

## Fundamentos matemáticos de los métodos formales

La matemática es una ciencia de la cual se pueden aprovechar muchas de sus propiedades para el desarrollo de los grandes y complejos sistemas actuales, en los que, idealmente, los ingenieros deberían estructurar su ciclo de vida de la misma forma que un matemático se dedica a la matemática aplicada: "presentar la especificación matemática del sistema y elaborar una solución con base a una arquitectura de software que haga posible su implementación". Aunque la especificación matemática de sistemas no es tan concisa como las expresiones matemáticas simples, dado que los sistemas software son mucho más complejos, no sería realista pensar que es posible especificarlos de la misma forma matemática. En todo caso, la ventaja de utilizar matemáticas en la Ingeniería de Software radica en que "suaviza" la transición entre sus actividades, ya que es posible expresar la especificación funcional, el diseño y el código de un programa, mediante notaciones. Esto se logra gracias a que la propiedad fundamental de la matemática es la abstracción, una excelente herramienta para modelar debido a que es exacta, y a que ofrece pocas probabilidades de ambigüedad, lo que permite verificar matemáticamente las especificaciones, y buscar contradicciones e incompletitudes.

Con las matemáticas es posible representar, de forma organizada, los niveles de abstracción en la especificación del sistema; es una buena herramienta para modelar, ya que facilita el diseño del esquema principal de la especificación; permite a analistas e ingenieros verificar la funcionalidad de esa especificación y a los diseñadores ver los detalles suficientes para llevar a cabo su tarea desde las propiedades del modelo. También proporciona un nivel elevado de verificación, ya que para probar que el diseño se ajusta a la especificación y que el código refleja exactamente el diseño, se puede utilizar una demostración matemática. Como se expresó antes, el término formal se caracteriza por utilizar una serie de categorías de modelos o formalismos matemáticos, estos modelos no sólo incluyen los fundamentos matemáticos, también a los procesos para la



producción de software. Entre estos fundamentos básicos se cuentan nociones como lenguajes formales, sintaxis, semántica, modelos, gramática, teorías, especificación, verificación, validación y pruebas; temáticas a las que deberían tener acceso los Ingenieros de Software, para contar con bases sólidas al momento de seleccionar un determinado formalismo, desde los cuales es posible, de forma más acertada, tener los primeros acercamientos a conceptos como requisitos, especificación formal e informal, validación, validez, completitud, correctitud y nociones sobre pruebas del software.

## Formalismo Matemático

Los formalismos matemáticos de los métodos formales son:

I. Lógica de primer orden y teoría de conjuntos. Se utilizan para especificar el sistema mediante estados y operaciones, de tal forma que los datos y sus relaciones se describan detalladamente, sus propiedades se expresen en lógica de primer orden, y la semántica del lenguaje tenga como base la teoría de conjuntos. En el diseño y la implementación del sistema los elementos, descritos matemáticamente, se pueden modificar, pero siempre conservarán las características con las que se especificaron inicialmente. En esta lógica se establece, mediante diversos sistemas de deducción, la definición del lenguaje empleado -lógica de predicados de primer orden-, y de la prueba. Tanto las reglas de inferencia, el diseño de las pruebas, como el estudio de las teorías de igualdad e inducción, representan aspectos claves de este formalismo.

II. Algebraicos y de especificación ecuacional. Describen las estructuras de datos de forma abstracta para establecer el nombre de los conjuntos de datos, sus funciones básicas y propiedades - mediante fórmulas ecuacionales-, en las que no existe el concepto de estado modificable en el tiempo. Tanto el cálculo ecuacional -pruebas mediante ecuaciones-, como los sistemas de reescritura, constituyen el soporte para realizar las deducciones necesarias. También se utilizan para especificar sistemas de información a gran escala, tarea en la que es necesario estudiar los mecanismos de extensión, de parametrización y de composición de las especificaciones (Ehrig & Mahr, 1990).

III. Redes de Petri (Murata, 1989), (Reising, 1991), (Manna & Pnueli, 1992), (Bause & Kritzinger, 2002), (Desel & Esparza, 2005), (Juhás et al., 2007). Establecen el concepto de estado del sistema mediante lugares que pueden contener marcas, y hacen uso de un conjunto de transiciones -con pre y post-condiciones-, para describir cómo evoluciona el sistema, y cómo produce marcas en los puntos de la red. Utilizan características semánticas de entrelazado o de base en la concurrencia real. En la



primera, los procesos paralelos no deben ejecutar las instrucciones al mismo tiempo, mientras que en la segunda existe esa posibilidad. Estas caracterizaciones son fundamentales para poder representar conceptos como estado y transición en los sistemas.

IV. Lógica temporal (Manna & Pnueli, 1992). Se utiliza para describir los sistemas concurrentes y reactivos, poseen una amplia noción de tiempo y estado. Sus especificaciones describen las secuencias válidas de estados -incluyendo los concurrentes- en un sistema específico. Para aplicar este formalismo es necesario establecer inicialmente una clasificación de los diferentes sistemas de lógica temporal de acuerdo con los criterios de proposicionalidad vs. primer orden, tiempo lineal vs. tiempo ramificado, de evaluación instantánea o por intervalos, y tiempo discreto vs. tiempo continuo. Una vez que se establece la formalización del tiempo es posible estudiar la aplicación del formalismo.

Los métodos formales proporcionan un medio para examinar simbólica y completamente el diseño digital –sea hardware o software-, y establecer las propiedades de correctitud o seguridad. Sin embargo, en la práctica raramente se logra -excepto en los componentes de seguridad de sistemas críticos-, debido a la enorme complejidad de los sistemas reales. Se ha experimentado con los métodos formales en varios proyectos para lograr el desarrollo de sistemas reales con calidad:

- Para automatizar la verificación tanto como sea posible (Wunram, 1990).
- En requisitos y diseños de alto nivel, en los que la mayor parte de los detalles se abstraen de diversas formas (Rushby, 1993).
- Sólo a los componentes más críticos (Lutz & Ampo, 1994).
- Para analizar modelos de software y hardware, en los que las variables se discretizan y los rangos se reducen drásticamente (McDermid, 1995).
- En el análisis de los modelos de sistemas de manera jerárquica, de tal forma que se aplique el "divide y vencerás" (Zhang, 1999).

Aunque el uso de la lógica matemática sea un tema de unificación en los métodos formales, no es posible indicar que un método sea mejor que otro; cada dominio de aplicación requiere métodos de modelado diferentes, lo mismo que el acercamiento a las pruebas. Además, en cada dominio en particular, las diferentes fases del ciclo de vida se pueden lograr mejor con la aplicación combinada de diferentes técnicas y herramientas formales (Pressman, 2004).

## Ventajas de los métodos formales

La utilidad de los métodos formales en la Ingeniería de Software es un tema que se debate desde hace varias décadas. Recientemente, con el surgimiento de la



Ingeniería de Conocimiento en la Sociedad de la Información, y la aplicación de los métodos formales en los procesos industriales, nuevamente surge el debate. A continuación se detallan algunas de las ventajas de utilizarlos.

Especificación.

Uno de los problemas más ampliamente reconocido en el desarrollo de software es la dificultad para especificar claramente el comportamiento que se espera del software, problema que se agudiza con el actual desarrollo basado en componentes, ya que el ingeniero tiene sólo una descripción textual de los requisitos, procedimientos, entradas permitidas y salidas esperadas. Asegurar que este tipo de software sea seguro es un problema, no sólo por su tamaño y complejidad, sino porque el código fuente no suele estar disponible para los componentes que se adquieren. Una forma de ofrecer "garantía rigurosa" del producto es definir, con precisión, el comportamiento esperado del software (NASA, 1995). La especificación formal proporciona mayor precisión en el desarrollo de software, y los métodos formales brindan las herramientas que pueden incrementar la garantía buscada. Desarrollar formalmente una especificación requiere conocimiento detallado y preciso del sistema, lo que ayuda a exponer errores y omisiones, y por lo que la mayor ventaja de los métodos formales se da en el desarrollo de la especificación (Clarke & Wing, 1996). En la formalización de la descripción del sistema se detectan ambigüedades y omisiones, y una especificación formal puede mejorar la comunicación entre ingenieros y clientes. Los métodos formales se desarrollaron principalmente para permitir un mejor razonamiento acerca de los sistemas y el software. Luego de diseñar una especificación formal, se puede analizar, manipular y razonar sobre ella, de la misma forma que sobre cualquiera expresión matemática. Una diferencia significativa entre una especificación formal de software y una expresión matemática de álgebra o cálculo, es que típicamente es mucho más grande -a menudo cientos o miles de líneas-. Para tratar con el tamaño y la complejidad de estas expresiones se desarrollan herramientas de software que se pueden agrupar en dos grandes categorías: probadoras de teoremas y verificadoras de modelos (Anderson et al, 1998). Las primeras ayudan al usuario a diseñar pruebas, generalmente para demostrar que las especificaciones cumplen con las propiedades deseadas. Para usarlas, es necesario que el ingeniero tenga cierto grado de habilidad y conocimiento, pero pueden manipular especificaciones muy grandes con propiedades complejas. Los desarrollos recientes en métodos formales introdujeron las verificadoras de modelos, que exploran, hasta cierto punto, todas las posibles ejecuciones del programa especificado. Pueden verificar si cumple con una propiedad específica mediante la exploración de todas las posibles ejecuciones, o producen



contraejemplos en los que la propiedad no se cumple. Aunque estas herramientas pueden ser completamente automáticas, no pueden resolver problemas tan grandes o variados como las probadoras de teoremas. Las herramientas más sofisticadas combinan aspectos de ambas, aplican las verificadoras a algunas partes de la especificación, pero confían en el usuario para probar las propiedades complicadas.

**Verificación**.

Para asegurar la calidad de los sistemas es necesario probarlos, y para asegurar que se desarrollan pruebas rigurosas se requiere una precisa y completa descripción de sus funciones, incluso cuando en la especificación se utilicen métodos formales. Una de las aplicaciones más interesantes de éstos es el desarrollo de herramientas, que pueden generar casos de prueba completos desde la especificación formal. Aunque gran número de herramientas para automatizar pruebas se encuentran disponibles en el mercado, la mayoría automatiza sólo sus aspectos más simples: generan los datos de prueba, ingresan esos datos al sistema y reportan resultados. Definir la respuesta correcta del sistema, para un determinado conjunto de datos de entrada, es una tarea ardua, que la mayoría de herramientas no puede lograr cuando el comportamiento del mismo se especifica en lenguaje natural. Debido a que la respuesta esperada del sistema se puede determinar únicamente mediante la lectura de la especificación, los ingenieros esperan que a las pruebas automatizadas se les adicione este faltante y crítico componente (Dan & Aichernig, 2002). La gran ventaja de las herramientas, que generan pruebas con base en los métodos formales, es que la especificación formal describe matemáticamente el comportamiento del sistema, desde la que se puede generar la respuesta a un dato de entrada en particular, es decir, la herramienta puede generar casos de prueba completos. Las técnicas de verificación formal dependen de especificaciones matemáticamente precisas y, desde un punto de vista costo-beneficio, generar pruebas desde la especificación puede ser uno de los usos más productivos de los métodos formales. Aproximadamente la mitad del tiempo del equipo de trabajo, en un desarrollo típico de software comercial, se invierte en esfuerzo para desarrollar pruebas, e "incluso con este nivel de esfuerzo sólo se eliminan los errores más evidentes" (Saiedian & Hinchey, 1996). Algunas mediciones empíricas demuestran que las pruebas, generadas con herramientas automatizadas, ofrecen una cobertura tan buena o mejor que la alcanzada por las manuales, por lo que los ingenieros pueden elegir entre producir más pruebas en el mismo tiempo, o reducir el número de horas necesarias para hacerlas (Gabbar, 2006).

**Validación.**



Mientras que la verificación se puede realizar semiautomáticamente y las pruebas mecánicamente, la validación es un problema diferente. Una diferencia específica entre verificación y validación es que la primera responde a si "se está construyendo el producto correctamente", y la segunda a si "se está construyendo el producto correcto" (Dasso & Funes, 2007). En otras palabras, la verificación es el conjunto de actividades que aseguran que el software implementa correctamente una función específica, y la validación es un conjunto de actividades diferentes que aseguran que el software construido corresponde con los requisitos del cliente (Amman & Offutt, 2008). Desde el conjunto de requisitos es posible verificar, formal o informalmente, si el sistema los implementa; sin embargo, la validación es necesariamente un proceso informal. Sólo el juicio humano puede determinar si el sistema que se especificó y desarrolló es el adecuado para el trabajo. A pesar de la necesidad de utilizar este juicio en el proceso de validación, los métodos formales tienen su lugar, especialmente en grandes y complejas aplicaciones, como en modelado y simulación. Una de sus aplicaciones más prometedoras es en el modelado de requisitos, ya que, al diseñarlos formalmente, el teorema provisto en la herramienta de prueba se puede utilizar para explorar sus propiedades, y a menudo detectar los conflictos entre ellos. Este método no sustituye al juicio humano, pero puede ayudar a determinar si se especificó el "sistema correcto", por lo que es más fácil determinar si las propiedades deseadas se mantienen (Flynn & Hamlet, 2006). Una diferencia significativa entre validar sistemas de modelado y simulación, y los de control o cálculo, es que los primeros tienen dos tipos de requisitos de validación: deben modelar y predecir el comportamiento de alguna entidad del mundo real, problema que se conoce como "validación operacional", y deben "validar el modelo conceptual", para asegurar que la hipótesis en la que se sustenta es correcta, y que su lógica y estructura son adecuadas para el modelo que se propone (Sargent, 1999). Debido a que el modelo conceptual describe lo que debe representar la simulación, es necesario incluir supuestos acerca del sistema, su entorno, las ecuaciones, los algoritmos, los datos, y de las relaciones entre las entidades del modelo. Aunque los algoritmos y las ecuaciones son declaraciones necesariamente formales, los supuestos y las relaciones se describen normalmente en lenguaje natural, lo que introduce potenciales ambigüedades e incomprensiones entre ingenieros y usuarios. Una tendencia relativamente reciente en los métodos formales, conocida como "métodos formales ligeros" (Jackson, 2001), demuestra tener potencial para detectar errores importantes en la declaración de requisitos, sin el costo de una verificación diseñada formalmente. La premisa básica de este enfoque es el uso de técnicas formales en el análisis de los supuestos, las relaciones y las propiedades de los requisitos, indicadas en su declaración o en



el modelo conceptual. Se puede aplicar a especificaciones parciales o a un segmento de la especificación completa, proceso que se realiza en tres fases: 1) reafirmar los requisitos y el modelo conceptual en una notación formal -o semiformal-, típicamente en una tabla de descripción de estados; 2) identificar y corregir las ambigüedades, conflictos e inconsistencias; y 3) utilizar un verificador de modelos o un probador de teoremas para estudiar el comportamiento del sistema, demostrar sus propiedades y graficar su comportamiento. Los ingenieros y usuarios pueden utilizar estos resultados para mejorar el modelo conceptual (Davis, 2005). Un aspecto particularmente interesante de este enfoque es que se ha utilizado para modelar y analizar el comportamiento del software, del hardware y de las acciones humanas en los sistemas (Mazzola et al, 2006).

Agerholm y Larsen (1997) describen su aplicación en un sistema de actividad extra-vehicular de la NASA; y Lutz (1997) describe la validación de requisitos de los monitores de errores a bordo de una nave espacial. Un detalle importante de este proyecto es que los ingenieros utilizaron el modelo de requisitos para un segundo proyecto, que se desarrolló a partir del primero, como una construcción en serie. Janssen et al (1999) describen la aplicación de un verificador de modelos para analizar procesos de negocios automatizados, como el procesamiento de reclamaciones de seguros.

## ¿Cuál es el costo de añadir el uso de métodos formales a los prácticas tradicionales?

Enrique Vázquez y Miguel Pérez Cer viño e Tumbeiro (2008) aplican dos métodos de desarrollo: el convencional y utilizando métodos formales en un caso práctico con el objetivo de hacer una comparación en el esfuerzo requerido y muestran también la calidad del software desarrollado con las dos metodologías. El software utilizado es un Sistema de Seguridad Compilable y tiene como funcionalidad: grabación de video digital, control automático de puertas, detección de intrusos, supervisión de guardias. Para este caso práctico, se utiliza VDM-SL Plat e Larsen (1992) para el desarrollo de un módulo del sistema y se hace partiendo de los mismos requisitos que en el desarrollo convencional. Se utiliza también IFDA con soporte de VDM-SL y C++. Entre los principales beneficios se encuentra la reducción del código de 12000 líneas del código original a 5500 líneas. La especificación fue introducida a IFDA, un generador de código C++ sobre la plataforma Solaris. Al probar la calidad del software, se hicieron de una a siete pruebas para cada requisito. La calidad fue medida en función de qué tanto se cumplen los requisitos. El software desarrollado con el método formal cumplió en un 82% los requisitos, mientras que con el método convencional, se cumplió sólo el 62%. El esfuerzo se midió en función del tiempo que se llevó para el desarrollo de ese módulo. Con el método convencional se



ocuparon 8 semanas para la obtención de los requisitos y 20 semanas para el diseño e implementación, haciendo un total de 28 semanas. En cambio, con el método formal se llevaron 18 semanas para la especificación abstracta, 5 semanas para la especificación ejecutable, 5 semanas para la validación y media semana para la generación de código en C++, haciendo un total de 28 semanas y media.

## Tendencia de los métodos formales

La industria del software tiene una larga y bien ganada reputación de no cumplir sus promesas y, a pesar de más de 60 años de progreso, tiene años -incluso décadas- por debajo de la madurez necesaria que requiere para satisfacer las necesidades de la naciente Sociedad del Conocimiento.

Es claro que aún no es posible, con la tecnología actual, asegurar el éxito de los proyectos de software, y que para proyectos grandes y complejos el enfoque ad hoc ha demostrado ser insuficiente. La falta de formalización en los puntos clave de la Ingeniería de Software la hace sensible a problemas que son inevitables en actividades altamente técnicas y detalladas como la creación de software. Las buenas prácticas en Ingeniería deben aplicarse en todo el proceso del desarrollo de sistemas, pero, aunque el desarrollo tecnológico aporta mucho material para alcanzarlo, todavía no se logra este objetivo. Incrementar la precisión y el control riguroso es esencial, y es el principal objetivo de los métodos formales, que utilizan esencialmente formalismos lógicos buscando mejorar el software y el hardware, en áreas como confiabilidad, seguridad, productividad y reutilización. Los ejes principales de su accionar son la verificación del código y el diseño, así como la generación de programas y casos de prueba desde las especificaciones. Los métodos formales deberían estar presentes como principios esenciales de las técnicas de prueba. Gaudel (1995) lo estableció como un tema importante de investigación; Hoare (2002a) describió el uso de aserciones formales no para probar el programa, sino para diseñar las pruebas; y Hierons et al. (2008) desarrollaron una investigación en aspectos formales de las pruebas. Los métodos formales se utilizan en el mantenimiento del software (Younger et al., 1996) y en su evolución (Ward & Bennett, 1995), y tal vez su más amplia aplicación sea en el mantenimiento de código heredado (Hoare, 2002b). Los métodos formales son un área de investigación muy activa, y se espera que cada día se incrementen las colaboraciones.

## Corolario

Los métodos formales son técnicas matemáticas, a menudo soportadas por herramientas, para el desarrollo de sistemas software y hardware. Su rigor matemático le permite a los ingenieros analizar y verificar sus modelos en



cualquier parte del ciclo de vida del desarrollo; y dado que la fase más importante en estos procesos es la Ingeniería de Requisitos, son útiles para elicitarlos, articularlos, representarlos y especificarlos. Sus herramientas proporcionan el soporte automatizado necesario para la integridad, trazabilidad, verificabilidad, reutilización, y para apoyar la evolución de los requisitos, los puntos de vista diversos y la gestión de las inconsistencias. Los métodos formales se utilizan en la especificación de software, y se emplean para desarrollar una declaración precisa de lo que el software tiene que hacer, evitando al mismo tiempo las restricciones del cómo se quiere lograr. La especificación es un contrato técnico entre el ingeniero y el cliente, que proporciona un entendimiento común de la finalidad del software; el cliente la utiliza para orientar la aplicación del software, y el ingeniero para guiar su construcción.

Los sistemas de software complejos requieren una cuidadosa organización de la estructura arquitectónica de sus componentes, un modelo del sistema que suprima detalles de la implementación, y que permita al arquitecto concentrarse en los análisis y decisiones más importantes para estructurar el sistema y satisfacer sus requisitos  son ejemplos de lenguajes de descripción arquitectónica basados en la formalización del comportamiento abstracto de los componentes y conectores arquitectónicos.

En la fase de implementación, los métodos formales se utilizan para verificar el código, ya que toda especificación tiene explícito un teorema de correctitud con el que, si se cumplen ciertas condiciones, el programa deberá conseguir el resultado descrito en la documentación. La verificación del código es un intento por demostrar este teorema, o al menos de encontrar por qué falla.

La capacidad para generar casos de prueba completos desde la especificación formal, representa un ahorro sustancial a pesar del costo de su desarrollo. La experiencia demuestra que las técnicas formales se pueden aplicar productivamente, incluso en las pruebas más completas. El proceso para desarrollar una especificación es la fase más importante de la verificación formal, y el enfoque de los "métodos formales ligeros" permite analizar formalmente las especificaciones parciales, y la definición temprana de requisitos.

**Ingeniería del Software I – 2021 Fa.Ce.Na – U.N.N.E.**

## Introduccion al Lenguaje Z

El Lenguaje Z es un lenguaje utilizado en Ingenieria del software para la especificación formal de un sistema de computo, como una fase previa al desarrollo del codigo de programa para el mismo en un lenguaje de programacion.
Desarrollado por Jean-Raymond Abrial mientras formaba parte del Grupo de investigacion en Programacion del Laboratorio de computacion de la Universidad de Oxford.
El lenguaje Z se basa en la teoria de conjuntos, el calculo lambda y la logica de primer orden.
Se definen construcciones denominadas esquemas para describir el espacio de estados del sistema y las operaciones que sobre el mismo se efectuan. En los esquemas se declaran variables y predicados que afectan los valores de las variables declaradas.

Ej. Sistema Cumpleanos de las personas

Necesitamos conocer el nombre de las personas y su fecha de cumpleaños

- Tipos basicos de datos:
  - [NOMBRE, FECHA]
- Primero: Describir su "espacio de estado " (esquema)

$$[NOMBRE, FECHA]$$

*AgendaCumple*

$$contactos : \mathbb{P}\ NOMBRE$$
$$cumple : NOMBRE \pfun FECHA$$
$$\hline$$
$$contactos = \mathrm{dom}\ cumple$$

Un posible estado:

Tres personas en el conjunto nombres conocidos (***known)***, con sus respectivas fechas de cumpleaños registrados por la función ***cumpleaños (birthday):***



$$known = \{ \text{John}, \text{Mike}, \text{Susan} \}$$

$$birthday = \{ \text{John} \mapsto 25\text{–Mar}, \ \text{Mike} \mapsto 20\text{–Dec}, \ \text{Susan} \mapsto 20\text{–Dec} \}.$$

El invariante se satisface, porque *birthday* registra una fecha de exactamente de los tres nombres contenidos en nombres conocidos *known*.

Para AgregarCumpleaños, el esquema seria:

*AgregarCumpleaños*

$\Delta AgendaCumple$

$nombre? : NOMBRE$

$fecha? : FECHA$

---

$nobre? \notin conocidos$

$cumpleaños' == cumpleaños \cup \{ nombre \pfun fecha \}$

- Una coleccion de objetos similares es llamado un *conjunto.*

*Ejemplos:*

- La coleccion de personas que inscribieron el curso de M.F.
- La coleccion de anos bisiestos desde 1986.

Si un conjunto tienen muchos elementos, le damos un nombre que lo describa.

Por ejemplo:

- *PERSONA – Conjunto de todas las personas*
- *ANOBISIESTO – Conjunto de todos los anos bisiestos pasados*

Regla de la sintaxis para los nombres de conjunto.

- Son escrito con mayuscula
- Son singulares



- No contiene espacios, underscores, puntos, tec.

Algunos conjuntos estandares.

- **Z El conjunto de todos los numeros enteros (no** confundir con la notacion Z)
- **N El conjunto de todos los numeros naturales**
- **Ø El conjunto vacio**

Precisiones a tener presente:

- **Ø es subconjunto de cualquier conjunto**
- **N es subconjunto de Z**

Algunos cuidados a tener siempre presente en la notación "Z"

- El conjunto de todos los elementos de $A$ que satisfacen la condición $P(x)$
  - $\{x : A \mid P(x)\}$
  - Nótese que:
    - $x \in A \equiv x : A$
    - La $\mid$ es usada para indicar un predicado
- Z utiliza la "builder set notation"
  - $\{x : A \mid P(x) \bullet F(x)\}$
  - Conjunto de todos los valores $F(x)$ donde $x$ pertenece a $A$ y $P(x)$ es verdad.
- La teoría de conjuntos de Z, se basa en la axomatización de
- Zermelo-Fraenkel

## Tipos Basicos. Introduccion

Para describir el ambito en el cual trabajamos, clasificamos los objetos en conjuntos llamados tipos.



El unico tipo incorporado como parte de la notacion Z, es el conjunto de los enteros, Z.

Todo otro tipo es definido por nosotros.

Para definir un *tipo basico debemos centrarnos en lo* esencial y los detalles los dejamos de lado. Esta focalizacion se denomina *abstraccion.*

Ej.: Una persona tiene un nombre, una fecha de nacimiento y una direccion. Definimos los tipos:

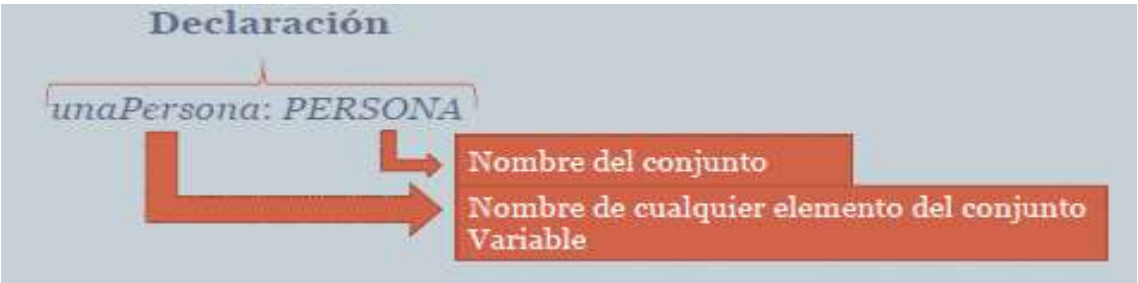
- [DIRECCION, FECHA, NOMBRE]

Entre parentesis cuadrados y en orden alfabetico

Se deben describir claramente. Por ejemplo; NOMBRE es el conjunto de todos los posibles nombre que cualquier persona pueda tener.

**Declaraciones**

Sea *PERSONA el conjunto de todas las posibles* personas sobre el Universo. Si queremos referirnos a alguna en particular escribimos:




Una variable tiene un nombre (*unaPersona), un* tipo (*PERSONA) y un valor extraido desde el tipo.*

Considere la siguiente declaracion:

- *unaEdadPersona: 0..130*

El nombre de la variable es "unaEdadPersona", su tipo es **Z,** debido que los valores que puede tomar la variables son todos elementos de **Z.**

Observacion: No puede hacer una declaracion si su tipo no ha sido definido.



## Nombres de Variables

Debemos usar nombres descriptivos para denominar variables.

Por ejemplo:

- *unaEdadPersona: 0..130 OK*
- *a: 0..130 No*

Convencion:

- Siempre comenzar con una letra minuscula.
- Escribir como una combinacion de letras mayusculas y minusculas, no usar solo minusculas.

No debe contener espacios en blanco, underscope, caracteres especiales

## El Tipo Entero

- El tipo Z es el unico tipo incorporado en Z, con valores
  - …-2, -1, 0, 1, 2, ….
- Operadores aritmeticos: +, -, *, *div y mod*
- Funciones: *max y min*
- Operadores relacionales: >, <, ≥ y ≤

## Precedencias en tipo entero

Precedencia es el orden en el cual las operaciones son llevadas a cabo.

En aritmetica:

1) Parentesis la mas alta prioridad

2) *, div, y mod en el orden escrito de izquierda a derecha

3) + y – en el orden escrito de izquierda a derecha

## Predicados

Los valores de las variables pueden ser restringidos por ***predicados y al satisfacerse un predicado se define un*** conjunto.

Los tipos de predicados son:



= ,∈: igualdad y pertenencia

> ,≥,<,≤ : relaciones

∧,∨ : conectivos logicos

Descripcion del nuevo conjunto: { declaracion | predicado}

Ej.:

$$\{n : \mathbb{Z} \mid n > 0\}$$
$$\{n : 1..9 \mid n \bmod 2 = 0\}$$
$$\{n : \mathbb{Z} \mid n \geq 0 \land m > 2 \land n \bmod 2 \neq 0\}$$

## Esquemas (Schemas)

Un esquema representa el estado de un sistema. Una coleccion de esquemas representa el comportamiento de un sistema computacional.

Sistema: Desarrollar un simple *contador.*

Estado

El contenido de la memoria del sistema es su ***estado.***

- Valor actual del contador
- Valor maximo que puede alcanzar

Estado inicial es cuando el contador esta en cero, maximo es 9999

- *Estadoinicial: contador = 0, maximo = 9999*

Podriamos tener un estado intermedio cuando el contador esta entre cero y el valor maximo

- *Estadointermedio: contador = 147, maximo =9999*

Podriamos tener un estado final cuando el contador alcanza su valor maximo y no puede seguir avanzando

- *Estadofinal: contador = 9999, maximo = 9999*

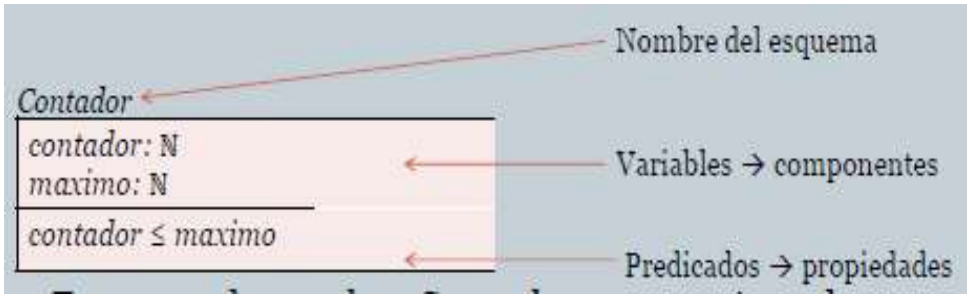


Contador y maximo son ambos numero. Podriamos representarlos como entero del tipo $\mathbb{Z}$, pero permitiriamos valores negativos para *contador y para maximo. De modo que introducimos* $\mathbb{N}$ *como una* abreviacion para el conjunto de todos los numero enteros excluyendo los valores negativos

- Declarando $\mathbb{N}$ de esta forma podemos usar los operadores <, > y =. El tipo de $\mathbb{N}$ es $\mathbb{Z}$.
- $\mathbb{N} == \{n : \mathbb{Z} \mid n \geq 0\}$

El simbolo == se conoce como el simbolo de defincicion.

Nos quedaría entonces el siguiente esquema del estado del sistema:




Un *esquema es un conjunto de variables junto con un* conjunto de predicados que limitan estas variables. Se dibuja como una caja abierta.

Esquema de nombre *Contador que contiene dos* variables, *contador y maximo, y un predicado "contador ≤ maximo".*

El nombre de un esquema debe comenzar con mayuscula y no puede contener espacios en blanco.

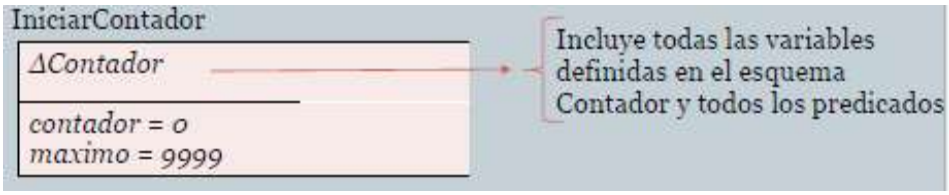
Si existiera mas de un predicado, estos se acoplan por medio de la conjuncion, actua por defecto.

El esquema en que se describe las variables de un sistema y los predicados de estas variables se denomina ***esquema del estado del sistema.***

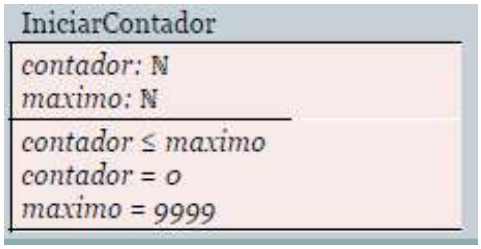
Este representa el estado de un objeto, la coleccion de valores almacenados en las variables en un instante de tiempo. Sus variables son llamadas variables de estado.



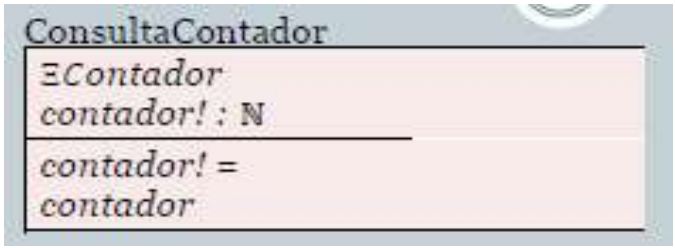
Podemos , ahora, definir el esquema IniciarContador que describe el estado inicial de Contador.




Que es equivalente a escribir:




Si tuviéramos que escribir el esquema para Una consulta al Sistema:



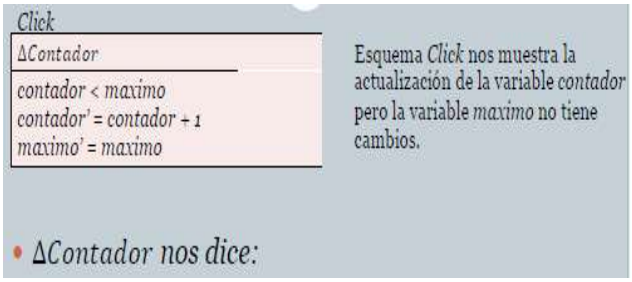

La letra griega "Xi" contador incluye todas las variables y predicados definidos en el esquema Contador; los valores almacenados en estas variables no cambiaran.

La declaracion ***contador!:*** ℕ *que el valor de salida de* contador pertenece al conjunto ℕ. Usamos el simbolo (***!)*** para la salida.

El predicado ***contador! = contador*** *nos dice que la* salida del contador es la misma que contador.

Si tuiveramos que realizar alguna funcionalidad que produjera un cambio en el sistema, el esquema seria:



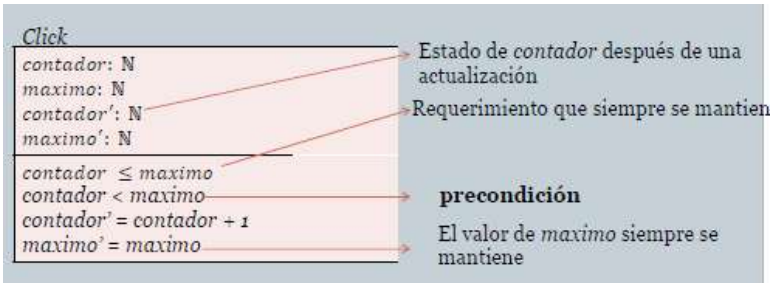



Incluye todas las variables y predicados definidos en el esquema Contador.

Los valores almacenados en alguna (o todas) de estas variables pueden cambiar.

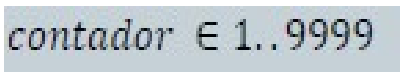
Toda variable seguida por el signo ' representa el estado despues de que una actualizacion se ha realizado.

**Cambio en el estado del sistema. Esquema equivalente**




Una precondición de una operacion en un esquema describe el conjunto de estados para los cuales la salida de la operacion es definida.

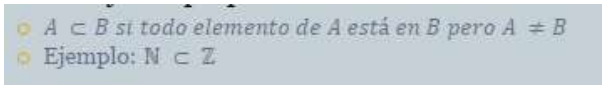
En nuestro caso, en el esquema "Click", el conjunto de estados para los cuales la salida esta definida es:




**Tipo Conjunto**

Una variable de un tipo conjunto es un conjunto y es usada para representar colecciones de elementos tales como una clase de estudiantes o un conjunto de habitaciones de hotel.

Subconjunto propio:





Subconjunto: $A \subseteq B$ si todo elemento de A está en B

Ejemplo en Z. Si tenemos la siguiente declaracion:

$x: \mathbb{F}\{1, 2, 3\}$; entonces
$x \in \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$

Declaraciones Tipo Conjunto

Ejemplo: grupos de alumnos estudian una serie de materias.

Tipos:

*ALUMNO* – El conjunto de todos los alumnos
*MATERIA* – El conjunto de todas las materias

Podriamos definir:

grupoAlumnos: $\mathbb{F}ALUMNO$
grupoMaterias: $\mathbb{F}MATERIA$

$\mathbb{F}$ ➔ Significa que grupoAlumnos es un subconjunto de grupo ALUMNO.

**Operaciones entre Conjuntos**

- Union → ∪
  - $\{1, 2\} \cup \{3 \cup 4\} = \{\{1, 2, 3, 4\}$
- Intersección → ∩
  - $\{1, 2, 3\} \cap \{3, 4\} = \{3\}$
- Diferencia → \
  - $\{1, 2, 3, 4\} \setminus \{2, 4\} = \{1, 3\}$
- Precedencia
  - Paréntesis
  - Intersección
  - Unión y diferencia
- Ejemplo:
  - $\{2, 4\} \cup (\{1, 2, 3\}/\{3, 4\}) \cap \{1, 3\} = ?$

**Calculo de Esquemas**



Anteriormente hemos visto como declaraciones y predicados fueron combinados en estructura llamadas esquemas. Vimos como un esquema puede representar el estado de un sistema, que un esquema puede cambiar el estado del sistema y una colección de esquemas modela el ambito de un sistema.

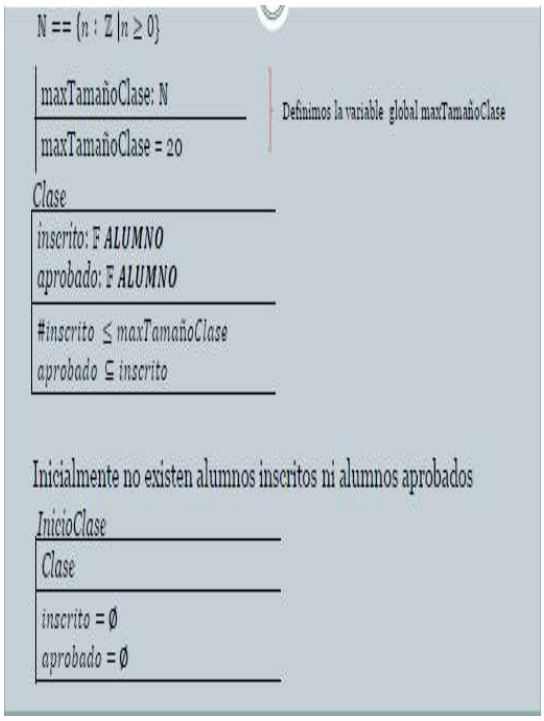
Veremos como esquemas grandes pueden ser formados combinando esquemas mas pequenos usando conjuncion (∧) y la disyuncion ∨ , por medio de un ejemplo.

Ej.: El Sistema Clase

Los estudiantes pueden disfrutar de la clase de lenguaje Z si esta no esta completa. Aquellos que completen exitosamente todas las tareas seran recompensados con un certificado firmado por el profesor.

Tipo basico *ALUMNO dado por el conjunto [ALUMNO]*

Existe un numero limite de alumnos que pueden disfrutar la clase.




Un alumno se puede inscribir en la clase si esta no está completa y si no se ha inscrito con anterioridad. Un nuevo alumno no puede haber aprobado todos sus trabajos.



*BienInscrito*

$\Delta Clase$
$alumno?: \mathbf{ALUMNO}$ → El signo ? significa entrada

$\#inscrito < maxTamañoClase$
$alumno? \in inscrito$
$inscrito' = inscrito \cup \{alumno?\}$
$aprobado' = aprobado$

Un alumno que ya existe es transferido al conjunto *aprobado* si ha Aprobado todos sus trabajos y no haya sido transferido previamente. Todo alumno en el conjunto *aprobado* debe también estar en el conjunto *inscrito*

*BienCompletado*

$\Delta Clase$
$alumno?: \mathbf{ALUMNO}$

$alumno? \in inscrito$
$alumno? \notin aprobado$
$inscrito' = inscrito$
$aprobado' = aprobado \cup \{alumno?\}$

Solamente aquellos alumnos que han aprobado pueden obtener un *certificado.*

*ObtenerCertificado*

$\Delta Clase$
$alumno?: \mathbf{ALUMNO}$

$alumno? \in aprobado$
$inscrito' = inscrito \setminus \{alumno?\}$
$aprobado' = aprobado \setminus \{alumno?\}$

**Algunos aspecto no considerados:**

- La clase esta completa y nadie puede ser inscrito en ella.
- Un alumno puede ser inscrito dos veces.
- Un alumno no inscrito puede ser aprobado

**Definicion de Tipo Libre (FREE)**

Posible solucion a temas pendientes, es decir, a los errores.

Dibujar una tabla de precondicione el conjunto de estados para los cuales las salidas exitosas estan bien definidas. Posteriormente agregamos una columna las condiciones en las cuales se producen errores.



| Esquema | Precondición de éxito | Condición de error |
|---|---|---|
| *BienInscrito* | *#inscrito < maxTamañoClase*<br>*alumno? ∉ inscrito* | Clase completa: *#inscrito ≥ maxTamañoClase*<br>Alumno ya inscrito: *alumno? ∈ inscrito* |
| BienCompletado | *alumno? ∈ inscrito*<br>*alumno? ∈ aprobado* | Alumno no inscrito: *alumno? ∉ inscrito*<br>Alumno ya aprobado: *alumno? ∈ aprobado* |
| ObtenerCertificado | *alumno? ∈ aprobado* | Alumno no aprobado: *alumno ∉ aprobado* |

En una *definicion de tipo libre, definimos REPORT* como el conjunto de todos los valores que describen o bien un esquema exitoso o las razones de su error.

$$REPORT ::= exito \mid claseLlena \mid yaInscrito \mid noInscrito \mid yaAprobado \mid noAprobado$$

El esquema Exito tiene exactamente una declaracion y un predicado. Este sera combinado con otros esquemas para indicar sus salidas exitosas.

*Exito*

$report! : REPORT$

$report! = exito$

Entonces, el esquema de Errores seria:

*ClaseLlena*

$\Xi Clase$
$report! : REPORT$

$\#inscrito \geq maximoTamañoClase$
$report! = claseLlena$

La clase ya está completa

*YaInscrito*

$\Xi Clase$
$alumno? : ALUMNO$
$report! : REPORT$

$alumno? \in inscrito$
$report! = yaInscrito$

Un alumno no puede ser inscrito de nuevo si ya lo estaba

*NoInscrito*

$\Xi Clase$
$alumno? : ALUMNO$
$report! : REPORT$

$alumno? \in inscrito$
$report! = noInscrito$

Si un alumno no está inscrito no puede ser aprobado



*YaAprobado*

| |
|---|
| $\Xi Clase$<br>$alumno? : ALUMNO$<br>$report! : REPORT$ |
| $alumno? \in aprobado$<br>$report! = yaAprobado$ |

Un alumno que está aprobado no puede ser aprobado nuevamente

*NoAprobado*

| |
|---|
| $\Xi Clase$<br>$alumno? : ALUMNO$<br>$report! : REPORT$ |
| $alumno? \in aprobado$<br>$report! = noAprobado$ |

Si un alumno que no está aprobado No puede obtener el certificado

Calculo de Esquema

Usamos la conjuncion (∧), para combinar dos esquemas, y la disyuncion (∨) para representar alternativas.

- $BienInscrito \land Exito$
- $(BienInscrito \land Exito) \lor ClaseLlena \lor YaInscrito$

- $Inscripcion \triangleq (BienInscrito \land Exito) \lor ClaseLlena \lor YaInscrito$
- $Completacion \triangleq (BienCompletado \land Exito) \lor NoInscripto \lor YaAprobado$
- $ObtencionCertificado \triangleq (ObtenerCertificado \land Exito) \lor NoAprobado$

**Relaciones Binarias**

Sean *A y B conjuntos una relacion binaria R es un* subconjunto del conjunto A × B. El conjunto *A se* denomina el dominio de la relacion y *B se denomina el* rango de la relacion.

- $ALUMNO = \{juan, maría, josé\}$
- $CURSO = \{cálculo, álgebra, programación\}$
- $R = \{(juan, cálculo), (juan, álgebra), (maría, álgebra), (josé, programación)\}$ es una relación pues $R \subseteq ALUMNO \times CURSO$



Como una relacion es un conjunto, entonces son validas todas las operaciones sobre conjunto previamente definidas.

**Notacion "maplet"**

- Un elemento de una relación es un par ordenado, por ejemplo (*juan, cálculo*). Este par ordenado puede ser escrito como: $juan \mapsto cálculo$. Esta es conocida como la **notación maplet**, y $juan \mapsto cálculo$ es conocido como un *maplet*.
  - $(juan, cálculo) = juan \mapsto cálculo$
- Usamos la notación maplet para representar un elemento en una relación binaria, usamos la notación de par ordenado para representar un par en particular. Por ejemplo:
  - $parejas = \{juan \mapsto maría, ana \mapsto pedro\}$
  - $unaPareja = (juan, maría)$

**Algunas funciones en Z**

- $first(juan, maría) = juan$
- $second(juan, maría) = maría$

- $dom\{1 \mapsto a, 2 \mapsto b, 3 \mapsto c\} = \{1,2,3\}$ → dominio de la relación
- $ran\{1 \mapsto a, 2 \mapsto b, 3 \mapsto c\} = \{a, b, c\}$ → rango de la relación

**Declaracion de una Relacion Binaria**

Supongamos que hemos definido los tipos del conjunto de todas las fechas y el conjunto d e todas las personas.

$$cita : FECHA \leftrightarrow PERSONA$$
$$cita = \{7nov \mapsto tomás, 7nov \mapsto ana, 8nov \mapsto josé\}$$

El signo doble flecha es el operador relacion binaria

Restricciones al Dominio



- Supongamos que tenemos la relación binaria
  - $\{1 \mapsto a, 2 \mapsto b, 3 \mapsto c\}$
  - Su dominio es {1,2,3}
  - Su rango es $\{a, b, c\}$
- Si queremos restringir la relación de modo que el dominio sea $\{2\}$, tendríamos $\{2 \mapsto b\}$ y escribimos:
  - $\{2\} \lhd \{1 \mapsto a, 2 \mapsto b, 3 \mapsto c\} = \{2 \mapsto b\}$
  - Donde $\lhd$ es el ***operador restricción de dominio***

Restricciones al Rango

- Supongamos que tenemos la relación binaria
  - $\{1 \mapsto a, 2 \mapsto b, 3 \mapsto c\}$
  - Su dominio es {1,2,3}
  - Su rango es $\{a, b, c\}$
- Si queremos restringir la relación de modo que el rango sea $\{b\}$, tendríamos $\{2 \mapsto b\}$ y escribimos:
  - $\{1 \mapsto a, 2 \mapsto b, 3 \mapsto c\} \rhd \{b\} = \{2 \mapsto b\}$
  - Donde $\rhd$ es el ***operador restricción de rango***

Relacion Inversa

- Supongamos que tenemos el tipo *BEBIDA*, el conjunto de todas las bebidas. Definimos ***costos*** como una relación binaria desde las bebidas al precio.

$$costos : BEBIDA \leftrightarrow \mathbb{Z}$$
$$costos = \{te \mapsto 150, café \mapsto 250, chocolate \mapsto 500, sopa \mapsto 600\}$$

- Podemos definir la relación inversa que se define desde el precio hasta las bebidas

$$compras : \mathbb{Z} \leftrightarrow BEBIDA$$
$$compras = \{150 \mapsto té, 200 \mapsto café, 500 \mapsto chocolate, 600 \mapsto sopa\}$$

- Se escribe $costos \sim = compras$

Compuesta



- Supongamos que tenemos los tipos *[PERSONA, OFICINA]*, el conjunto de todas las personas y oficinas respectivamente. Definamos las relaciones binarias:

$$tieneAnexo : PERSONA \rightarrowtail \mathbb{Z}$$

$$tieneAnexo = \{juan \mapsto 317, tomás \mapsto 208, jimy \mapsto 326, maría \mapsto 225\}$$

$$anexoOficina : \mathbb{Z} \rightarrowtail OFICINA$$

$$anexoOficina = \{317 \mapsto A306, 208 \mapsto A39, 326 \mapsto A306, 225 \mapsto A39\}$$

  - Podemos relacionar las personas con sus oficinas partiendo desde *PERSONA* llegamos a *OFICINA* vía $\mathbb{Z}$. Por ejemplo, *juan* tiene el anexo 317 que está en la oficina *A306*, con lo cual concluimos que *juan* tiene la oficina *A306*.

$$tieneAnexo; anexoOfina = \{juan \mapsto A306, tomás \mapsto A39, jimy \mapsto A306, maría \mapsto A39\}$$

## Imagen Relacional

- Supongamos que tenemos la relación binaria
  - $\{1 \mapsto a, 2 \mapsto b, 3 \mapsto c\}$
  - Su dominio es {1,2,3}- Su rango es $\{a, b, c\}$
- Deseamos responder a la pregunta ¿cuál es la segunda coordenada del par cuya primera componente es 2?. La respuesta es $b$, escribimos:
  - $\{1 \mapsto a, 2 \mapsto b, 3 \mapsto c\}[\![\{2\}]\!] = \{b\}$
  - Donde $[\![\ \ ]\!]$ es el ***operador imagen relacional***
- Ejemplo: Si $R = \{(1, a), (2, b), (3, c), (4, d), (5, e)\}$ y $S = 2..4$ evalúe:
  - $R[\![S]\!]$
  - $ran(S \lhd R)$