ESCUELA SUPERIOR DE INGENIERÍA

INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

Framework para la gestión de invariantes dinámicos en *WS-BPEL*

Francisco Javier Santacruz López-Cepero

14 de septiembre de 2011

ESCUELA SUPERIOR DE INGENIERÍA

INGENIERO TÉCNICO EN INFORMÁTICA DE SISTEMAS

Título

- Departamento: Lenguajes y Sistemas Informáticos
- Directores del proyecto: Manuel Palomo Duarte, Antonio García Domínguez
- Autor del proyecto: Francisco Javier Santacruz López-Cepero

Cádiz, 14 de septiembre de 2011

Fdo: Francisco Javier Santacruz López-Cepero

Agradecimientos

- A Manuel Palomo Duarte, por ofrecerme la oportunidad de realizar este proyecto así como su apoyo y orientación.
- A Antonio García Domínguez, por su ayuda, buen hacer y todos sus precisos y eficaces consejos.
- A Pablo Recio Quijano, por su intercambio de conocimientos al inicio de este proyecto.
- A los miembros del grupo de investigación *SPI&FM*, por su colaboración.
- A mi familia y amigos, por su constante apoyo y comprensión en mis horas de trabajo.

Licencia

Notación y formato

A lo largo de este texto, se empleará una notación propia para destacados, listados, citas y notas que se especifica a continuación. Esta notación incluye el uso de cursiva para *Nombres Propios*, así como de tipografía monoespaciada para menciones a literales específicos como nombres de `variable`, `órdenes` o `funciones`.

Las citas literales extraídas de documentación o bibliografía, serán sangradas y se encontrarán firmadas abajo a la derecha, como puede verse en el ejemplo:

> El Nellie, un bergantín de considerable tonelaje, se inclinó hacia el ancla sin una sola vibración de las velas y permaneció inmóvil. El flujo de la marea había terminado, casi no soplaba viento y, como había que seguir río abajo, lo único que quedaba por hacer era detenerse y esperar el cambio de la marea. El estuario del Támesis se prolongaba frente a nosotros como el comienzo de un interminable camino de agua. A lo lejos el cielo y el mar se unían sin ninguna interferencia, y el espacio luminoso de las velas curtidas de los navíos que subían con la marea parecían racimos encendidos de lonas agudamente triangulares, en los que resplandecían las botavaras barnizadas. La bruma que se extendía por las orillas del río se deslizaba hacia el mar y allí se desvanecía suavemente.
>
> Joseph Conrad [Con02]

Son listados incrustaciones de código original dentro del propio texto del documento, añadidos al mismo a modo de ilustración o de ejemplo. Desde el texto, nos referiremos a los listados mencionando su numeración[1], e incluso podremos referenciar líneas concretas en ellos. Por ejemplo, el listado 1, el cual es un simple extracto del guión de instalación automática de *IdiginBPEL*. En la línea 6 podemos ver como las opciones de instalación son `idg`, la cual instala *IdiginBPEL*, `takuan`, que hace lo propio con *Takuan*, y `both` que realiza ambas acciones. En este documento serán empleados listados en lenguaje *Bash*, *Python* y *xml*.

Listado 1: Listado de ejemplo.

```
1
2  if len(sys.argv) > 1 :
3      __INSTALL_OPTS__ = sys.argv[1]
4
5      # Check the option
6      if __INSTALL_OPTS__ not in ('idg', 'takuan', 'both') :
7          log.error('Unkwown option: %s' % __INSTALL_OPTS__)
8          log.info('install.py [idg|takuan|both]')
9          sys.exit()
```

[1] De la misma forma que emplearán figuras, tablas y referencias cruzadas a secciones concretas.

Índice general

Índice de figuras

Índice de tablas

Índice de listados

CAPÍTULO 1

Introducción

Este Proyecto de Fin de Carrera (*PFC*) ha sido realizado en colaboración con el grupo de investigación *SPI&FM* (Software Process Improvement and Formal Methods) en su línea de Ingeniería Web. Esta línea de investigación se encuentra centrada en el desarrollo y perfeccionamiento de tareas de prueba de software para el lenguaje *WS-BPEL* [OAS07]. En concreto, el enfoque principal de la investigación se dirige hacia las pruebas de caja blanca realizadas sobre composiciones en este lenguaje.

Dentro de la sección de investigación del grupo *SPI&FM* dirigida hacia las pruebas de caja blanca, la colaboración se realizó con la rama enfocada a la generación dinámica de invariantes para *WS-BPEL*. Esta técnica se encuentra basada en la extracción de propiedades características de una composición de código ejecutable *WS-BPEL 2.0* a partir del análisis estadístico de su comportamiento durante la ejecución de una batería de pruebas del mismo. Los esfuerzos del grupo en este sentido se materializaron en el desarrollo de la herramienta de código libre *Takuan*, un generador de invariantes dinámicos para el lenguaje *WS-BPEL* [Pal11].

La labor del alumno en colaboración con la investigación en curso del citado grupo, será ayudar al refinado y expansión de la herramienta *Takuan*. La tarea concreta a realizar consiste en la elaboración de una herramienta nueva y externa basada en *Takuan*: *IdiginBPEL*, la cual recoge y amplia de forma gráfica la funcionalidad ya implementada en la anterior. El sistema creado por el grupo, *Takuan*, carece originalmente de interacción gráfica alguna[1] y está basada en el uso de la consola de órdenes. Fue pues necesario repensar el proceso completo de *Takuan*, ampliándolo y adaptándolo necesariamente a las exigencias de una aplicación más flexible que

[1] Si exceptuamos el asistente gráfico realizado como «plugin» para *NetBeans*, del cual existe más información en la sección 2.7 (página 16).

pusiese a disposición del usuario un completo control sobre cada etapa del proceso, sin contar con las facilidades que provee una interfaz de este tipo, y la inclusión de un mejor ajuste sobre las pruebas a realizar.

Surge pues la propuesta de este *PFC*, de la necesidad por parte del grupo de disponer de una herramienta completa, gráfica y de uso sencillo, para la realización de las pruebas de caja blanca basadas en la generación dinámica de invariantes para el lenguaje *WS-BPEL* realizada por *Takuan*. Hasta la fecha, estando implementada la generación de los mismos, la interfaz del sistema se consideraba poco eficaz y flexible, así como no apta para un público más general.

1.1. Objetivos

El propio nombre que recibe el proyecto, *IdiginBPEL*, hace posible pensar en cual es el objetivo de la herramienta; siendo la frase «I-dig-in-BPEL» traducible como *"Cavo o escarbo en BPEL"*, es claro cómo esta pretende proporcionar un método sencillo (*I*, yo, uno mismo puede hacerlo) de acceso a un anális profundo de los programas realizados con este lenguaje. *IdiginBPEL* es así mismo compuesto a partir del acrónimo *IDIGB*, esto es, *Improved Dynamic Invariant Generation for BPEL*, *Generación Mejorada de Invariantes Dinámicos para BPEL*. De su propio nombre pues se desprenden los principales objetivos de este proyecto:

- **Sencillez de uso**: Mediante la creación de una interfaz gráfica, recomposición del proceso en términos más sencillos, abstracción de las operaciones en otras más generales y el desarrollo de nuevas de funcionalidades de apoyo, se pretende mejorar la experiencia de uso de la herramienta *Takuan* y proveer de una herramienta gráfica de operación sencilla y fácil comprensión para no expertos.
- **Mayor potencia**: Al repensar y diseñar una nueva interfaz, así como una nueva manera de realizar el proceso completo de generación de invariantes, ha sido posible dotar al sistema de mayores posibilidades, añadiendo un sistema de proyectos que permite la portabilidad de los estudios realizados, la posibilidad de ejecución en bloque de casos de prueba y proporcionando, interfaz mediante, un control más fino sobre qué, exactamente, entra en ejecución o se realiza un análisis estadístico de búsqueda de invariantes.
- **Eficiencia**: A partir de la abstracción realizada en base al análisis sobre las diversas operaciones que conforman el proceso completo de la búsqueda de invariantes, es posible mejorar la eficiencia de las tareas a realizar, permitiendo al usuario omitir pasos que solo deben realizarse una sola vez por cada composición, y así, realizar una única parte del proceso de forma independiente cuantas veces sea necesario.

Es por tanto *IdiginBPEL* una capa de abstracción por encima de la herramienta de generación dinámica de invariantes *Takuan*, descrita con mayor detalle en el capítulo 2. Esta herramienta, formada por diversas partes y basada en una interfaz de uso de consola de órdenes, es la base

funcional de *IdiginBPEL*, siendo la función principal del proyecto el permitir una reorganización de su proceso, añadiendo posibilidades antes vedadas debido a su ejecución en bloque, y proveyendo de una sencillez de uso mejorada.

Es objetivo de este proyecto eliminar, en lo posible, la necesidad del usuario del sistema de manejar ficheros de texto, ejecutar comandos de consola o manejar proyectos armados manualmente basados en convenciones, así como dotarlo de mayores posibilidades a la hora de controlar el proceso.

1.2. Alcance

Esta sección explica los diferentes resultados obtenidos por la labor del alumno durante el desarrollo del proyecto. El *PFC* deriva pues los siguientes productos:

- Estudio del proceso de *Takuan* y su adaptación a una interfaz de tipo gráfico.
- Paquete de código *Python* el cual compone un envoltorio lógico a *Takuan*, dirigido a proyectos.
- Paquete de código *Python* el cual implementa una capa gráfica de usuario para la interacción del mismo con el proceso de *IdiginBPEL*.
- Ficheros de traducción para la interfaz de la aplicación.
- Documentación adjunta describiendo en detalle el sistema completo.

Se excluyeron de los objetivos a realizar inicialmente planteados la inclusión de un visor de código, el cual presentaba problemas más allá de las intenciones del proyecto y el empleo de *IdiginBPEL* con un servidor *ActiveBPEL* remoto.

1.3. Estructura del documento

Este documento se encuentra organizado en los siguientes capítulos y secciones:

- **Introducción**: Breve resumen y descripción de este PFC.
- **Fundamentos**: En este capítulo queda descrito el sistema de *Takuan*, a partir del cual se basa el desarrollo completo de la herramienta *IdiginBPEL* y este proyecto. La sección incluye una descripción resumida de *Takuan*, su trayectoria, así como cierto detalle sobre su implementación; información necesaria para la comprensión de *IdiginBPEL*.

- **Planificación**: Describe las labores de planificación realizadas, la estrategia que se emplea durante el desarrollo y una relación de los hitos del mismo. También detalla y justifica la elección de las diferentes herramientas y tecnologías a partir de las cuales se realizaron las tareas de desarrollo.

- **Análisis y Diseño**: Dividida en ”Análisis” y ”Diseño”, incluye una descripción y especificación formal de la aplicación basada en *UML 2.0*. Establece y justifica el uso de diferentes recursos y describe la arquitectura general de la herramienta.

- **Implementación**: El capítulo describe de forma detallada la implementación técnica de la herramienta creada, *IdiginBPEL*. En ella se incluye el proceso de desarrollo, las tecnologías empleadas, los problemas encontrados y otros detalles internos.

- **Conclusiones**: Breve resumen de los logros conseguidos, experiencia personal del alumno y posibles líneas abiertas para futuras ampliaciones del desarrollo realizado.

- **Apéndices**: Incluyen información adicional tales como manuales, procedimientos empleados o descripciones de herramientas, que por su aspecto técnico no tienen lugar en el resto del documento.

    - **Herramientas**: Resumen de las herramientas empleadas, manuales realizados por el alumno durante el empleo de las mismas y breves descripciones de técnicas concretas empleadas durante el desarrollo.
    - **Manual**: Manual detallado de uso de *IdiginBPEL*. El manual se encuentra dividido en dos partes: *Guía Introductoria* y *Manual*; siendo la *Guía Introductoria* un resumen sencillo de su uso típico.

CAPÍTULO 2

Fundamentos

2.1. Takuan

La herramienta *Takuan* [Man08] es un *Generador Dinámico de Invariantes para el lenguaje de Web Services WS-BPEL*, útil para la aplicación de pruebas de caja blanca al lenguaje de Servicios Web *WS-BPEL 2.0* [IBM03]. El proyecto, desarrollado en el marco del grupo de Mejora del Proceso Software y Métodos Formales (*SPI&FM* por *Software Process Improvement & Formal Methods* Group [SPI]), permite la generación de invariantes dinámicos a partir de los cuales es posible inferir el comportamiento de un programa dado y comprobar su lógica interna, pudiendo emplearse estos para mejorar la comprensión del programa, verificar de su comportamiento o depurar su código.

Figura 2.1: Logotipo de *Takuan*

Takuan realiza un análisis dinámico de las composiciones y sus procesos, a partir de trazas o registros de ejecución para obtener así un resultado más ajustado, por tratarse de un entorno real, sin las asunciones generales e imprecisiones en las que se podría incurrir empleando técnicas estáticas como redes de Petri y máquinas de estado finito [Mor08]. Esta se encuentra basada en el uso de otros tres sistemas *ActiveBPEL*, *BPELUnit* y *Daikon*. Todos ellos se encuentran bajo licencia libre, así como *Takuan*.

- ***ActiveBPEL***: Motor de ejecución *WS-BPEL 2.0*. Desarrollado por la empresa Active Endpoints [**?**]
- ***BPELUnit***: Biblioteca de pruebas unitarias para *WS-BPEL 2.0* [BPE06].
- ***Daikon***: Generador dinámico de invariantes para *C/C++*, *Perl*, *Java* y otros [Dai00].

Takuan es, por tanto, un sistema compuesto, que integra estas tres herramientas y código propio dentro de su flujo. La arquitectura, e implementación de *Takuan* serán comentadas en mayor profundidad en las Secciones 2.5 y 2.6.

2.2. El lenguaje *WS-BPEL 2.0*

El lenguaje de ejecución de proceso de negocio en servicios web, o «Web Services Business Process Execution Language» (*WS-BPEL*) es un lenguaje usado en la definición de acciones entre procesos de negocio y servicios web. En su momento actual de desarrollo, el lenguaje se encuentra definido como estándar *OASIS* [1] actualmente por su versión 2.0. Enfocado a la programación a gran escala («Programming in the large») [2] empleando el formato *xml* como base para permitir su serialización.

Originalmente, *Microsoft* e *IBM* habían definido de forma paralela dos lenguajes enfocados a la programación a gran escala con grandes similaridades entre ellos, de forma que ambas compañías decidieron unificar y combinar ambos lenguajes en uno nuevo llamado *BPEL4WS*. En 2003, junto con otras compañías, la propuesta de estándar de *BPEL4WS* fue enviada a *OASIS* para su estandarización, siendo aprobada en 2004 bajo el acrónimo *WS-BPEL* versión 1.0, aunque el acrónimo *WS-BPEL* es comúnmente más utilizado.

WS-BPEL es un lenguaje de orquestación; La analogía de orquestación trata de explicar un sistema centralizado dirigido por el ”director de orquestra”, donde se controlan todas las salidas desde un único punto. Este modelo se contrapone a la coreografía, representando un sistema

[1] Organization for the Advancement of Structured Information Standards (OASIS) [Wikd]. Es un consorcio global que dirige el desarrollo, convergencia, y adopción de estándares de e-negocio y servicios web.

[2] «Programming in the large» normalmente se refiere a las interacciones de las transiciones de estado de alto nivel de un proceso *WS-BPEL*. Mientras que «Programming in the small» maneja el comportamiento a bajo nivel del proceso, a menudo ejecutados como una sola instancia o incluyendo acceso a recursos lógicos tales como bases de datos o ficheros [Wika].

distribuido donde cada elemento es independiente e interactúa de forma autónoma con los demás. La principal diferencia entre un lenguaje de orquestación y un lenguaje de coreografía se encuentra en el control de flujo y la ejecución. Una orquestación puede definir intercambio de mensajes controlado, una coreografía especifica protocolo de intercambio de mensajes entre pares, definiendo secuencias de mensajes con el propósito de garantizar la interoperabilidad. Un protocolo de coreografía no es directamente ejecutable, ya que necesita de procesos que cumplan con sus directrices.

El lenguaje *WS-BPEL* emplea los Web Services como sistema de comunicación externo mediante mensajes. El sistema de mensajes de *WS-BPEL* se basa en el protocolo Web Services Description Language (*WSDL*), para definir los mensajes; También incluye un sistema de «plugins» mediante el cual pueden escribirse expresiones y consultas en múltiples lenguajes. También provee de programación estructurada mediante las construcciones *if-then-else, while*... Así como de un sistema de ámbito, permitiendo variables locales, manejadores de eventos...

2.3. Pruebas de caja blanca

Las pruebas de caja blanca (o transparente) son un método de prueba de software consistente en comprobar la estructura y el funcionamiento de la lógica de un sistema en cada uno de sus pasos internos [Mye04] en contraposición con las pruebas de caja negra («white box») donde únicamente se comprueba la funcionalidad de la misma, por el método de comprobar sus entradas y sus productos o salidas.

Al realizar este tipo de pruebas se eligen entradas, valores e interacciones de forma estudiada, con el objetivo de que el flujo de programa pase por todas las posibles ”rutas de código” y cada parte interna provea de las salidas apropiadas. A diferencia de en las pruebas de caja negra, existe un conocimiento completo de las interioridades del sistema, del cual intenta extraerse la mayor cantidad de información y que sea correcto en cada punto.

Las herramientas más comunes para las técnicas de caja blanca incluyen:

- Pruebas de control de flujo.
- Pruebas de flujo de datos.
- Pruebas de ramas.
- Pruebas de rutas.

2.4. Invariantes dinámicos

Takuan es un generador de ”Invariantes dinámicos” o ”Invariantes potenciales”, de forma que parece apropiado dedicar un momento a definir que es exactamente. Un invariante es una declaración que siempre es cierta, no importa que ocurra o que valores se tomen durante un programa dado. El método fue propuesto por Robert Floyd [Flo67] para diagramas de flujo, y más tarde adaptado al código en general por Tony Hoare [Hoa69]. Posteriormente el tema también fue tratado por Ernst Dijkstra [Dij75, Dij76] que propuso su propio sistema.

De forma explicativa podríamos hablar de invariante como un predicado que es cierto para una secuencia de operaciones si este es verdadero antes de comenzar la secuencia y lo sigue siendo al terminarla [Wikb].

Sin embargo para *Takuan* un invariante resultaría más parecido a una propiedad o característica significativa, expresable en forma de predicado matemático que cumple el programa en algún punto de su flujo. *Takuan* pues hereda la interpretación de *Daikon* de invariante, como puede leerse en la definición que puede encontrarse en la página del proyecto [Dai00], de la cual se expone un extracto:

> An invariant is a property that holds at a certain point or points in a program; these are often seen in assert statements, documentation, and formal specifications. [..] Examples include:
>
> - ”.field >abs(y)”;
> - ”y = 2*x + 3”;
> - ”array a is sorted”;
> - ”for all list objects lst, lst.next.prev = lst”;
> - ”for all treenode objects n, n.left.value <n.right.value”;
> - ”p != null ->p.content in myArray”;
> - and many more.

Definición de invariante según *Daikon* [Dai00].

> Un invariante es una propiedad que se mantiene en cierto punto o puntos de un programa; estos son a menudo vistos en asertos, documentación y en especificaciones formales. [..] Ejemplos incluyen
>
> - ”.field >abs(y)”;
> - ”y = 2*x + 3”;
> - ”array a is sorted”;
> - ”for all list objects lst, lst.next.prev = lst”;

- ”for all treenode objects n, n.left.value <n.right.value”;
- ”p != null ->p.content in myArray”;
- y muchos más.

Definición de invariante según *Daikon* [Dai00].

La generación dinámica de invariantes es pues una técnica de prueba dinámica [Wikc] consistente en deducir probables invariantes en base al estudio estadístico del comportamiento de un sistema dado trabajando con datos de prueba. Ha resultado una técnica efectiva a la ayuda de pruebas de caja blanca [Bjø97, Ern01] pues, aunque a diferencia de los invariantes generados estáticamente, no se hallan siempre libres de errores debido a que dependen de la calidad de las ejecuciones realizadas, tienen un alcance mucho mayor.

Conviene pues distinguir entre *invariante* como propiedad del programa e *invariante dinámico* o *invariante potencial*, con una definición más laxa del mismo:

Definición 2.4.1. *Un predicado es un **invariante potencial** o **invariante dinámico** si expresa una propiedad que se mantiene como verdadera durante las distintas pruebas realizadas sobre un programa dado [Pal11].*

La simulación de un motor *WS-BPEL* es muy compleja, dado que existe una larga lista de características no triviales que deben ser implementadas, como concurrencia o manejadores de eventos. Además, durante el proceso, es necesario transformar el código de la composición *WS-BPEL* a otro lenguaje para comprobar su lógica interna. Sin alguna de estas características, no es posible probar la composición de forma precisa, siendo este proceso propenso a errores y a veces no representativo al no estar basado en la ejecución de código *WS-BPEL* en un entorno real, invocando servicios existentes [Pal08a].

De esta manera, a la vista de los diversos errores provocados por las simulaciones y transformaciones en otros lenguajes, *Takuan* emplea ejecuciones reales de casos de prueba sobre código de composiciones *WS-BPEL* empleando la herramienta de pruebas de unidad *BPELUnit* a partir de las cuales consigue inferir mediante un análisis estadístico en el analizador *Daikon*, en base al comportamiento del programa, propiedades representativas del programa a analizar. Propiedades que pueden ser usadas para mejorar la comprensión sobre el desempeño del programa, detectar errores típicos, recabar información sobre el mismo en base a complementar su documentación o suplir la falta de ella [RC07].

La generación automática de invariantes puede servir para mejorar un programa *WS-BPEL* de distintas formas [Gar08a]:

- **Depuración**: La aparición de un invariante con valores no esperados puede llevarnos al descubrimiento de un error en el programa. Valores de banderas sin sentido, llamadas a funciones con parámetros erróneos, límites superados en condiciones de iteración y otros.

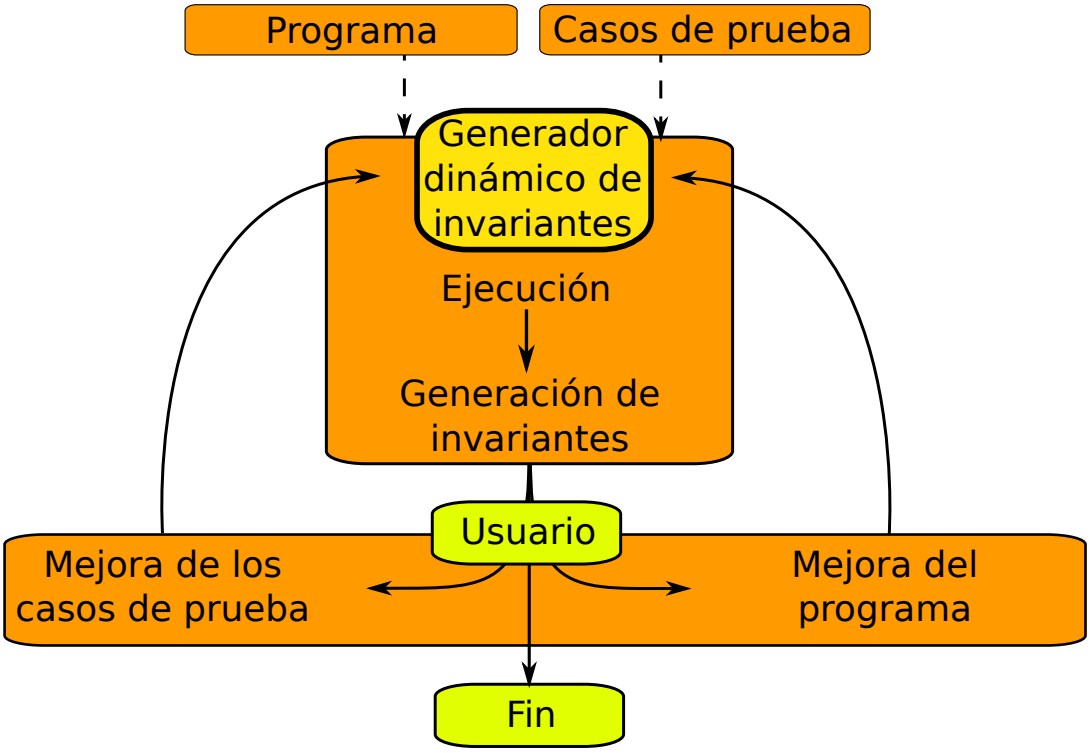


Figura 2.2: Esquema que muestra el sistema de retroalimentación para la mejora de casos de prueba unitarios de *BPELUnit* [Pal10]

- **Ampliación**: A la hora de modificar y ampliar un programa, es posible observar los invariantes que deben mantenerse y los que no deben entre diferentes versiones del mismo programa. Esto ayudaría a comprobar si durante la ampliación se cometió algún error que comprometa el normal funcionamiento del programa.
- **Documentación**: Los invariantes más representativos de la composición pueden ser expuestos en la documentación del sistema, completándola de esta manera.
- **Verificación**: Pueden emplearse los invariantes obtenidos por el programa para compararlos con la especificación del mismo y ayudar a la verificación de la corrección del mismo.
- **Mejora de pruebas**: Un invariante potencial no muy acertado puede indicar que los casos de prueba que están siendo empleados para inferirlo no son realmente representativos del uso real del programa, y por tanto, puede indicar en qué forma deben ser mejorados estos casos de prueba. El proceso de mejora de casos de prueba puede verse ilustrado en el esquema de la figura 2.2. En él, se indica como a partir de una composición *WS-BPEL* y una suite de casos de prueba unitarios *BPELUnit*, se generan invariantes, los cuales contienen información que permite mejorar tanto los casos de prueba como la composición, en un proceso contínuo de refinado.

2.5. Arquitectura

Takuan es la única herramienta de su tipo existente para emplear en *WS-BPEL 2.0* [Pal11] disponible hasta la fecha. Todo el código de *Takuan* se encuentra bajo licencia libre, disponible en [Man08] accesible para su prueba, estudio o modificación.

La arquitectura de *Takuan* consiste en un sistema compuesto que integra diferentes sistemas libres en su flujo, de forma que trabaja realizando el trabajo intermedio de forma que se compatibilicen las tareas de sus distintos componentes.

El sistema toma una composición *WS-BPEL 2.0* junto a una serie de casos de prueba definidos y produce como salida una lista de invariantes que el programa cumple durante todas las ejecuciones y pruebas realizadas. Para realizar esta tarea, se toman una serie de pasos.

1. Instrumentación.
2. Ejecución.
3. Análisis.

Primero es instrumentada la composición *WS-BPEL* de forma que quede preparada para a continuación ser ejecutada de forma correcta en el servidor *ActiveBPEL*. El siguiente paso es ejecutar los casos de prueba unitarios definidos mediante *BPELUnit* contra el servidor *ActiveBPEL*, empleando las modificaciones realizadas durante la instrumentación para que cada caso de prueba deje una traza donde pueden obtenerse de valores de variables, rutas tomadas en el código durante toda la vida del proceso. Una vez obtenidas estas, son tratadas y pasadas al analizador estadístico *Daikon*, el cual realiza un estudio sobre las trazas hallando las propiedades que se derivan de ellas.

2.6. Implementación

Los componentes de *Takuan*, mencionados anteriormente en la sección 2.1, son *ActiveBPEL*, *BPELUnit* y *Daikon*.

- ***ActiveBPEL***: Motor de ejecución *WS-BPEL 2.0* [Act08] empleado para la ejecución de pruebas y en la obtención de registros de ejecución para las pruebas ejecutadas mostrando del comportamiento del programa. El libro «The open source alternative» [Mee08] describe *ActiveBPEL* como puede leerse en las citas:

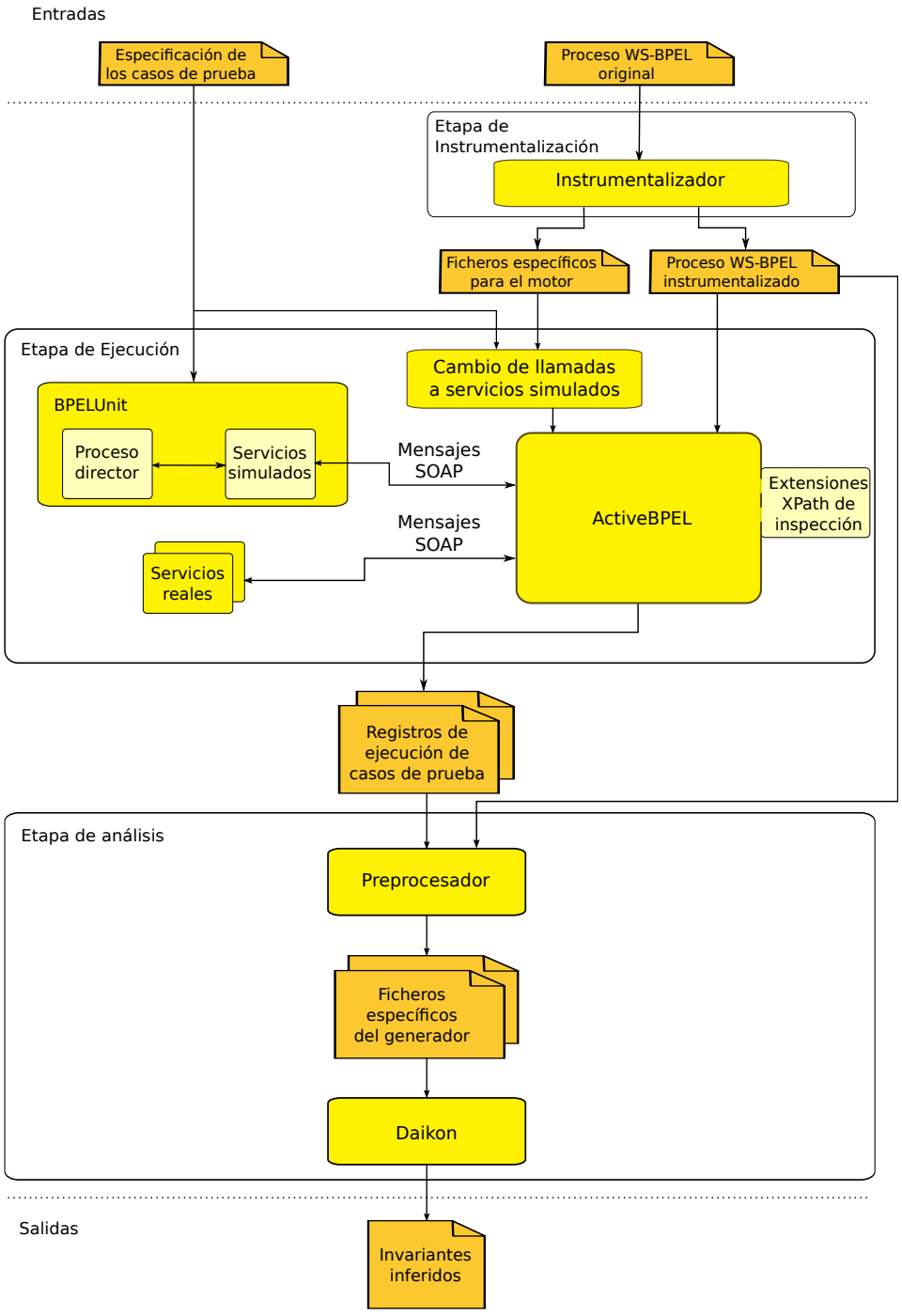


Figura 2.3: Arquitectura del Framework de generación de invariantes potenciales *Takuan* [Gar08a]

> Active Endpoints, Inc. [..] launched a separate open source product that complements its commercial products. The open source ActiveBPEL engine is a run-time environment for executing processes based on the WSBPEL specification [..]. The ActiveBPEL engine is also the core of Active Endpoints' ActiveBPEL Enterprise Server family of products. Active Endpoints licenses the ActiveBPEL engine under the GPL and its ActiveBPEL Enterprise products on commercial licensing terms.
>
> Extracto de la descripción de *ActiveBPEL* en [Mee08].

> Active Endpoints, Inc., [..] lanzó un motor de código abierto por separado que complementa sus productos comerciales. El motor de código abierto ActiveBPEL es un entorno de run-time para ejecución de procesos basados en la especificación WSBPEL [..]. El motor ActiveBPEL es también el núcleo de la familia de productos ActiveBPEL Enterprise Server de Active Endpoints. Active Endpoints licencia el motor ActiveBPEL bajo GPL y sus productos ActiveBPEL Enterprise, en términos de licencia comercial.
>
> Traducción de la descripción de *ActiveBPEL* en [Mee08].

ActiveBPEL resulta ser un motor liviano en comparación con sus contrapartidas, e incluye soporte para la implementación de extensiones *XPath*, lo cual es empleado por *Takuan* para la obtención de los registros de ejecución (o trazas) a partir de las que se puede consultar la actividad de la composición ejecutada [Pal11].

- ***BPELUnit***: Es un «framework» de automatización para pruebas unitarias disponible para *WS-BPEL 2.0*. Similar a software tan popular como *JUnit* [JUn00] el cual se emplea para pruebas unitarias en el lenguaje de programación *Java*. Permite a los desarrolladores probar de manera sencilla pequeñas porciones de código que han escrito de forma que puede comprobarse el buen funcionamiento del sistema y verificar que se mantiene de tal manera tras cambios o mantenimiento [BPE06].

  Las pruebas consisten en una secuencia de operaciones de entrada/salida, las cuales son ejecutadas por el «framework», sobre el cliente así como cada sistema externo (o «partner» del proceso *WS-BPEL 2.0*, forzando al proceso *WS-BPEL* a seguir cierta ruta dentro del código. El caso de prueba pasa si todas las operaciones previstas han sido realizadas con éxito durante la ejecución del caso [BPE08]. El sistema también permite la simulación de sistemas externos (o «partners») de forma que es posible probar por completo una composición que dependa de comunicaciones externas sin necesidad de establecer el entorno completo, cosa que en ocasiones es imposible. Este falso entorno que recrea las condiciones de ejecución normal de forma que pueda observarse la ejecución del proceso de forma controlada se conoce como «mockup». Un esquema del funcionamiento teórico a alto nivel del proceso de ejecución de pruebas unitarias en *BPELUnit* se encuentra disponible en la figura 2.4.

- ***Daikon*** implementa la detección dinámica de *invariantes potenciales* [3]. *Daikon* ejecuta los programas, observa la ejecución del mismo e informa de las propiedades que han sido

[3] Ver sección 2.4 sobre *invariantes* e *invariantes potenciales*

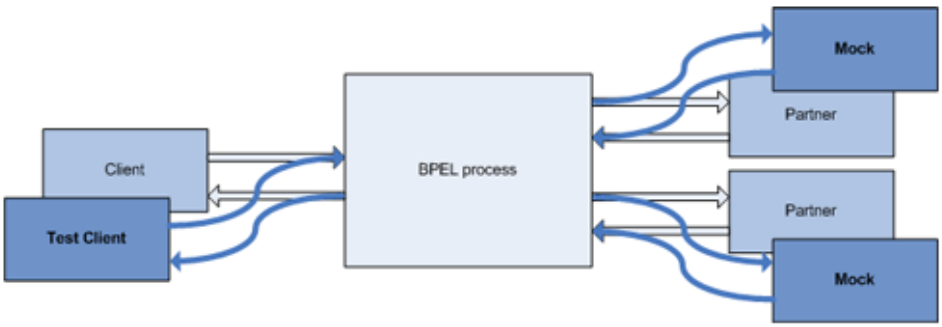


Figura 2.4: Esquema del proceso de *BPELUnit* [BPE08]

observadas durante la ejecución. Detecta propiedades para programas implementados en *C*, *C++*, *Eiffel*, *IOA*, *Java* y *Perl* [Dai00].

Daikon es software libre, disponible bajo licencia *MIT*, de forma que es posible acceder a su código fuente, así como modificarlo libremente. El código fuente de *Daikon* puede encontrarse junto a su distribución, que incluye el *CVS* completo con el código fuente.

A la hora de integrar todos estos distintos componentes en el flujo de programa, *Takuan* emplea la herramienta de control de ejecución basada en *xml*, *Apache ANT* [Apa11]. Esta herramienta, similar en muchos aspectos a la bien conocida *GNU Make* [Ger], se ocupa de manejar el flujo completo de *Takuan*, realizando cada sección del proceso de forma secuencial, partiendo de un directorio donde se encuentran los ficheros convenidos y generando finalmente un directorio de resultado que incluye todos los productos (invariantes) y subproductos (instrumentaciones, registros de ejecución, listados de estructuras de datos y variables) obtenidos a lo largo del proceso.

Los distintos pasos que realiza la tarea *ANT* en el flujo de *Takuan* son las siguientes, descritas por orden de ejecución.[4]

1. **Instrumentación**: Esta etapa toma la responsabilidad de preparar la composición *WS-BPEL 2.0* para ser ejecutada en el servidor *ActiveBPEL* adecuadamente y de forma que sea posible dejar un registro de la ejecución de la misma. Para ello se añaden instrucciones antes y después de cada entrada en una secuencia o rama del código, en cada cambio de valor de una variable, bucle o secuencia condicional, de forma que impriman los valores a consultar en cada momento.

   Esta etapa se encuentra implementada en *Java*, conocida como *InstrumentadorProcesoBPEL*, modificando el fichero de código fuente *WS-BPEL 2.0* añadiendo las consultas referidas anteriormente en cada punto del programa y acompañando a la composición de

[4] Algunas de estas tareas tienen subtareas o rutinas que se realizan por separado. Se han agrupado en *Instrumentación*, *Ejecución* y *Análisis* por encontrarse correspondencia lógica directa con las etapas descritas en la arquitectura en la sección 2.5.

ficheros auxiliares que serán necesarios para su correcta ejecución en el servidor *ActiveBPEL*.

Lo descrito implementa la etapa correspondiente con la parte superior derecha del diagrama de arquitectura de *Takuan* que puede encontrarse en la figura 2.3.

2. **Ejecución**: Una vez se dispone del código fuente *WS-BPEL* correctamente instrumentalizado para su ejecución y de los correspondientes casos de prueba *BPELUnit* para probar la composición, es necesario ejecutar estos contra el servidor *ActiveBPEL* de forma que se genere registro de ejecución que pueda ser empleado más tarde para inferir invariantes a partir del comportamiento de la composición analizada.

   En este punto es empleada la conveniente característica del servidor *ActiveBPEL* de permitir extensiones *XPath*. El servidor donde se ejecutan estas pruebas dispone de una extensión propia (*ExtensionesXPathUCA*) que permite que las consultas de valores añadidas al fichero fuente *WS-BPEL* durante la fase de instrumentación, se vean reflejadas en el registro de ejecución; obteniendo así de esta manera una traza detallada de la ejecución de cada caso de prueba contra su composición, que trabajado realmente en el servidor, sin necesidad de ser simulada.

   Lo descrito implementa la etapa correspondiente con la parte central ("Etapa de Ejecución") del diagrama de arquitectura de *Takuan* que puede encontrarse en la figura 2.3.

3. **Análisis**: Una vez obtenidos los correspondientes "registro de ejecución de caso de prueba" o "trazas" en la etapa de *Ejecución*, es preciso analizarlas estadísticamente para encontrar los invariantes que hubiere. Para realizar este análisis se emplea el analizador estadístico *Daikon*, ya descrito en apartados anteriores.

   Sin embargo, el sistema *Daikon* no admite como entrada válida el registro de ejecución tal y como el servidor *ActiveBPEL* lo genera, sino que especifica su propio formato de traza, a partir del cual trabaja. Es necesario, por tanto, transformar los registros de ejecución de caso de prueba generados durante la etapa de *Ejecución* al formato intermedio de *Daikon*. Esta tarea recae sobre el módulo intermedio *analizadorDaikon*.

   Este nuevo módulo se encuentra formado por una serie de guiones escritos en el lenguaje de programación *Perl* que transforman los registros de ejecución generados por el servidor *ActiveBPEL* y sus extensiones *XPath* en un formato de traza adecuado para *Daikon*. El proceso también genera archivos complementarios que incluyen declaraciones de variables y estructuras empleadas durante la composición *WS-BPEL* de forma que también puedan ser inspeccionadas y analizadas por *Daikon*. Debido al formato de traza de destino, es preciso transformar diversas estructuras de datos como matrices o árboles a otras planas y más sencillas. La conversión contempla dos maneras diferentes de aplicar estas transformaciones, bien mediante la división en un mayor número de estructuras diferentes y más pequeñas («matrix slicing») o bien mediante su conversión y aplanado a una estructura lineal y más grande («flattening»).

   Tras obtener las trazas en su correspondiente formato, estas son leídas por *Daikon* que realiza el análisis completo para obtener los diferentes invariantes potenciales. Opcionalmente y de forma adicional, *Daikon* puede emplear el sistema *Simplify*, el cual permite reducir la complejidad de los invariantes resultantes simplificando las expresiones producidas por *Daikon*. El resultado del análisis por *Daikon* es una lista de invariantes que

cumple el programa separados por punto de ejecución o secuencia. Lo descrito implementa la etapa correspondiente con la parte central ("Etapa de Análisis") del diagrama de arquitectura de *Takuan* que puede encontrarse en la figura 2.3.

2.7. Limitaciones

Takuan emplea *Apache ANT* como herramienta principal para el manejo de su flujo de ejecución. Esto acarrea una serie de consecuencias negativas a nivel de usabilidad como las que se listan a continuación:

- *ANT* es una herramienta de consola, lo cual provoca una interacción con el usuario nada visual, árida y en ocasiones complicada, dificultando la adopción y popularización de la herramienta por un público más general.
- La ejecución de las tareas obliga a la realización de todas las etapas del proceso de *Takuan*(esto es, *Instrumentación*, *Ejecución* y *Análisis*) todas y cada una de las veces que recurre a la herramienta tras realizar una modificación. Esta ejecución en bloque, hace difícil la múltiple ejecución de casos de prueba modificados, toma tiempo extra realizando de nuevo procesos ya hechos (como la *Instrumentación*, que solo debe hacerse una vez) y produce peores resultados.
- Los casos de prueba que van a ser ejecutados se encuentran en ficheros de tipo *BPELUnit* Test Suite (*bpts*) que deben ser modificados para incluir o excluir nuevos casos. Este es un proceso manual de edición de ficheros fuente que no es muy conveniente, sobre todo teniendo en cuenta que la mejora de casos de prueba mediante la generación de invariantes, uno de los objetivos principales de *Takuan*, implica la continua edición de este fichero, haciéndolo incómodo y poco práctico.
- El análisis a realizar por *Daikon* se hace siempre sobre una única ejecución de los casos de prueba. Esto produce malos resultados con las composiciones que dependen de factores aleatorios (generando invariantes falsos o desorientantes) y limita la precisión de los invariantes potenciales obtenidos a partir de casos de prueba que no siempre dan el mismo resultado ante la misma entrada. Por ello sería interesante poder almacenar múltiples ejecuciones de casos de prueba a analizar.

Estas características derivadas de la implementación de *Takuan* perjudican el uso de la herramienta y limitan en ciertos aspectos la utilidad de la misma debido a ser sencillamente demasiado manual en ocasiones, ofreciendo poca interactividad al usuario o requiriendo conocimientos más profundos de los que serían necesarios en varias etapas del proceso, muchas de ellas son repetidas al ser ejecutadas en bloque sin necesidad real de hacerlo, retrasando un proceso largo ya de por si.

Para solucionar parte de estas limitaciones y hacer del proceso de ejecución de *Takuan* algo más sencillo y ameno, se desarrolló una extensión o «plugin» para la herramienta de desarrollo

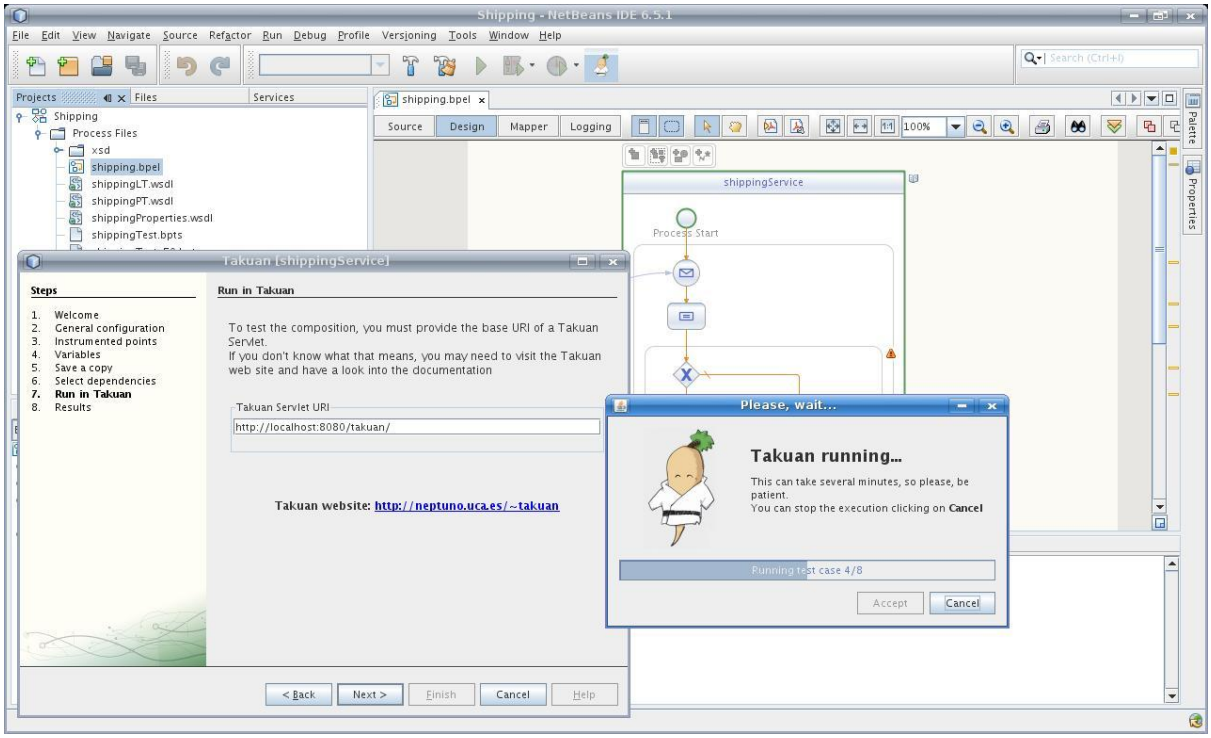


Figura 2.5: «Plugin» de *Takuan* para el editor *NetBeans* [Pal10]

de código abierto *NetBeans* [Ora11], que permite una interacción gráfica a la hora de generar invariantes para una composición *WS-BPEL 2.0* que esté siendo editada mediante el *IDE* en ese momento. En la figura 2.5 puede verse un ejemplo del uso de este «plugin», que permite, mediante un asistente gráfico, la ejecución de composiciones y su posterior análisis.

CAPÍTULO 3

Planificación

3.1. Descripción general del proyecto

IdiginBPEL es una herramienta que permite la generación dinámica de invariantes, basada en el generador de invariantes *Takuan*. Se encuentra escrita en el lenguaje de programación *Python* empleando el compuesto de bibliotecas gráficas *PyGTK*, y la herramienta de definición de interfaces *Glade*. El sistema consiste en una estructura en dos capas:

Una capa lógica, que sustituye el control de flujo de *Takuan* por uno propio y maneja así los diferentes componentes individuales que lo conforman. Esta capa redirige el sistema de análisis de *Takuan* reorientándolo así a proyectos basados en un fichero *WS-BPEL*.

Figura 3.1: Logotipo de *IdiginBPEL*

En la mencionada segunda capa, gráfica o de presentación, el sistema provee de una interfaz gráfica que muestra de forma sencilla el proceso completo y permite un mayor control en cada uno de los pasos que conforman el proceso completo de *Takuan*. La aplicación se ha desarrollado siguiendo un modelo de ciclo de vida iterativo, trabajando sobre un mismo prototipo que iba siendo mejorado de forma continua.

Las herramientas que se han empleado durante el desarrollo de este proyecto, bien sean *IDES*, bibliotecas o programas quedarán descritos en esta y subsiguientes secciones. Un listado de ellas:

- **Vim**: Editor de texto.
- **PyGtk**: Funciones *Python* para la biblioteca *GTK*.
- **Glade**: Diseño *WYSIWYG* [Nie99] de interfaces *GTK*.
- **Doxygen**: Generación automática de documentación.
- **BoUml**: *IDE* para desarrollo *UML*.
- **Git**: Control de versiones distribuido.
- **Subversion**: Control de versiones centralizado.
- **Pylint**: Comprobador estático de código.

3.1.1. *Vim*

El desarrollo completo de la aplicación se escribió empleando el editor de texto *Vim*, sin interfaz gráfica, directamente sobre la consola de comandos junto al uso de una útil serie de «plugins» que proveyeron al editor de capacidades de *IDE* para el lenguaje elegido para el desarrollo: *Python*.

Vim es un programa de edición de texto simple y potente cuya máxima es:

> Minimal effort. Maximal effect.
>
> A byte of Vim [H.04]

Originalmente distribuido por Bram Moolenaar en 1991, *Vim* es el acrónimo de *Vi Improved*. Acrónimo que tiene su explicación en ser *Vim* una extensión muy completa del original editor de texto *vi*. Fue liberado bajo una licencia «GNU Compatible» que incluye cláusulas que animan a los usuarios a realizar donaciones para los niños de Uganda.

La interfaz de *Vim* se encuentra basada en texto[1] y su funcionalidad se realiza a través de la linea de comandos. Su característica más destacada es la existencia de dos modos que diferencian lo que el usuario puede hacer en cada momento y como se interpreta lo que este teclea. Existe el modo ”edición”, en el cual lo que el usuario teclea se introduce directamente al texto, y el modo ”comando” o ”normal”, en el cual las teclas pulsadas por el usuario son interpretadas como acciones u órdenes.

El programa también mantiene un entendimiento diferente del texto, reconociendo entre líneas, frases, palabras, párrafos y otro tipo de estructuras de mayor complejidad que simplemente caracteres. Esta característica, unida al modo ”comando” permite al usuario realizar acciones muy complejas sobre el texto a un nivel más alto y más completo que el que permiten habitualmente el resto de editores, pudiendo de esta manera, eliminando así una barrera conceptual a la hora de editar el texto en si.

El editor se encuentra ”hecho por desarrolladores para desarrolladores”, e incluye muchas características útiles para escribir en lenguajes de programación, tales como coloreado e identación de texto según una sintaxis, comunicación con la «shell», expresiones regulares según la sintaxis de la herramienta *GNU sed*, pestañas, múltiples portapapeles, marcas de posición, «folding» o desdoblado de texto, diccionarios, depuradores de lenguajes, comprobadores de sintaxis y más.

El uso de *Vim* durante el desarrollo de *IdiginBPEL* estuvo ayudado de las siguientes extensiones o «plugins», las cuales dieron al editor capacidades de *IDE* (Integrated Development Environment):

- **NerdTree**: Exploración del sistema de ficheros a través del editor.
- **pydoc**: Autodocumentación para *Python* disponible directamente en el editor.
- **taglist**: Autocompletado de texto inteligente y autodocumentado.
- **libList**: Navegador de clases y código.
- **vimpdb**: Depurador *Python* integrado en el editor.
- **pyflakes**: Comprobador de código *Python* al vuelo.

3.1.2. *Python*

Python es un lenguaje imperativo interpretado dinámicamente tipado con manejo automático de memoria y orientado a objetos. Puede leer más sobre el lenguaje en el apéndice A.1 en la página 103. Esta sección se dedica a justificar la elección del lenguaje para la tarea de desarrollo realizada en *IdiginBPEL*.

El proyecto *Takuan* incluye muchos componentes distintos implementados en diferentes tecnologías que cubren una amplia gama de diferentes lenguajes y herramientas. Ejemplos de ello son:

[1] Aunque existe una versión gráfica, conocida como *gVim*, que añade barras, botones y menús.

- WS-BPEL
- WSDL
- Java
- Perl
- XSLT
- XPath
- ANT
- C++
- Modula-3
- Bash (shell)

Como puede apreciarse, la lista no es corta. El sistema a desarrollar implicaba la necesidad de integrar todas estas partes trabajando juntas de nuevo e interactuar con ellas. Junto a eso, los requisitos para el desarrollo demandaban flexibilidad durante el mismo debido a la naturaleza iterativa del ciclo de vida empleado, que podría haber obligado a la creación de un nuevo prototipo en cada una de ellas. Esta flexibilidad necesaria, implicaba la utilización de medios de desarrollo rápidos.

Por tanto las exigencias del sistema para con el desarrollo exigían de:

1. Unión y compatibilización de múltiples lenguajes y herramientas.
2. Desarrollo rápido y robusto.
3. Facilidad de uso.
4. Amplia disponibilidad de herramientas.

El lenguaje *Python* es usualmente llamado un ”lenguaje pegamento” [vR98] usándose como integrador de manera habitual en diferentes plataformas y entornos, lo cual soluciona el punto **1** de nuestra lista.

Con respecto a la facilidad de uso y su velocidad de desarrollo, programar código en *Python* resulta entre 5 y 10 veces más rápido que programar en *C/C++* y hasta entre 3 y 5 veces más rápido que *Java* [Lut09, vR98]. Esto lo hace un lenguaje ideal para el desarrollo veloz y ágil que se exige, cumpliendo de esta manera con los puntos **2** y **3** de la lista.

En cuanto a la disponibilidad de herramientas en *Python*. El lenguaje cuenta con una de las más grandes bibliotecas estándar [Pyt11c], y suele decirse que viene con ”pilas incluidas” [Lut09]. Proveyendo, de forma nativa, facilidades para el manejo de ficheros *xml* (fundamental para tratar muchos de los lenguajes listados anteriormente), control de hilos y comunicación con el sistema operativo. Esto solucionaba pues el punto **4** de nuestra lista.

Por todas estas razones, *Python* fue seleccionado como el lenguaje apropiado para realizar el desarrollo de *IdiginBPEL*, hasta ahora, con buenos resultados.

3.1.3. *GTK*

Siendo *IdiginBPEL* un desarrollo planteado para la plataforma *Linux*, y desarrollado en el lenguaje *Python*, se destacaban las siguientes bibliotecas gráficas libres alternativas, las cuales disponían de «bindings» para el lenguaje:

1. pyQt
2. wxPython
3. Tkinter
4. pyGTK

Qt [Nok11b] actualmente propiedad de *Nokia*, es un completo framework multiplataforma que incluye bibliotecas gráficas, soporte para «threading», un completo IDE y más características. Existen «bindings» realizados para la biblioteca, llamados *pyQt* [Pyt11d]. Los «bindings» no se encuentran mantenidos por *Nokia* sino por la comunidad.

wxWidgets es una opción, a través de los bindings *wxPython* [wxW11] son los «bindings» para el lenguaje *Python* de la biblioteca gráfica *wxWidgets*. Cuidadosamente orientada a objetos, multiplataforma y muy eficiente es utilizada frecuentemente en aplicaciones con altos requisitos de portabilidad.

Tkinter [Pyt11e] se describe como la biblioteca estándar de interfaces de usuario gráficas. Consiste en una capa orientada a objetos encima de *Tcl/Tk*. Multiplataforma, sencillo y bien documentado, aunque no destaca por destacar visualmente. Sólo incluye la biblioteca gráfica, no es un framework completo.

GTK también conocida como "*GIMP Toolkit*" [Gno11d], es también un framework multiplataforma cuyo fin es crear interfaces gráficas de usuario. Ofrece un completo conjunto de componentes y es posible emplearlo en proyectos de muy diferente tamaño. Incluye una desarrollador gráfico de interfaces llamado *Glade* que permite construir la interfaz en tiempo de ejecución a partir de su descripción en un fichero *xml*. Sus bindings, *PyGTK* son robustos y maduros.

La decisión tomada descartó *wxWidgets* debido a la mala experiencia de las pruebas realizadas con ella, obligando a un desarrollo basado en código, con grandes declaraciones de elementos que ensuciaban el programa. En el mismo fallo incurría *Tkinter*, al que que cabe destacar una aún peor calidad gráfica final debido a la pobre integración del sistema de ventanas en cada plataforma. La decisión final quedó pues entre *Qt* y *PyGTK*.

Ambos sistemas, *Qt* y *PyGTK*, conforman un framework bastante completo. Si bien *Qt* contiene mayor funcionalidad y cubre muchos más aspectos del desarrollo aparte de la interfaz gráfica de usuario, este aspecto no se valoró tanto debido a que no parecía conveniente implementar usando una biblioteca externa a través de unos «bindings», funcionalidad ya existente dentro de la biblioteca estándar del lenguaje *Python*. Las cuestiones determinantes a la hora de elegir *PyGTK* fueron:

- **Mejor documentación para los «bindings»**: *Qt* cuenta con muy buena documentación [Nok11a], pero la documentación disponible para *pyQt* [Riv11] es muy pobre y carece de tutorial y ejemplos. Por contra, la documentación de *GTK* y *PyGTK* [Gno11a, Gno11b] es mucho más rica y completa.
- **Madurez**: *pyQt* fue desarrollada dos años más tarde que su contrapartida para *GTK*, y estudiando su código fuentes puede comprobarse como su nivel de madurez es inferior.
- **Comunidad y soporte**: Tras realizar un estudio de penetración de las diferentes bibliotecas en la comunidad de software libre, la comunidad de desarrollo *PyGTK* es visiblemente más grande y activa [Gno11c]. Las bibliotecas *pyQt* cuentan con un menor número de seguidores, de aplicaciones desarrolladas [Pyt11a] y por tanto de soporte en caso de encontrar posibles problemas durante el desarrollo.
- **Experiencia previa**: El alumno poseía cierta experiencia y familiaridad previa con el framework de *GTK* al haber desarrollado anteriormente aplicaciones basadas en *gtkmm*, los «bindings» para C++ de *GTK*.

Por todas estas razones, la biblioteca gráfica elegida para el desarrollo de *IdiginBPEL* fue *PyGTK*.

3.1.4. *Glade*

Podríamos considerar a *Glade* como un diseñador gráfico de interfaces. *Glade* consiste en una aplicación [Gno11g] y en dos bibliotecas, *libglade* y *gtkbuilder* [Gno11f, Gno11e].

La aplicación, de la que podemos ver una imagen en la figura 3.2 (página 25), consiste en un diseñador gráfico de interfaces, el cual, aproximándose a la filosofía «WYSIWYG» [Nie99], permite ir visualizando de forma aproximada la interfaz que se va creando a golpe de ratón. Esta interfaz que se define es exportada en un formato descriptivo basado en *xml*.

La otra parte de *Glade*, las bibliotecas, permiten leer en tiempo de ejecución los *xml* que describen la interfaz, creando esta de forma automática, sin necesidad de definir largas listas de elementos dentro del código de la aplicación, a costa de una ligera sobrecarga a la hora del arranque de la misma. Estas bibliotecas son dos:

- **libglade**: Original y externa al proyecto *GTK*. Abandonado su mantenimiento, se recomienda el uso de *gtkBuilder*.
- **gtkBuilder**: Adaptación por parte del equipo de *GTK* de la anteriormente biblioteca externa *Glade*. No es exactamente igual que la original, aunque mantiene gran parte de la funcionalidad de forma similar.

La biblioteca elegida para realizar la interfaz fue, de forma natural *gtkBuilder*. Los ficheros de descripción de interfaz *xml* (ficheros `.glade`) generados con la aplicación son abiertos y parseados por la aplicación en su arranque y conforman de forma modular, dinámicamente, la interfaz casi completa de la aplicación.

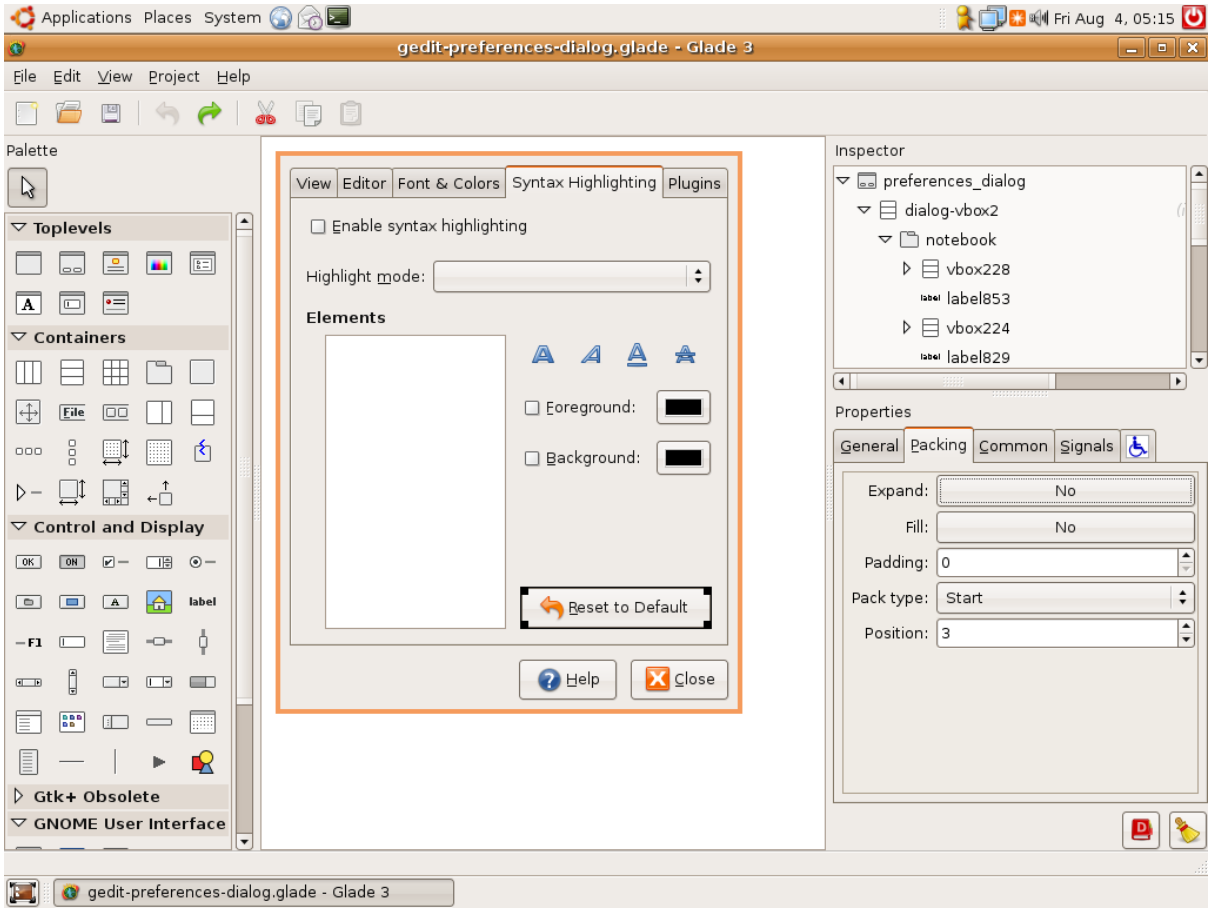


Figura 3.2: Diseñador de interfaces «WYSIWYG» *Glade*

3.1.5. *Doxygen*

Doxygen [Van] es una herramienta para la generación automática de documentación. La idiosincrasia de la herramienta consiste en que la documentación sea escrita por los propios desarrolladores dentro de los mismos ficheros fuente que editan, a la vez que crean el código que van a documentar, ahorrando así tiempo y esfuerzo en la escritura de una documentación a posteriori, proceso tedioso y propenso a errores.

La idea de funcionamiento de la herramienta es pues, que el usuario de la misma escriba la documentación dentro de la propia aplicación, como comentarios que siguen un patrón especial, en el código de los ficheros fuente que desea autodocumentar, y *Doxygen* será capaz de leer estos fuentes, extraer la información referente a la aplicación y generar la documentación automáticamente.

La herramienta genera documentación para una gran variedad de lenguajes[2], siendo esta documentación disponible en multitud de formatos de salida como *LaTeX*, *HTML* y otros. Esta documentación incluye diagramas, listados de funciones y clases así como páginas escritas directamente por el usuario.

3.1.6. *BOUML*

La herramienta de modelado *UML 2.0* (*Unified Modeling Language*) *BOUML* [Aur11] permite dibujar dibujar diversos diagramas de clase, de secuencia, casos de uso, componentes entre otros, de forma cómoda y sencilla, pudiendo ser exportadas estas composiciones de fácilmente a formato `png` o `svg`.

La herramienta permite además la generación de código *Java/C++/Python/Perl* a partir de la descripción realizada en *UML* en el programa, y su proceso complementario, es decir, la ingeniería inversa sobre un programa *Java/C++* a partir de su código.

Implementada en *C++* y empleando las bibliotecas gráficas *Qt*, el programa es multiplataforma y puede emplearse en *Windows*, *Linux* y *Mac*. Actualmente, como reza un anuncio en su página principal, su desarrollo se encuentra parado; al parecer el desarrollador ha sufrido violaciones de licencia e insultos en diversos debates y solo mantendrá errores de la aplicación sin mejorarla en característica alguna.

[2]Entre los que no se encuentra *Python*, para ver cómo se adaptó la herramienta para su uso con el lenguaje elegido para el desarrollo de *IdiginBPEL*, consulte el apéndice A.3 disponible en la página 109.

3.1.7. Control de versiones

Durante el desarrollo de un proyecto de cierta enjundia[3] es absolutamente necesario el empleo de una herramienta de *Control de Versiones* [Wik11a]. Una herramienta de *Control de Versiones* es aquella que gestiona los cambios que se realizan sobre el contenido o configuración de diferentes componentes a los cuales supervisa y controla. Este tipo de herramientas se hallan basados en el concepto de versión o revisión, los cuales se interpretan como una variación en el contenido o configuración de los mismos elementos que se manejan por el sistema.

> What is version control, and why should you care? *Version control* is a system that records changes to a file or set of files over time so that you can recall specific versions later. For the examples in this book, you will use software source code as the files being version controlled, though in reality you can do this with nearly any type of file on a computer.
>
> If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allows you to revert files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover. In addition, you get all this for very little overhead.
>
> Scott Chacon [Cha09]

> ¿Qué es el control de versiones, y por qué debería importarme? El *Control de versiones* es un sistema que guarda cambios sobre un fichero o conjunto de ficheros a lo largo del tiempo de forma que puedas recuperar versiones específicas más tarde. A partir de los ejemplos en este libro, usarás código fuente de software como los ficheros a ser controlados, sin embargo en realidad puedes hacer esto con prácticamente cualquier tipo de fichero informático.
>
> Si eres un diseñador gráfico o web y quieres mantener cada versión de una imagen o estructura (lo cual querrás hacer muy probablemente), emplear un Sistema de Control de Versiones (VCS) es una sabia decisión. Te permite devolver ficheros a un estado anterior, revertir el proyecto entero a un estado anterior, comparar cambios en el tiempo, ver quien fue el último que modifico algo que podría haber causado un problema, quien introdujo una característica, y cuando, y más cosas. Usar un VCS significa generalmente que si fastidias las cosas o pierdes ficheros, siempre puedes recuperarlos fácilmente. Además, obtienes todo esto por muy poco esfuerzo.
>
> Scott Chacon [Cha09]

Existen varios tipos de sistemas de control de versiones:

[3] *IdiginBPEL* consiste en más de 6.100 líneas de código, considerando únicamente de código *Python*.

- **Sistemas de control de versiones local**: Un SCV local mantiene una simple base de datos que incluye todos los cambios a los ficheros bajo control de revisión en alguna otra parte de la propia máquina de desarrollo.
- **Sistemas de control de versiones centralizados**: Esta clase de sistemas mantienen todos los ficheros versionados en una localización central. Esto provee varias ventajas, permitiendo a más desarrolladores trabajar juntos y facilita la administración al simplificar las copias.
- **Sistemas de control de versiones distribuidos**: Con sistemas de control distribuidos los clientes no únicamente obtienen las versiones desde un lugar centralizado, ellos mismos mantienen toda la información acerca del proyecto bajo control, y pueden obtenerla y se sincronizan con otros nodos idénticos a ellos. Esto permite que si un servidor cae junto con el repositorio, los usuarios que estaban colaborando basándose en el pueden no solo seguir trabajando, sino restaurar la información perdida.

Durante el desarrollo de este proyecto y por razones externas se han empleado dos sistemas de control de versiones diferentes al mismo tiempo. Como se verá, uno se prefirió sobre el otro, empleándose este último a modo de espejo.

Subversion

Apache Subversion (normalmente abreviado *svn*), es un un sistema de control de versiones centralizado creado en el año 2000 por la empresa *CollabNet* para mantener el historial de versiones de ficheros código fuente, páginas web y documentación. Su objetivo era el sustituir de forma compatible al por aquel entonces más ampliamente usado *Concurrent Versions System* (*CVS*).

Subversion se ha expandido desde entonces, convirtiéndose en poco tiempo en la herramienta de control de versiones más usada por todo tipo de clientes y desarrolladores. El código referente a la implementación de este sistema se encuentra licenciado bajo la *Apache License*, y es por tanto software libre. Es un sistema relativamente sencillo que puede ser integrado en las rutinas de programación de muchos equipos sin modificar en exceso su forma y ritmo de trabajo, lo cual ayudó a su rápida aceptación por parte de la comunidad de desarrolladores.

Durante el desarrollo de *IdiginBPEL*, especialmente durante sus dos primeras iteraciones *Subversion* fue la herramienta elegida como sistema control de versiones. Esta decisión no resultó tomada por motu propio por parte de la organización; de hecho vino impuesta por las condiciones establecidas en la IV Edición del Concurso Universitario de Software Libre [Con11], que especificaba el uso de la *Forja de RedIris* [Jav09b], que únicamente disponía de servidores de control de versiones *Subversion*.

De esta forma, el proyecto se inició y avanzó empleando la herramienta de control de versiones centralizada *Subversion*, al ser la única alternativa disponible para participar en el concurso. Más tarde, a la finalización del mismo, se pudo cambiar de herramienta, sin embargo, gran parte del desarrollo se realizó bajo *Subversion*.

Git

Git es un sistema de control de versiones distribuido y desarrollado por *Linux* Torvalds, creador y mantenedor del *Kernel* de *Linux*. Sus características incluyen operaciones en local, integridad general mediante operaciones criptográficas y la característica de no manejar parches, sino «snapshots» o «imágenes» de los ficheros que maneja.

El origen de *Git* [Cha09], se encuentra ligado al desarrollo del *Kernel* de *Linux*. Durante la mayor parte del mantenimiento del *Kernel* (1991-2002), los cambios realizados al mismo eran distribuidos y aplicados mediante ficheros comprimidos y parches aplicados al sistema. A partir de 2001, la comunidad comenzó a emplear la herramienta de control de versiones distribuida de licencia propietaria *BitKeeper*. En 2005, tras un cambio en las condiciones de uso de *BitKeeper* que impedía su empleo gratuito, la comunidad *Linux* comenzó el desarrollo de su propia herramienta basada en lo aprendido con *BitKeeper*. Los objetivos de este nuevo sistema eran:

- Velocidad
- Diseño simple
- Buen soporte para desarrollo no-lineal
- Completamente distribuido
- Manejo eficiente de grandes proyectos (tamaño y velocidad).

Desde su nacimiento en 2005, *Git* ha evolucionado y madurado para ser fácil de usar y aún así mantener estas cualidades iniciales. Es realmente rápido, eficiente con grandes proyectos y cabe destacar su buen sistema de ramas para desarrollo no lineal.

En el momento de la finalización de la IV Edición del Concurso Universitario de Software Libre, la restricción a emplear una forja de *RedIris* quedaba eliminada, siendo pues posible un cambio a elección del desarrollador sobre el lugar donde hospedar y manejar el proyecto, y de la tecnología a emplear para ello.

Se estimó que el uso de la herramienta de control de versiones distribuida *Git* mejoraría el flujo de trabajo en el desarrollo de la aplicación, al manejar de forma mucho más efectiva que *Subversion* diferentes ramas de código, y, sobre todo, al permitir el uso del sistema de control de versiones de forma local al usuario y sin conexión. Era un factor limitante para el usuario el desarrollo en lugares donde no existía conexión a la red, y por tanto era imposible el uso del sistema de control de versiones centralizado *Subversion*, haciendo pues del desarrollo una tarea dependiente de la conexión a Internet, no siempre disponible.

De forma externa al proyecto no debe existir idea de un gran cambio, ya que el proyecto hospedado en la forja de *RedIris* no se abandonó, sino que se mantuvieron ambos sistemas de control de versiones durante todo el desarrollo restante de la aplicación. Este tema queda cubierto en el apéndice A.2 que puede leerse en la página 105.

3.2. Calidad del código

Durante el desarrollo de la aplicación se han empleado diversos tipos de herramientas para mantener un nivel aceptable de calidad en el código escrito. Estas herramientas aseguran que el programa se encuentra razonablemente distribuido, es legible, no contiene errores obvios y no comete faltas de diseño graves.

3.2.1. Comprobación dinámica

La comprobación dinámica del código se realiza tratando de compilar la aplicación de forma contínua mientras se trabaja sobre ella. Esto permite encontrar tanto errores de sintaxis como lógicos, detectando variables declaradas pero no utilizadas, errores de comparación y asignación, código peligroso, ambiguo o mal formateado etc...

Este tipo de comprobación se realizó mediante el uso de la herramienta *Pyflakes*, la cual es capaz de comprobar el código *Python* que se escribe de forma constante, resaltando tanto fallos como errores potenciales. La aplicación da al usuario indicaciones para que se apegue a las reglas de formato vigentes para el lenguaje *Python*, definidas en el documento «Style Guide for Python Code», más conocido como *PEP8* [Gui01].

Este documento contiene una pormenorizada guía de estilo para el lenguaje, definiendo de forma clara la mayoría de condiciones que un código limpio, legible y bien escrito debe cumplir. Estas indicaciones y directivas son comprobadas por la herramienta y resaltadas en el editor de texto al incumplirse. La norma también apela al sentido común del programador, exhortándole a saltarse estas mismas normas en el momento en que el seguirlas resulte perjudicial para la legibilidad y comprensibilidad del código a escribir.

3.2.2. Comprobación estática

La comprobación estática del código revisa la legibilidad del programa vigilando que este sigua la guía de estilo del lenguaje, comprueba también errores comunes de programación, aspectos indicativos de mal diseño, como identificadores inadecuados o poco expresivos, funciones demasiado largas, clases demasiado grandes, paquetes demasiado pequeños... Junto a otra variedad de factores que permiten obtener una medida aproximada de la calidad del código escrito.

Este tipo de comprobación está basado en ciertas premisas y condiciones previamente establecidas, muchas de ellas apoyadas simplemente en la mera convención. Además, este método se encuentra limitado a la información que pueda obtenerse del programa escrito de forma *estática*, esto es, sin ejecutarlo, sino simplemente explorando su código, limitando así la información de tipos de datos y fallos de ejecución que pueda encontrar.

Para la comprobación estática de código se empleó la herramienta *Pylint* [Pyl06], creada y mantenida por Sylvain Thenault [Pyt11b],

la cual toma paquetes de código *Python* y comprueba una larga serie de requisitos indicativos de la calidad del código como:

- Clases demasiado grandes.
- Funciones demasiado largas.
- Variables sin definir.
- «Imports» innecesarios.
- Tipos incorrectos para ciertas operaciones.
- Excepciones no capturadas.
- Nombres de variable inadecuados.
- Comportamientos peligrosos (como la orden `return` dentro de un bloque `finally`).
- Y otros...

Pylint tras realizar una comprobación del código genera una nota numérica que indica de forma resumida la calidad del código resultante. Este proyecto, *IdiginBPEL*, obtiene **7.4** sobre 10 en la evaluación de *Pylint*[4]. A pesar de ser una buena nota, existen detalles por las que esta baja de forma significativa, como la presencia de clases que siguen el patrón *Fachada* [Gam94], y que son necesariamente muy grandes, puntuando negativo en tamaño de clase, número de atributos y miembros por clase así como otras medidas de control de código que bajo ciertos supuestos y en excepciones no deben aplicarse, por lo que la nota no es realmente representativa, puntúa a la baja.

[4]La evaluación se realiza sobre los paquetes `idg` e `idgui` llamando a la orden con los siguientes argumentos: `pylint idg idgui --additional-builtins _` de esta forma no se interpreta la llamada a la función _ de gettext, como un error, al no encontrarse esta definida hasta la ejecución del programa. De otra manera, la nota baja equivocadamente a 4.

3.3. Ciclo de vida

Durante el desarrollo de la aplicación *IdiginBPEL* se ha seguido un modelo de ciclo de vida iterativo incremental o creciente. Una definición de este modelo puede encontrarse en la siguiente definición:

> Este modelo [de ciclo de vida iterativo incremental] persigue el desarrollo de un sistema de programas de forma incremental, permitiéndole al desarrollador sacar ventaja de lo que se ha aprendido a lo largo del desarrollo anterior, incrementando, versiones entregables del sistema. El aprendizaje viene de dos vertientes: el desarrollo del sistema, y su uso (mientras sea posible). Los pasos claves en el proceso son comenzar con una implementación simple de los requerimientos del sistema, e iterativamente mejorar la secuencia evolutiva de versiones hasta que el sistema completo esté implementado. En cada iteración, se realizan cambios en el diseño y se agregan nuevas funcionalidades y capacidades al sistema.
>
> Extraido de [Som02, Wik11b]

Es por tanto que el proyecto se realizó en base a la creación de un prototipo, el cual tras diversas iteraciones y rediseños fue incluyendo funcionalidades hasta alcanzar las expectativas esperadas por el cliente; en este caso, los tutores *Manuel Palomo Duarte* y *Antonio García Domínguez*.

La elección de este modelo de desarrollo fue tomada a la vista de las siguientes necesidades:

- **Ausencia inicial de modelo claro para la interfaz**: Inicialmente el modelo para la interfaz no se encontraba definido y se ignoraba cuál sería el modelo más efectivo para envolver las operaciones realizadas por *Takuan*. A la vista de la ausencia de una idea fuerte y preconcebida en el cliente, se estimó como alta la probabilidad de cambio en los requisitos inicialmente tomados.
- **Necesidad de presentación del programa en estados intermedios**: Al encontrarse *Takuan* siendo desarrollado como parte del grupo *SPI&FM* sujeto a periódicas reuniones en seminarios en las que se trataba y controlaba el progreso del proyecto.
- **Probabilidad de cambio en el sistema subyacente**: Al encontrarse por aquel momento *Takuan* aún en desarrollo y al depender *IdiginBPEL* de pequeños detalles de implementación de este otro, fueron previstos probables cambios forzados debidos a cambios de versión en el programa base (como de hecho ocurrieron durante la cuarta iteración).

3.4. Descripción del desarrollo y calendario

El desarrollo se compuso de cinco iteraciones, incluyendo la iteración de inicialización o arranque, a lo largo de las cuales el prototipo pasó de ser una interfaz falsa sin funcionalidad alguna al programa completamente operativo y probado.

Iteración primera. Inicialización

Esta primera iteración comprende el período de desarrollo entre la definición y exposición del problema a abordar con *IdiginBPEL* , las diferentes entrevistas con el cliente a partir de las cuales se obtuvieron los requisitos de la aplicación, el trabajo de documentación realizado a fin de comprender y lograr abarcar la complejidad del sistema ya desarrollado (y aún en desarrollo en aquel momento) *Takuan* y la creación del primer prototipo de la interfaz, como un esqueleto gráfico aún sin funcionalidad.

Resumen de tareas realizadas durante esta iteración.

- Documentación acerca de *Takuan*.
- Recolección de requisitos.
- Propuesta y aceptación de interfaz sobre «mockup».
- Definición de formato de proyecto.
- Documentación y estudio acerca de las tecnologías a emplear durante el desarrollo.
- Desarrollo y presentación de interfaz ejecutable sin funcionalidad. Primer prototipo.

La primera tarea a realizar durante la primera iteración consistió en la formación y documentación acerca de *Takuan*, el sistema a abordar a continuación. Esta documentación incluyó el acceso al respositorio de código fuente de *Takuan* [SPI11], donde se exploró el código de los distintos componentes que conformaban el sistema, se realizaron pruebas y se pudo ver el programa en funcionamiento, así como la lectura de diversos artículos en inglés y español que explicaban de forma resumida la estructura general del sistema y su arquitectura:

- *An Architecture for Dynamic Invariant Generation in WS-BPEL Web Service* [Pal08a].
- *Framework para la Generación Dinámica de Invariantes en Composiciones de Servicios Web con WS-BPEL* [Gar08a].
- *Implementación de un framework para la generación dinámica de invariantes en composiciones de servicios web con WS-BPEL* [Gar08b].
- *Takuan: A Dynamic Invariant Generation System for WS-BPEL Compositions* [Pal08b].
- *Enhancing WS-BPEL Dynamic Invariant Generation Using XML Schema and XPath Information* [Pal09].

Adicionalmente también sirvió como apoyo documental a fin de comprender la estructura interna del sistema el Proyecto de Fin de Carrera de Alejandro Álvarez Ayllón, titulado *Reingeniería y ampliación del generador dinámico de invariantes potencial para composiciones de servicios web en* WS-BPEL Takuan [Á10].

Una vez suficientemente documentado, habiendo instalado el sistema, experimentado con él y analizado su estructura e implementación, se desarrolló un prototipo en papel de varias páginas de «mockups», que fueron presentados en Octubre de 2009. Tras la aprobación y modificación de la operatividad de la interfaz, realizada en dos entrevistas con el cliente, se procedió a proyectar la realización de una versión ejecutable de la misma interfaz, aún sin funcionalidad.

A fecha de *20 de Noviembre de 2009* fue presentada la versión ejecutable de la interfaz realizada en *PyGTK*. Esta puede verse en las figuras 3.3a (página 35) y 3.3c (página 35). Para la fecha comentada, la interfaz aún se encontraba carente de funcionalidad, hueca, esto es, no presentaba acción ejecutable alguna y no era capaz de realizar una réplica del proceso de *Takuan* completo.

Otros hitos dignos de mención durante esta primera iteración fueron la inscripción del proyecto en la IV Edición del Concurso Universitario de Software Libre, el cual hizo que para la primera parte del desarrollo se emplease la forja de *RedIris* [Jav09b] y la apertura del blog del proyecto [Jav09a]. Esta iteración duró de octubre de 2009 a diciembre de 2009.

Iteración segunda. Funcionalidad

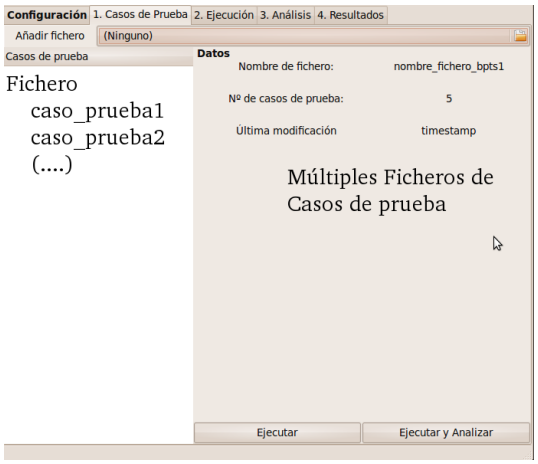
En esta segunda iteración se dotó de funcionalidad al esqueleto previamente diseñado y construido pero que no podía realizar acción alguna. En el mismo proceso se implementaron los procesos necesarios para suplir las carencias de *Takuan* a la hora de realizar importaciones de ficheros y otras tareas. Se implementó la lógica necesaria para que el programa operase empleando el formato definido para los proyectos.

En esta misma iteración comenzaron a encontrarse los primeros problemas de desarrollo. En su mayoría, relacionados con inconsistencias entre estándares basados en *xml*. También se encontró un error en el código del instrumentador de *Takuan*, que fue reportado y resuelto por su autor, Antonio García Domínguez; también se mejoraron los guiones *Perl* de preproceso mediante una serie de parches.

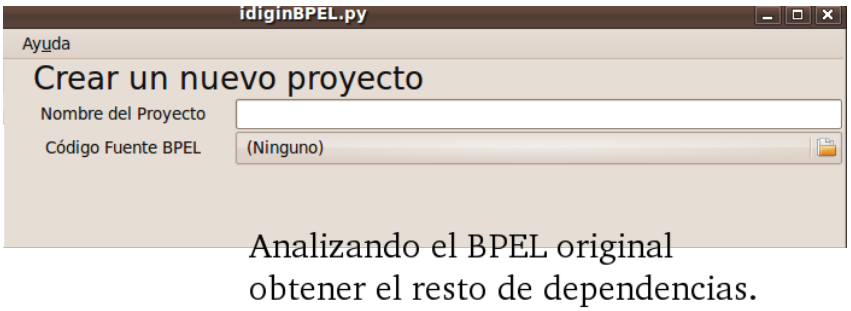
Resumen de tareas realizadas durante esta iteración.

- Implementación del sistema de control de *Takuan* desde *Python*.
- Implementación de la búsqueda recursiva de dependencias *bpel*.
- Implementación del manejo de la *Instrumentación*.
- Implementación del manejo de la *Ejecución* en el servidor *ActiveBPEL*.
- Implementación de la conversión de trazas y del *Análisis* por *Daikon*.

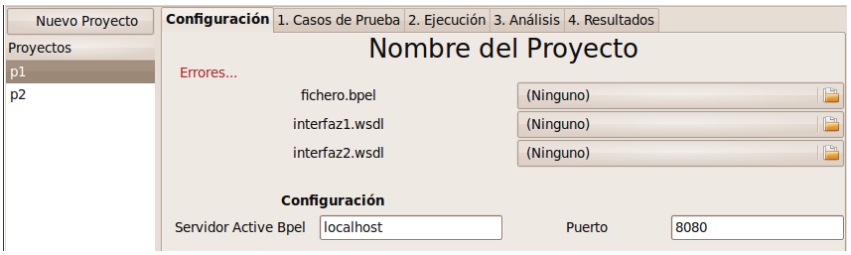
(a) Portada y lista de proyectos



(b) Casos de prueba

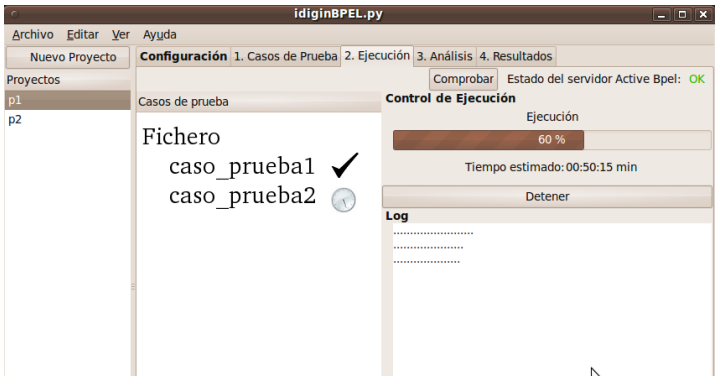


(c) Crear proyecto

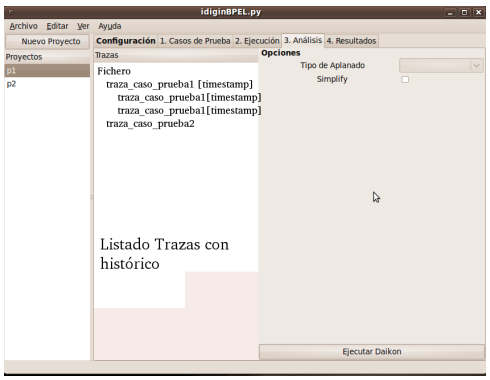


(d) Pantalla de proyecto

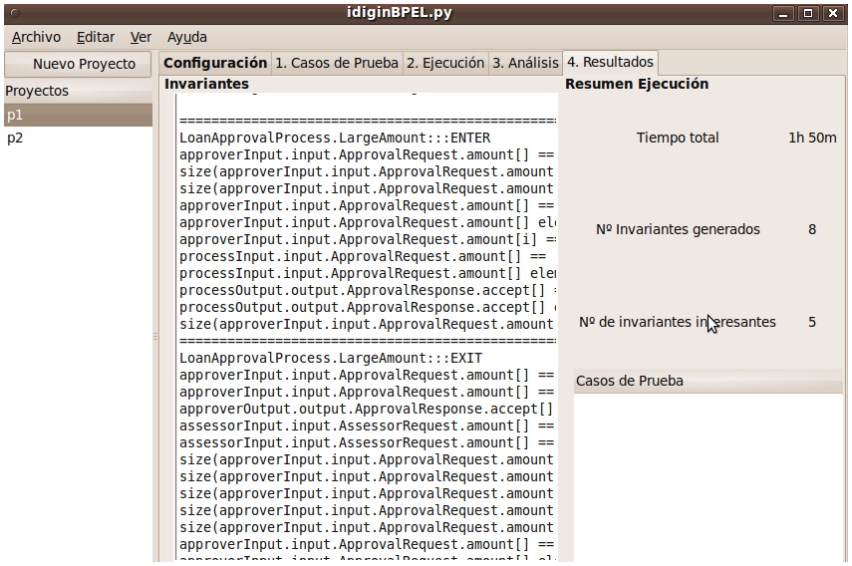
Figura 3.3: «Mockup» ejecutable o ”esqueleto” de la aplicación realizado como primer prototipo (1/2).



(a) Ejecutar casos



(b) Analizar casos



(c) Invariantes resultado

Figura 3.4: «Mockup» ejecutable o ”esqueleto” de la aplicación realizado como primer prototipo (2/2).

- Detección de errores en *Takuan*. Pequeña ampliación de guiones *Perl*.

La primera versión básicamente funcional de *IdiginBPEL* se presentó en el seminario del grupo *SPI&FM* del 8 de marzo del 2010. Esta iteración por tanto abarca desde diciembre de 2010 y termina aquí con la funcionalidad básica completa. Tras la presentación se recibieron sugerencias, notificaciones de cambios y diversas ideas de mejora sobre nuevas funcionalidades a añadir. Hasta aquí, la arquitectura de la aplicación había sido muy sencilla, con cuatro clases muy grandes que aglutinaban la mayoría de la funcionalidad existente. Llegados a este punto, se vio la necesidad de reestructurar la aplicación de forma que existiese más orden y menor acoplamiento, labor que será abordada en la tercera iteración.

Iteración tercera. Reestructuración

Tras contar con la funcionalidad básica de *Takuan*, es decir, pudiendo emular el comportamiento aproximado de la antigua herramienta de la misma forma, esta iteración se centró en la inclusión de nuevas funcionalidades, ajuste del funcionamiento de las ya existentes, corrección de errores existentes y, principalmente, en una reestructuración de la arquitectura empleada.

Principales tareas realizadas durante la tercera iteración.

- Importación y Exportación de proyectos en formato *.idg*.
- Control del servidor *ActiveBPEL* desde la propia aplicación.
- Depuración de la detección de trazas y el almacenamiento de las mismas.
- Reestructuración de la aplicación y creación de una abstracción para manejar los ficheros fuente *bpel*, *bpts*, *ant* y otros.
- Desarrollo de la edición de opciones y ayuda.

La poca idoneidad de los prototipos de desarrollo rápido obliga, siguiendo el mimo modelo de ciclo de vida iterativo, a bien descartarlos y comenzar desde cero, o invertir recursos en reciclarlos. Debido a que en nuestro caso, al llegar a la tercera iteración, se disponía de un prototipo completamente funcional que había recibido ya cierto refinamiento, se resolvió reestructurar el sistema en lugar de descartarlo, juzgándose más sencillo cambiarlo que comenzar desde cero, lo cual hubiese implicado realizar de nuevo toda la depuración que había pasado cada funcionalidad implementada. Durante el tiempo en el cual se realizó la reestructuración, el cuerpo de código del proyecto no solo creció, sino que disminuyó de forma considerable.

Iteración cuarta. Adaptación

Tras implementar la mayoría de las principales funcionalidades, el programa, después de la tercera iteración, se encuentra listo para un uso básico. Esta iteración se centró por tanto en funcionalidades nuevas no críticas, en la mejora del soporte para registro de mensajes («logging»), internacionalización , en la implementación de un comparador de versiones de invariantes y en la adaptación de la aplicación a cambios de *Takuan*.

Resumen de tareas realizadas durante esta iteración.

- Nuevo sistema de «Logging».
- Soporte para futuras traducciones mediante gettext.
- Comparación de invariantes.
- Mejora de la interfaz y corrección de errores.
- Adaptación de cambios en la estructura de *Takuan*.

Hasta mediados de la tercera iteración se empleó la instrucción de la biblioteca estándar de *Python* `print` para obtener información extra sobre la aplicación por la salida estándar de la misma. Cuando la salida creció y se le añadió información de depuración a la misma, se vio la necesidad de migrar a un sistema algo más robusto, que permitiese la salida o no de la información de depuración y errores según se tratase de una versión de desarrollo del programa o más general e hiciese que los mensajes mantuviesen una estructura, cierto orden y fuesen fácilmente trazables hasta su origen. Esta reordenación facilitó también el soporte para traducciones que se implementó a partir de la herramienta *gettext*.

Como parte de la reestructuración de la interfaz, se implementó la característica de comparación entre invariantes generados en el proyecto. Esta herramienta nueva permitía ver los efectos en los invariantes generados entre diferentes ejecuciones de los mismos casos de prueba unitarios *BPELUnit* o entre casos distintos y así contribuir a la mejora de los mismos.

Otra de las tareas a realizar durante esta iteración fue la reescritura y adaptación del guión *ant* principal llamado `base-build.xml` a la última versión de *Takuan*. El cambio de ruta en algunos componentes de la instalación o de nombre en algunos componentes *java*, dejaban rota la instalación de *IdiginBPEL*, la cual era incapaz de funcionar adecuadamente con versiones nuevas de *Takuan*. También se adaptó la inicialización del servidor *ActiveBPEL*, el cual, contando ahora con un nuevo guión *bash* de manejo que permitía su uso sencillo, se delegó el arranque y parada del servidor sobre este nuevo guión, que es invocado por *IdiginBPEL*, sin necesidad ahora de mantener actualizado *IdiginBPEL* ante posibles pequeños e incómodos cambios en la respuesta del servidor a la hora de inicializarlo, pararlo o comprobar su funcionamiento.

Quinta iteración. Documentación

Al encontrarse el programa en un nivel de madurez aceptable, esta quinta iteración se compone básicamente de actividades de documentación, depuración, prueba y solución de errores.

Resumen de tareas realizadas durante esta iteración.

- Autodocumentación del programa con *Doxygen*.
- Escritura de la memoria del proyecto.
- Prueba de la interfaz gráfica.
- Solución de errores tipográficos y visuales.

Durante el desarrollo mismo de la aplicación la documentación fue generándose conjunta la misma creación del código, siguiendo las indicaciones de la guía de estilo de *Python* para documentación, ya que el propio lenguaje permite incluir esta dentro de su mismo código. Se empleó aparte la herramienta *Doxygen* para poder generar una documentación aparte de forma automática. La escritura de la memoria del proyecto tuvo lugar entre junio y agosto de 2011. Esta fase del ciclo de vida se hubiese realizado antes de no haber mediado motivos laborales y una estancia en el extranjero (Reino Unido) durante el curso académico 2010-2011, lo que impidió su presentación.

3.4.1. Diagrama de Gantt

En este apartado se incluye el diagrama de *Gantt*, que puede verse en la imagen incluida en la figura 3.5 (página 40)[5], que describe la planificación del proyecto a lo largo de su tiempo de desarrollo. Puede verse el tiempo tomado por cada iteración (alrededor de 100 días) así como las tareas más problemáticas, que tomaron más tiempo.

[5]Debido a las restricciones de tamaño que el formato de este documento impone, el diagrama se muestra completo para poder apreciarse de forma general, y en varias ampliaciones en las figuras 3.6 (página 41), 3.7 (página 42) y 3.8 (página 43), en las que puede apreciarse la planificación con mayor detalle.

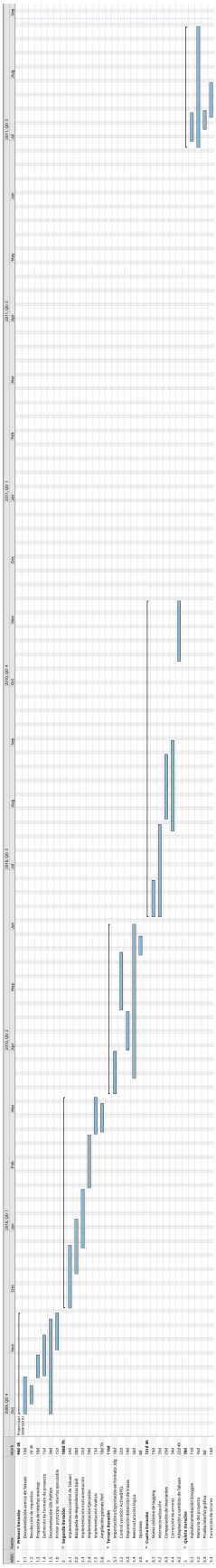


Figura 3.5: Diagrama de Gantt completo mostrando la planificación del proyecto

WBS	Name	Work
1	**Primera iteración**	**94d 4h**
1.1	Documentación acerca de Takuan	13d
1.2	Recolección de requisitos	7d 4h
1.3	Propuesta de interfaz mockup	10d
1.4	Definición de formato de proyecto	15d
1.5	Documentación Gtk Python	34d
1.6	Primer prototipo: Interfaz ejecutable	15d
2	**Segunda iteración**	**108d 7h**
2.1	Implementación control de Takuan	24d
2.2	Búsqueda de dependencias bpel	20d
2.3	Implementación Instrumentación	22d
2.4	Implementación Ejecución	19d
2.5	Implementación Análisis	13d
2.6	Ampliación guiones Perl	10d 7h
3	**Tercera iteración**	**116d**
3.1	Importación y Exportación en formato .idg	16d
3.2	Control servidor ActiveBPEL	22d
3.3	Depuración detección de trazas	14d
3.4	Reestructuración lógica	56d
3.5	Opciones	8d
4	**Cuarta iteración**	**131d 4h**
4.1	Nuevo sistema de logging	15d
4.2	Internacionalización	35d
4.3	Comparación de invariantes	25d
4.4	Corrección de errores	34d
4.5	Adaptación a cambios de Takuan	22d 4h
5	**Quinta iteración**	**78d**
5.1	Autodocumentación Doxygen	11d
5.2	Memoria del proyecto	45d
5.3	Prueba interfaz gráfica	8d
5.4	Corrección de errores	14d

Figura 3.6: Ampliación de la primera parte del diagrama de Gantt 3.5 de la página 40 (1/3)

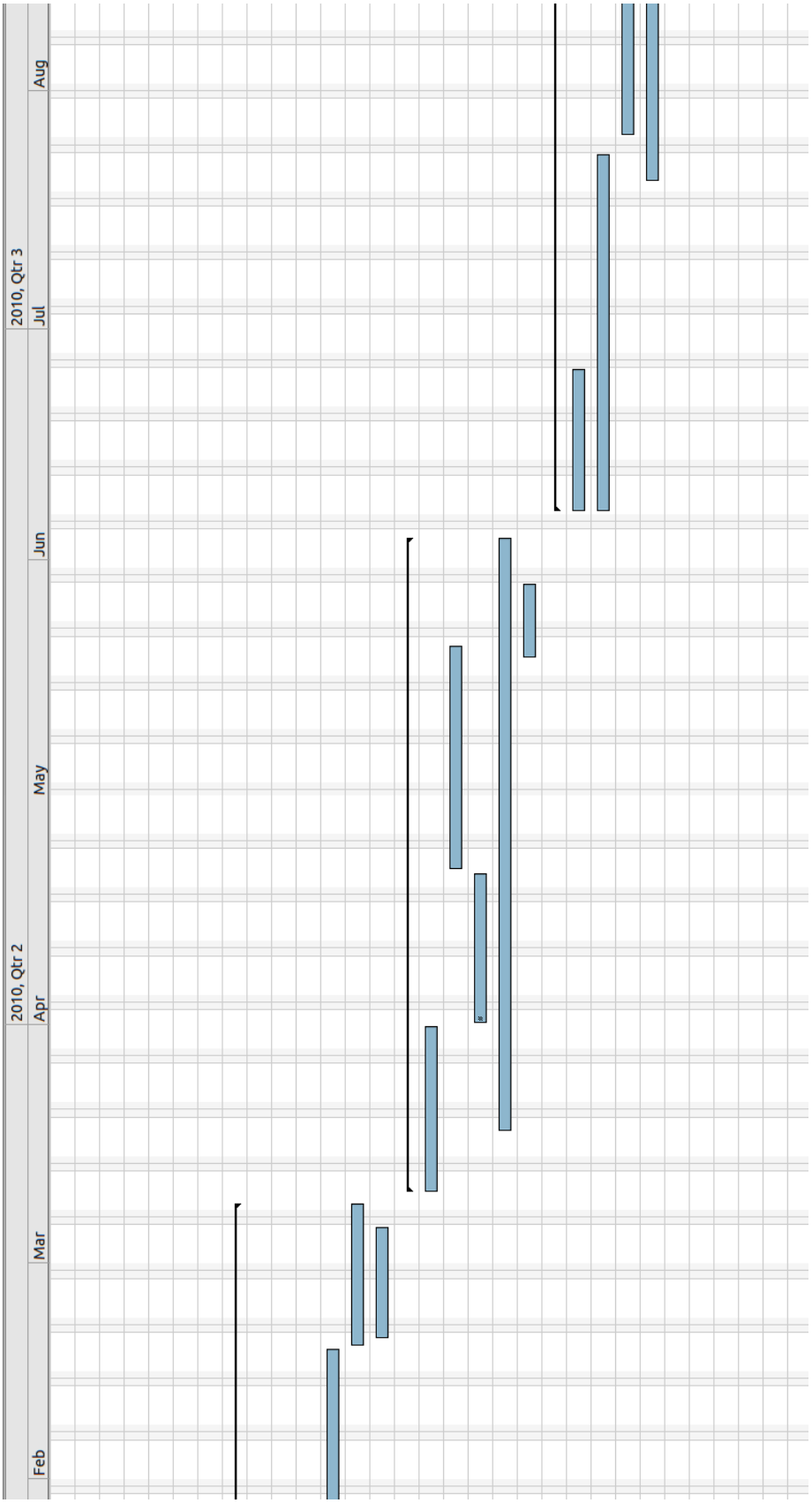


Figura 3.7: Ampliación de la segunda parte del diagrama de Gantt 3.5 de la página 40 (2/3)

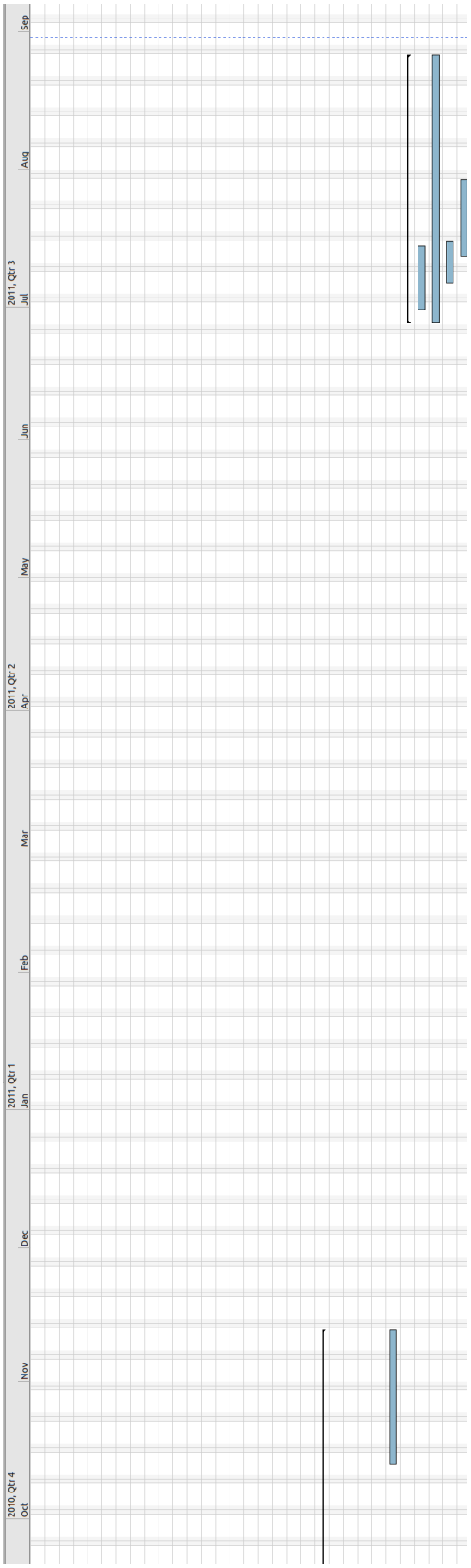


Figura 3.8: Ampliación de la tercera parte del diagrama de Gantt 3.5 de la página 40

CAPÍTULO 4

Análisis y Diseño

4.1. Requisitos

Esta sección recoge los requisitos funcionales y no funcionales recogidos durante la especificación del sistema a desarrollar. La recogida de requisitos se realizó mediante una serie entrevistas personales, en las cuales se utilizaron propuestas y prototipos en papel a partir de los cuales se decidieron los datos que debian procesarse y cómo. Algunos requisitos, (como la comparación de invariantes) fueron añadidos en iteraciones posteriores, tras ver el progreso de *IdiginBPEL* y concebir nuevas necesidades.

4.1.1. Requisitos funcionales

Requisitos funcionales de la aplicación, visibles en la tabla 4.1 (página 46). Estos requisitos establecen el comportamiento interno del sistema, definiendo qué debe hacer exactamente. Los casos de uso visibles en la sección 4.2.1 definen la funcionalidad necesaria para el cumplimiento de estos requisitos. Cada uno de ellos incluye un nombre que lo identifica y un resumen explicativo.

4.1.2. Requisitos no funcionales

Requisitos no funcionales de la aplicación, recogidos en la tabla 4.2 (página 47). Estos requisitos no funcionales (o atributos de calidad) especifican el comportamiento específico y otras indicaciones sobre la utilización del sistema sin definir funcionalidades concretas de la aplicación.

Buscar de dependencias de código *WS-BPEL 2.0*	A partir de un fichero de código *.bpel* el sistema debe ser capaz de inferir sus dependencias de forma recursiva (en caso de que haya más de un nivel de dependencias) y ser capaz de encontrarlas y manejarlas
Manejar suites de prueba *BPELUnit* de forma genérica	A partir de múltiples suites de casos de prueba unitarios *BPELUnit*, el sistema debe ser capaz de manejar casos de prueba individuales de forma independiente.
Ejecución de casos de prueba *BPELUnit*	El sistema debe ser capaz de controlar la ejecución de casos de prueba unitarios *BPELUnit* contra un servidor web dado.
Recolección de registros de ejecución *ActiveBPEL*	El servidor *ActiveBPEL* adaptado a *Takuan* genera registros de ejecución de cada caso de prueba unitario *BPELUnit* que se ejecuta contra el. El sistema debe ser capaz de recolectar tales registros de ejecución y manejarlos.
Ejecución en *Daikon*	El sistema manejará la transformación los registros de ejecución de casos de prueba unitarios *BPELUnit* manejados por la aplicación al formato propio de *Daikon* y controlará la ejecución de este último.
Comparación de invariantes	Los invariantes manejados por la aplicación deben poder ser comparables de forma que el sistema muestre sus diferencias.
Sistema de proyectos	El sistema se encontrará orientado a proyectos, almacenando toda la información relativa al estudio de una composición *WS-BPEL 2.0* de forma independiente.
Exportación de proyectos	Los proyectos generados con la herramienta deben poder ser exportados a un sistema externo.
Importación de proyectos	Los proyectos generados por otras instancias de *IdiginBPEL* y exportados por ellas deben poder ser incluidos en otra cualquiera mediante un proceso de importación.

Tabla 4.1: Requisitos funcionales del sistema.

Visualización Gráfica	El proceso completo de *Takuan* debe verse reflejado de forma gráfica mediante una interfaz de usuario con ventanas y usable a través el ratón.
Muestra de ayuda	El sistema debe tener texto de ayuda disponible en alguna parte del mismo de forma que sea bien visible.
Independencia de los proyectos	Los proyectos realizados por la aplicación deben ser independientes de la instancia de *IdiginBPEL* en la cual sean generados.
Internacionalización	La aplicación debe estar al menos en idioma inglés y tener soporte para su expansión a otras lenguas.
Gestión gráfica de casos de prueba	Los casos de prueba incluidos en la aplicación para ser ejecutados contra el servidor deben poder ser visibles y manejables por el usuario. El usuario debe poder elegir que casos concretos entran en la ejecución.
Gestión gráfica de dependencias rotas *WS-BPEL 2.0*	Las dependencias de un fichero *WS-BPEL 2.0* que no se encuentren disponibles deben marcarse como tales y aparecer en la interfaz como rotas.
Gestión gráfica de registros de ejecución	Los registros de ejecución obtenidos y almacenados por el sistema deben ser mostrados de forma gráfica, identificados por proveniencia de forma que el usuario pueda elegir cuales entran en el análisis.

Tabla 4.2: Requisitos no funcionales del sistema.

4.2. Análisis del sistema

4.2.1. Casos de uso

La imagen de la figura 4.1 muestra el diagrama de casos de uso empleado por *IdiginBPEL*. Estos casos de uso recogen la funcionalidad básica a realizar por el sistema y definen las posibles interacciones con los usuarios del mismo [Arl05].

La herramienta empleada para la realización de todos los esquemas y diagramas mostrados en este documento es *Bouml*, de la cual puede leerse más en la sección 3.1.6 (página 26).

Caso de uso: Crear Proyecto

Descripción: El *Sistema* crea un nuevo proyecto *IdiginBPEL*.
Actor principal: Usuario.
Precondiciones: Ninguna.

Flujo principal:

1. El *Usuario* selecciona en la interfaz el crear un nuevo proyecto.
2. El *Usuario* introduce un nombre *name*, identificativo para el proyecto.
3. El *Sistema* lo comprueba el identificador *name*, y lo valida.
4. El *Usuario* indica la ruta al fichero *WS-BPEL 2.0 path*, con la composición en torno al estudio de la cual girará el proyecto a crear.
5. El *Sistema* comprueba la ruta *path* y la valida.
6. El *Sistema* importa el fichero `.bpel` disponible en *path* junto a sus dependencias, y crea un nuevo proyecto *P* a partir de una plantilla.
7. El *Sistema* realiza el proceso de instrumentación sobre los ficheros fuente disponibles en el proyecto *P* recientemente importados durante la creación del proyecto.
8. El *Sistema* muestra un resumen del proyecto *P*, incluyendo los ficheros listados en él y su estado.

Postcondiciones:

1. Un nuevo proyecto *P* con nombre *name* ha sido creado por el *Usuario*con el fin de analizar el fichero de código fuente `.bpel` disponible en en la ruta *path*.

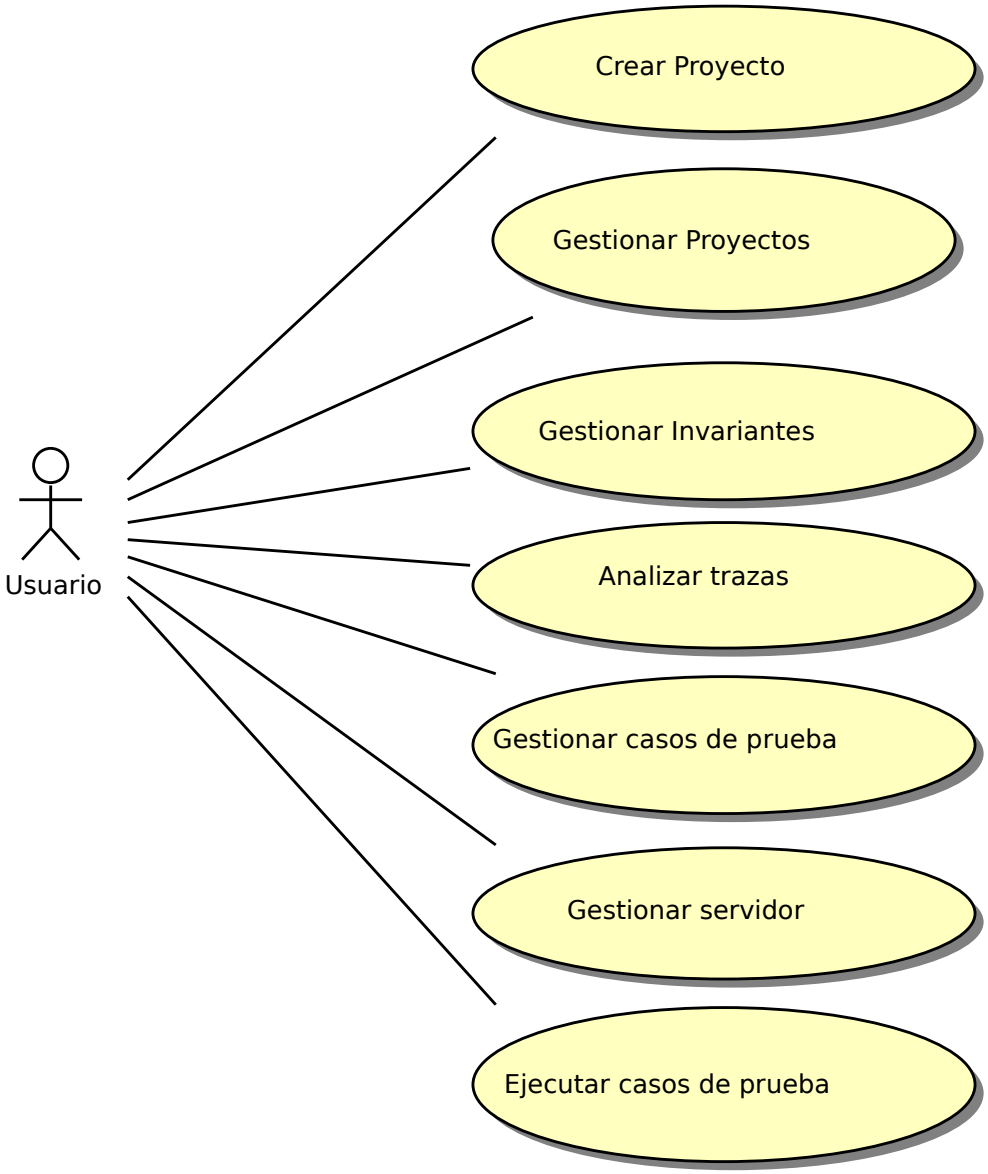


Figura 4.1: Diagrama de casos de uso de la aplicación

Flujos alternativos

Flujo Alternativo: Nombre de proyecto inválido.
Descripción: El valor nombre *name* del proyecto no puede usarse para la creación del proyecto debido a encontrarse vacío, a la presencia de caracteres inválidos o a ser demasiado largo.
Precondiciones:

1. El valor nombre de proyecto *name* ya existe o contiene caracteres considerados inválidos.

1. El flujo alternativo comienza en el paso **3.** del flujo principal.
2. El *Sistema* informa al *Usuario* de que el valor *name* del proyecto no es válido y la razón del mismo, de forma que el *Usuario* pueda volver a introducirlo tras solucionar el problema detectado.

Flujo Alternativo: Dependencias rotas.
Descripción: En el momento de la importación del fichero *WS-BPEL 2.0*, durante la búsqueda de dependencias, una o más dependencias no pudieron ser encontradas o importadas al proyecto.
Precondiciones:

1. Existe una creación de proyecto *P* en marcha.
2. El fichero *WS-BPEL 2.0* a analizar tiene dependencias rotas.

1. El flujo alternativo comienza en el paso **6.** del flujo principal.
2. El *Sistema* informa al *Usuario* de que el fichero *WS-BPEL 2.0* sobre el cual se está construyendo el fichero tiene dependencias rotas. Aún así, crea el proyecto y muestra en la interfaz todas las dependencias del proyecto *P*, indicando claramente cuales están rotas y cuales no.

Flujo Alternativo: Ruta al fichero *WS-BPEL 2.0* inválida.
Descripción: El valor *path* con la ruta especificada por el usuario al fichero *WS-BPEL 2.0* a importar para la creación del proyecto es inválida.
Precondiciones:

1. La ruta *path* suministrada al fichero *WS-BPEL 2.0* para iniciar el proyecto no es válida.

Flujo Alternativo:

1. El flujo alternativo comienza en el paso **5.** del flujo principal.
2. El *Sistema* informa al *Usuario* de que la ruta *path* del fichero a añadir al proyecto no es válido y la razón del mismo. El *Usuario* deberá volver a introducir la ruta.

Flujo Alternativo: Importación de proyecto.
Descripción: El *Usuario* especifica importar un proyecto *E* en lugar de crearlo desde cero.
Precondiciones: Ninguna.

Flujo Alternativo:

1. El flujo alternativo comienza en el paso **1.** del flujo principal.
2. El *Usuario* selecciona la opción importación de proyecto.
3. El *Usuario* suministra una ruta *path* a un fichero de formato `.idg`.
4. El *Sistema* desempaqueta el fichero y crea el proyecto *E*.

Caso de uso: Gestionar Proyectos

Descripción: El *Sistema* borra un proyecto *P*, previamente existente de *IdiginBPEL*.
Actor principal: Usuario.
Precondiciones:

1. Existe al menos un proyecto *P* en la instancia actual de *IdiginBPEL*.

Flujo principal:

1. El *Usuario* selecciona un proyecto existente *P*.
2. El *Usuario* selecciona la opción *Borrar Proyecto*.
3. El *Sistema* comprueba la selección del proyecto.
4. El *Sistema* borra el proyecto *P*.

Postcondiciones:

1. El proyecto *P* previamente seleccionado ya no existe; ha sido completamente eliminado.

Flujos alternativos

Flujo Alternativo: El proyecto se encuentra abierto.
Descripción: El proyecto *P* a borrar se encuentra abierto en el momento del borrado y es necesario cerrarlo antes.
Precondiciones:

1. El proyecto seleccionado *P* a borrar se encuentra abierto.

1. El flujo alternativo comienza en el paso **3.** del flujo principal.
2. El *Sistema* cierra el proyecto *P* y continúa el proceso.

Flujo Alternativo: Exportación de proyecto.
Descripción: El *Usuario* decide exportar el proyecto *P*.
Precondiciones:

1. Existe al menos un proyecto *P* ya creado.

Flujo Alternativo:

1. El flujo alternativo comienza en el paso **2.** del flujo principal.
2. El *Usuario* selecciona la opción *Exportar Proyecto*.
3. El *Usuario* selecciona un nombre *name* y una ruta *path* para la exportación.
4. El *Sistema* empaqueta el proyecto y lo dispone con nombre *name* en el directorio indicado por el valor de ruta *path*.

Caso de uso: Gestionar Invariantes

Descripción: El *Sistema* visualiza un invariante.
Actor principal: Usuario.
Precondiciones:

1. Existe al menos un proyecto *P* y este se encuentra abierto.
2. Existe al menos un invariante *I* ya generado en el proyecto.

Flujo principal:

1. El *Usuario* selecciona un invariante *I*.
2. El *Sistema* carga el invariante *I* previamente seleccionado y lo muestra en pantalla.

Postcondiciones:

1. Se puede ver el invariante seleccionado *I* cargado en el visualizador.

Flujos alternativos

Flujo Alternativo: Comparación de invariantes.
Descripción: El *Usuario* compara dos invariantes entre sí.
Precondiciones:

1. Existen al menos dos invariantes (*I* y *J*) en la instancia abierta del proyecto *P*.
2. Ya existe un invariante seleccionado (*I*) que está siendo visualizado.

1. El flujo alternativo comienza en el paso **2.** del flujo principal.
2. El *Usuario* entra en el llamado modo de *Comparación* pulsando el botón *Comparar*
3. El *Sistema* entra en modo de comparación.
4. El *Usuario* selecciona un segundo invariante *J* en el listado de invariantes disponible, en el cual no se muestra el invariante previamente seleccionado *I*.
5. El *Sistema* calcula las diferencias entre el invariante actualmente cargado (*I*) y el nuevo invariante seleccionado (*J*), muestra ambos, marcando donde difieren.

Caso de uso: Analizar Trazas

Descripción: El *Sistema* envía los registros de ejecución al analizador *Daikon*.
Actor principal: Usuario.
Precondiciones:

1. Existe al menos un proyecto *P* y este se encuentra abierto.
2. Existe al menos un registro de ejecución *R* ya generado en este proyecto.

Flujo principal:

1. El *Sistema* muestra todos los registros de ejecución disponibles.
2. El *Usuario* selecciona uno o más registros de ejecución (*SEL*).
3. El *Sistema* comprueba la selección (*SEL*) y comienza el proceso de transformación y análisis de los registros de ejecución seleccionados.

Postcondiciones: Ninguna.

Flujos alternativos

Flujo Alternativo: Selección inválida.
Descripción: El *Usuario* no ha realizado una selección vacía.
Precondiciones:

1. Existe al menos una traza en el listado de trazas disponibles seleccionada por el usuario *SEL*.

1. El flujo alternativo comienza en el paso **3.** del flujo principal.
2. El *Sistema* avisa al usuario de que debe seleccionar al menos un registro de ejecución e impide la ejecución del análisis sin registros de ejecución.

Caso de uso: Gestionar casos de prueba

Descripción: El *Usuario* añade una suite de casos de prueba unitarios *BPELUnit*.
Actor principal: Usuario.
Precondiciones:

1. Existe al menos un proyecto *P* y este se encuentra abierto.

Flujo principal:

1. El *Usuario* introduce la ruta (*path*) al fichero `.bpts` correspondiente a una suite de casos de prueba unitarios *BPELUnit*.
2. El *Sistema* la valida y lee el fichero de la suite, detecta los casos de prueba unitarios considerados individualmente al sistema y los añade al proyecto.
3. El *Sistema* lista los casos de prueba disponibles.

Postcondiciones:

1. Los casos de prueba unitarios pertenecientes a la suite *BPELUnit* incluida en el fichero `bpts` en la ruta *path* son añadidos y se encuentran ahora disponibles para ser seleccionados por el usuario y entrar así en ejecución.

Flujos alternativos

Flujo Alternativo: Fichero `.bpts` inválido.
Descripción: El *Usuario* ha seleccionado un fichero de código fuente `.bpts` corrupto o mal formado, que no puede ser empleado por la aplicación.
Precondiciones: Ninguna.

1. El flujo alternativo comienza en el paso **2.** del flujo principal.
2. El *Sistema* avisa al usuario de que debe seleccionar un fichero válido.

Caso de uso: Gestionar Servidor

Descripción: El *Usuario* inicia el servidor *ActiveBPEL*.
Actor principal: Usuario.
Precondiciones:

1. El servidor *ActiveBPEL* se encuentra instalado el sistema junto al resto de la instalación de *Takuan*.

Flujo principal:

1. El *Usuario* activa el botón de *Iniciar servidor* ActiveBPEL.
2. El *Sistema* comprueba si el servidor *ActiveBPEL* no estaba previamente activo y lo inicia.

Postcondiciones:

1. El servidor *ActiveBPEL* se encuentra funcionando y su estado es de «Running».

Flujos alternativos

Flujo Alternativo: Detener servidor *ActiveBPEL.*
Descripción: El *Usuario* detiene el servidor *ActiveBPEL.*
Precondiciones:

1. El servidor *ActiveBPEL* se encuentra en estado «Running».

1. El flujo alternativo comienza en el paso **2.** del flujo principal.
2. El *Sistema* comprueba si el servidor *ActiveBPEL* no se encontraba ya previamente detenido y le envía la orden de parada.

Flujo Alternativo: Comprobar servidor *ActiveBPEL.*
Descripción: El *Usuario* comprueba el estado del servidor *ActiveBPEL.*
Precondiciones: Ninguna.

1. El flujo alternativo comienza en el paso **1.** del flujo principal.
2. El *Usuario* pulsa el botón de *Comprobar Servidor* ActiveBPEL.
3. El *Sistema* actualiza el estado del servidor en la interfaz de la aplicación a «Running» o «Stopped».

Caso de uso: Ejecutar Casos de Prueba

Descripción: El *Sistema* ejecuta casos de prueba *BPELUnit*.
Actor principal: Usuario.
Precondiciones:

1. El servidor *ActiveBPEL* se encuentra en estado «Running».
2. Existe al menos un caso de prueba unitario *BPELUnit* listado como disponible por la aplicación.

Flujo principal:

1. El *Usuario* realiza una selección de casos de prueba unitarios *BPELUnit* (*SEL*) y pulsa el botón *Ejecutar*.
2. El *Sistema* comprueba la selección de casos de prueba unitarios *BPELUnit* (*SEL*) e inicia la ejecución de esos casos.

Postcondiciones:

1. El servidor *ActiveBPEL* se encuentra funcionando y su estado es de «Running».

Flujos alternativos

Flujo Alternativo: Selección inválida de casos de prueba unitarios *BPELUnit*.
Descripción: El *Usuario* ha realizado una selección incorrecta de casos de prueba unitarios *BPELUnit*.
Precondiciones: Ninguna.

1. El flujo alternativo comienza en el paso **2.** del flujo principal.
2. El *Sistema* avisa de que la selección *SEL* no es correcta e impide la ejecución.

Flujo Alternativo: Detener la ejecución.
Descripción: El *Usuario* detiene la ejecución de los casos de prueba unitarios *BPELUnit* en el servidor *ActiveBPEL*.
Precondiciones:

1. Existe una ejecución en marcha.

1. El flujo alternativo comienza en cualquier momento del flujo principal.
2. El *Usuario* pulsa el botón de *Detener Ejecución*.
3. El *Sistema* detiene la ejecución en el momento.

4.2.2. Modelo conceptual de datos

En el diagrama que puede verse en la figura 4.2 disponible en la página 58 figura el modelo conceptual de datos de la aplicación. En él pueden observarse la parte fundamental de la capa lógica de la aplicación, sobre la que recaen las responsabilidades de mantener disponible la información y realizar las operaciones funcionales del programa.

La estructura de clases que puede verse en el esquema 4.2 muestra como la clase `Idg` mantiene la estructura lógica principal de la aplicación, así como las opciones y la configuración general. Cada clase `Idg` puede mantener hasta un proyecto en abierto, el cual se representa en la clase `Project`. Esta clase mantiene toda la información relacionada con el proyecto en proceso de edición. Lista dependencias, casos de prueba, invariantes, trazas...

La clase de `Opt` mantiene los datos básicos de programa accesible para todos los módulos. Estos son leídos desde fichero en el arranque y son susceptibles de ser modificados y guardados por el usuario durante la ejecución de la aplicación.

Cada diferente clase de fichero que el programa debe manejar tiene su propia clase, datos y funcionalidad. Esto es, ficheros de invariantes, suites de prueba *BPELUnit*, ficheros de configuración, ficheros *ANT*... Están contenidos en el proyecto al cual pertenecen, y cada uno de ellos posee su propia clase representativa.

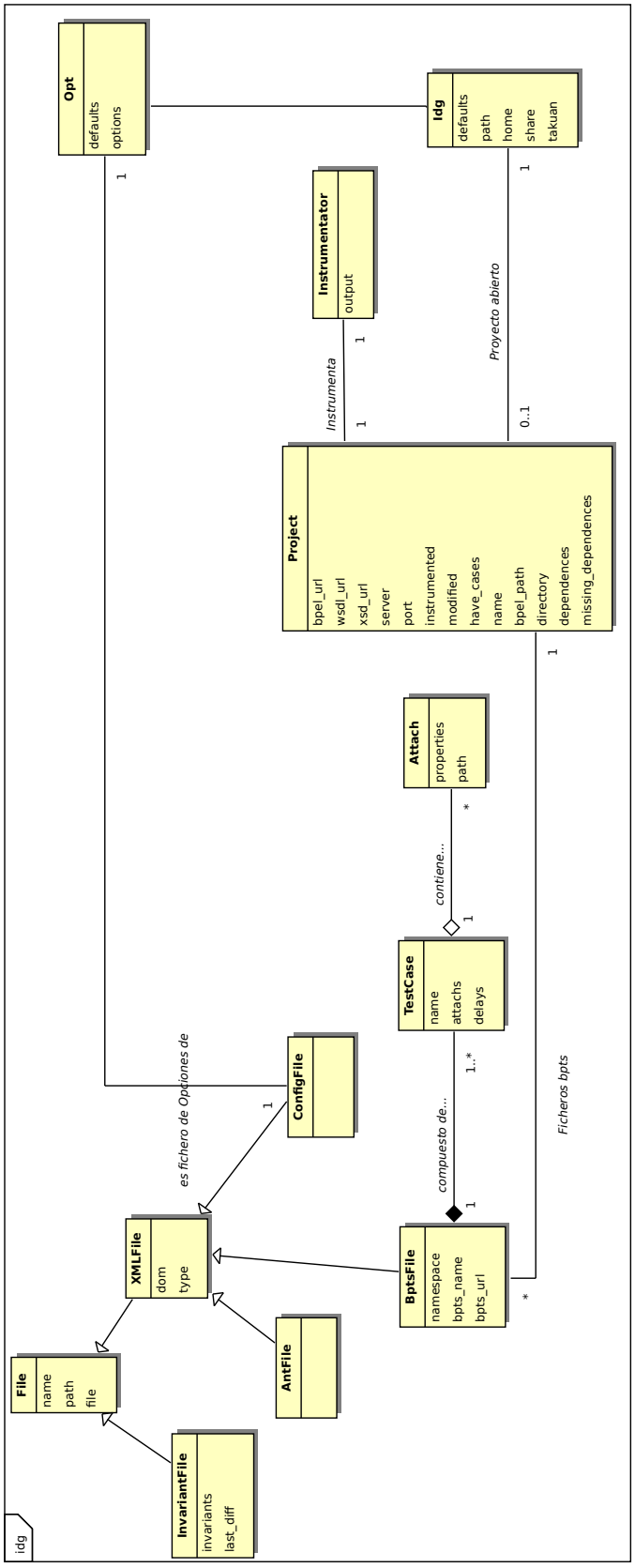


Figura 4.2: Diagrama de modelo conceptual de datos

- **Invariants File**: Clase que maneja un fichero con la salida de invariantes producida por *Daikon*.
- **Bpts File**: Clase que maneja una suite de casos de prueba unitarios *BPELUnit*. Estas suites se dividen a su vez en uno o más **TestCase**s, los cuales, según la composición, opcionalmente pueden incluir **Attachments**.
- **Config File**: Manejo de los ficheros de configuración propios de *IdiginBPEL*.
- **Ant File**: Manejo de ficheros *xml* que conforman guiones *ANT*.

La clase `Project` mantiene las muchas o ninguna suites *BPELUnit* que se encuentren asociadas al proyecto. De forma parecida, la clase `Opt` mantiene una referencia al fichero de configuración de la aplicación, en el cual delega la lectura y escritura de las opciones modificadas durante la ejecución de la aplicación.

4.2.3. Diagramas de secuencia

En esta sección se encuentran los diagramas de secuencia del sistema. Estos diagramas ilustran la sucesión de eventos, enviados por los usuarios del sistema, y las respuestas que el sistema realiza en cada caso [Arl05].

4.2.4. Diseño

En la siguiente sección es presentado el modelo de datos de diseño. Este incluye, a diferencia del diagrama de modelo conceptual (Figura 4.2, página 58), las operaciones disponibles en cada clase así como las clases reales empleadas en la implementación, tales como las de manejo de errores y las relacionadas con la interfaz gráfica y la interacción con el usuario.

Refiriéndonos a la sección lógica, se aplicó un patrón de *Fachada* (*Facade* [Gam94]) a partir de la clase `Idg`, la cual provee una interfaz de acceso al resto de las clases, que dependen de ella para la obtención de opciones mediante el acceso a la clase `Opt`, traducciones, configuración del módulo de «logging» y otras. La mayor parte de la funcionalidad de la aplicación recae sobre la clase `Proyecto`, la cual realiza la mayoría de las operaciones necesarias relacionadas con la gestión de uno proyecto. Esto incluye su creacion, la gestión de configuración y opciones, manejo de ficheros asociados, y gran parte del control del proceso o flujo de *Takuan*.

La jerarquía de clases `File`, `InvFile`, `XMLFile`, `ANTFile`, `BPTSFile`, `ConfigFile` representan cada una el manejo de cada tipo de fichero al cual representan y mantienen la responsabilidad de realizar las operaciones propias de cada tipo. La clase `File` representa a la vez la interfaz a seguir por el resto de ficheros e implementa la funcionalidad básica de un fichero de proyecto. La clase `XMLFile`, la cual hereda de `File`, añade a esta el manejo genérico de ficheros *xml*, incluyendo operaciones concretas de ficheros de código basado en

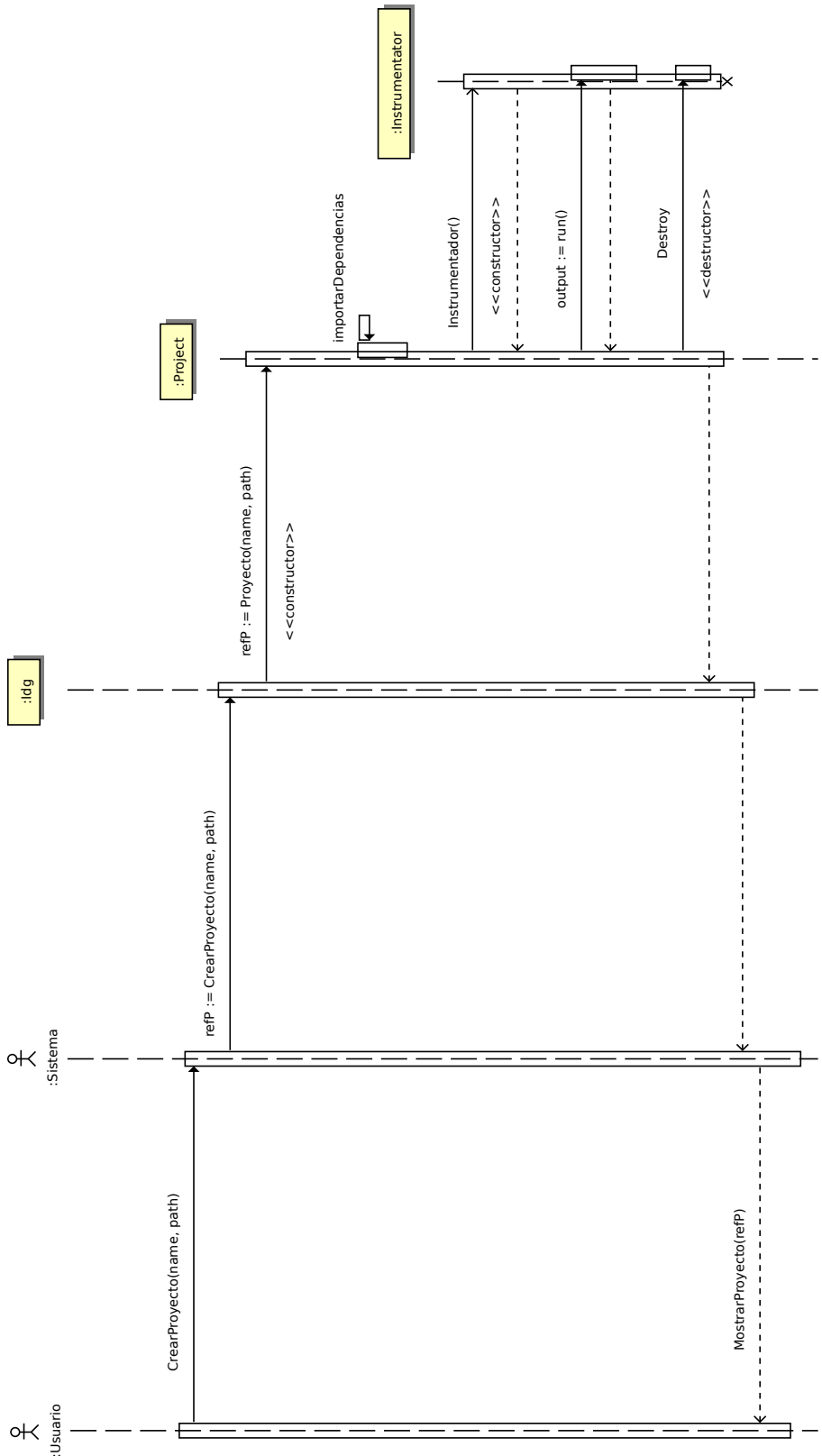


Figura 4.3: Diagrama de secuencia del sistema para el caso de uso *Crear Proyecto*

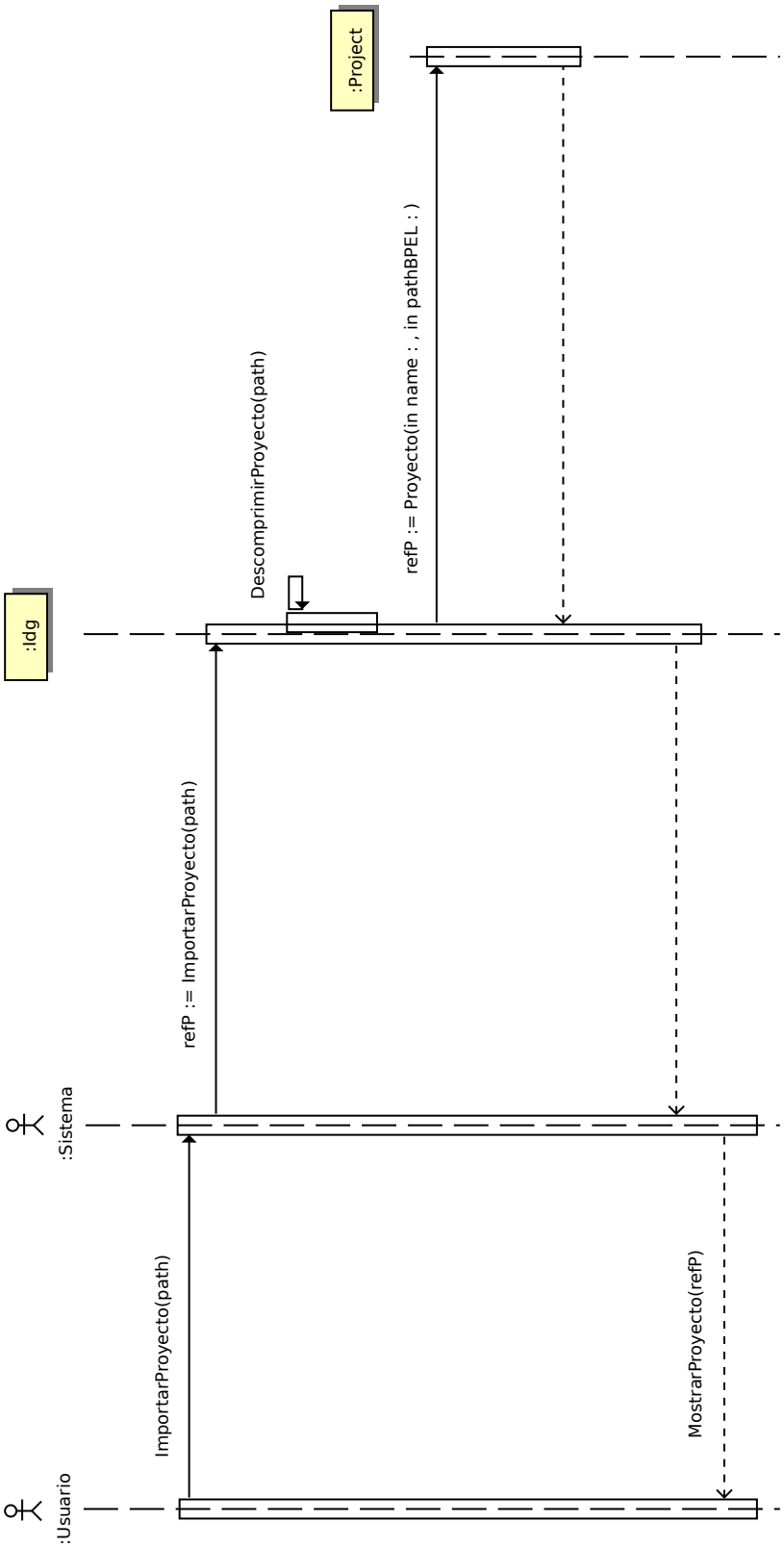


Figura 4.4: Diagrama de secuencia del sistema para el flujo alternativo *Importar Proyecto* del caso de uso *Crear Proyecto*

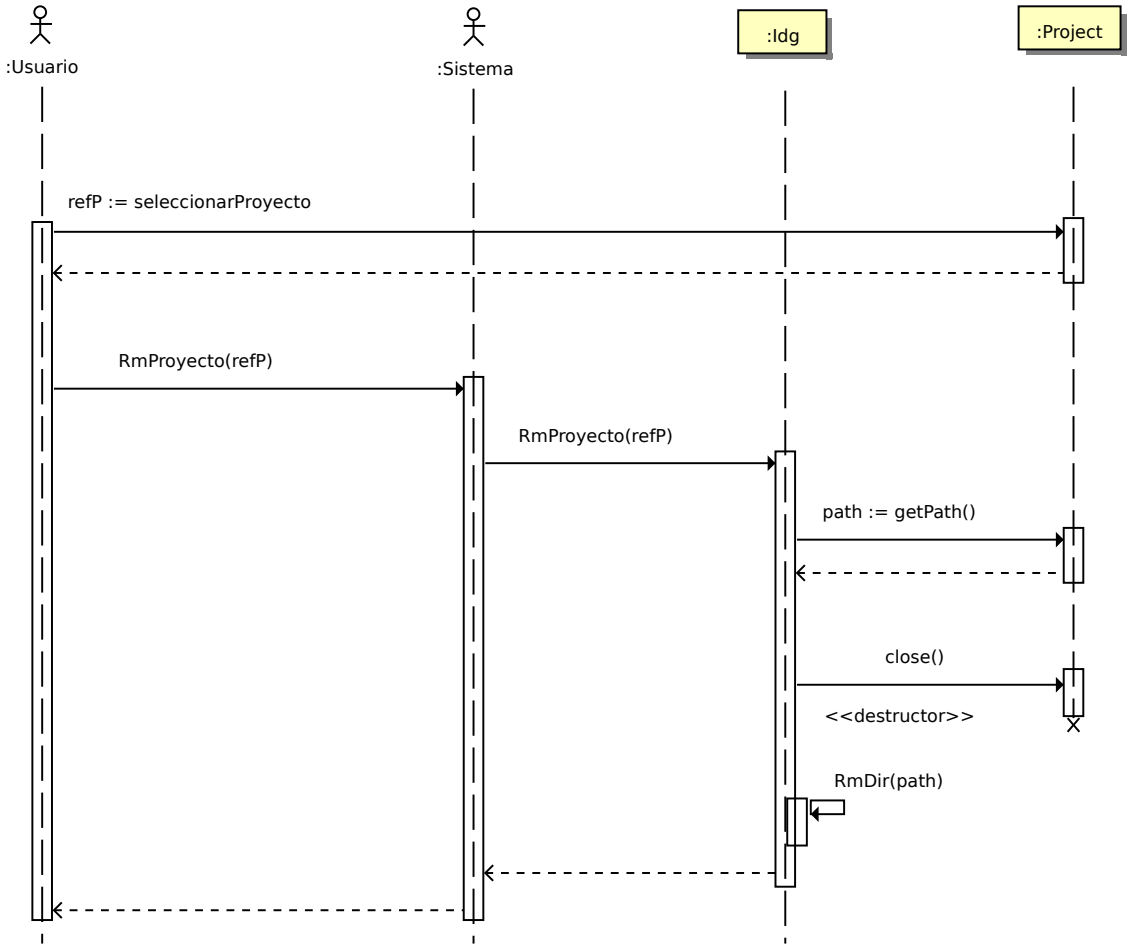


Figura 4.5: Diagrama de secuencia del sistema para *Borrar Proyecto* del caso de uso *Gestionar Proyectos*

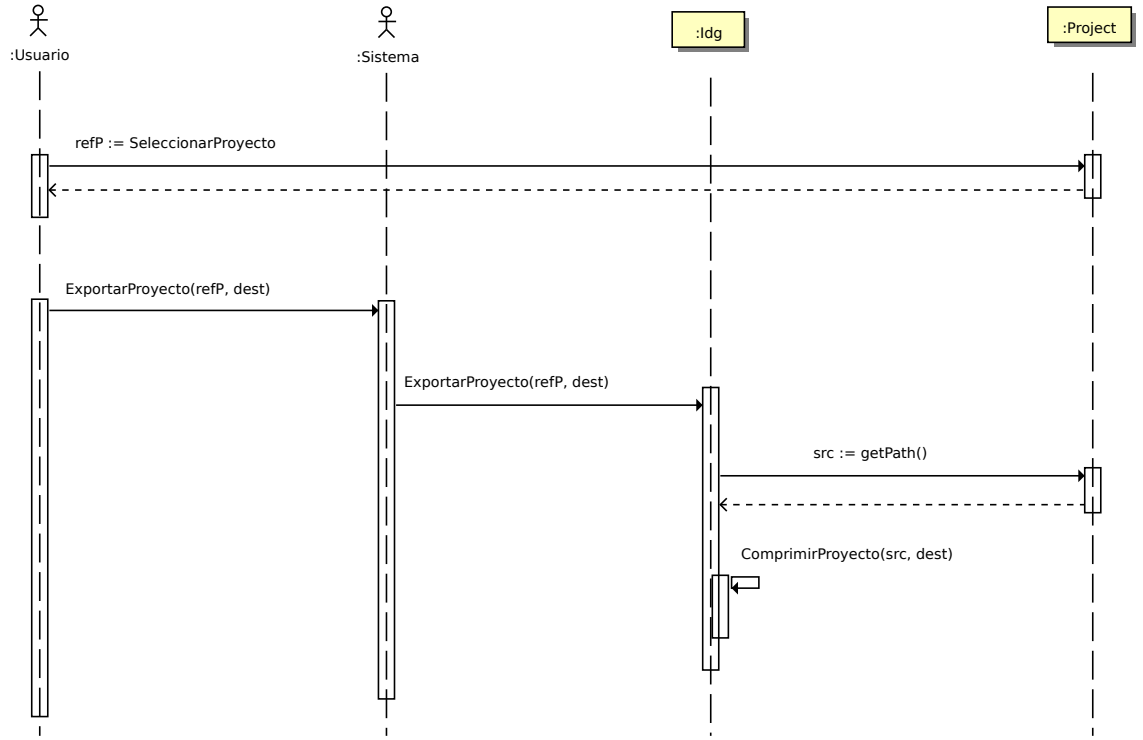


Figura 4.6: Diagrama de secuencia del sistema para el flujo alternativo *Exportar Proyecto* del caso de uso *Gestionar Proyectos*

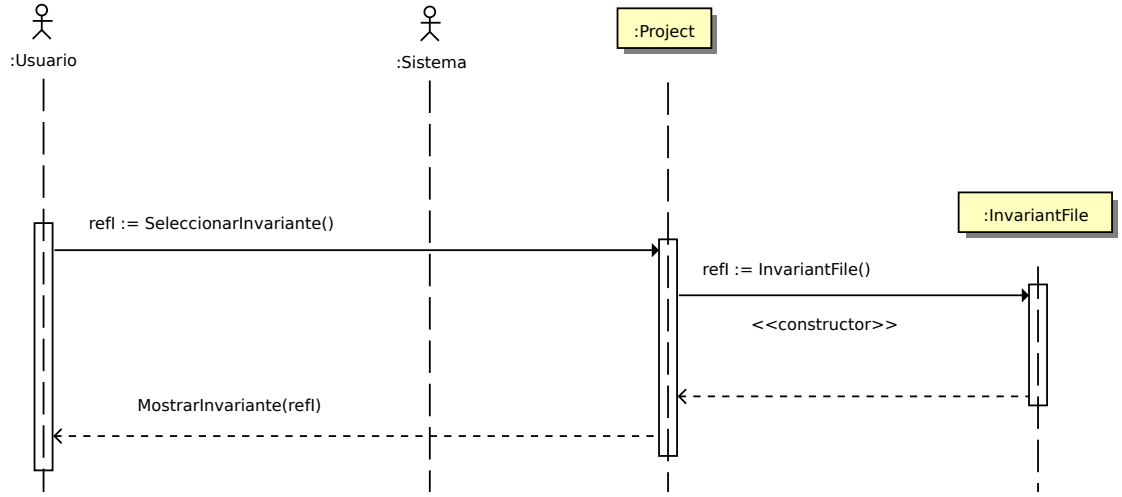


Figura 4.7: Diagrama de secuencia del sistema para *Visualizar Invariante* del caso de uso *Gestionar Invariantes*

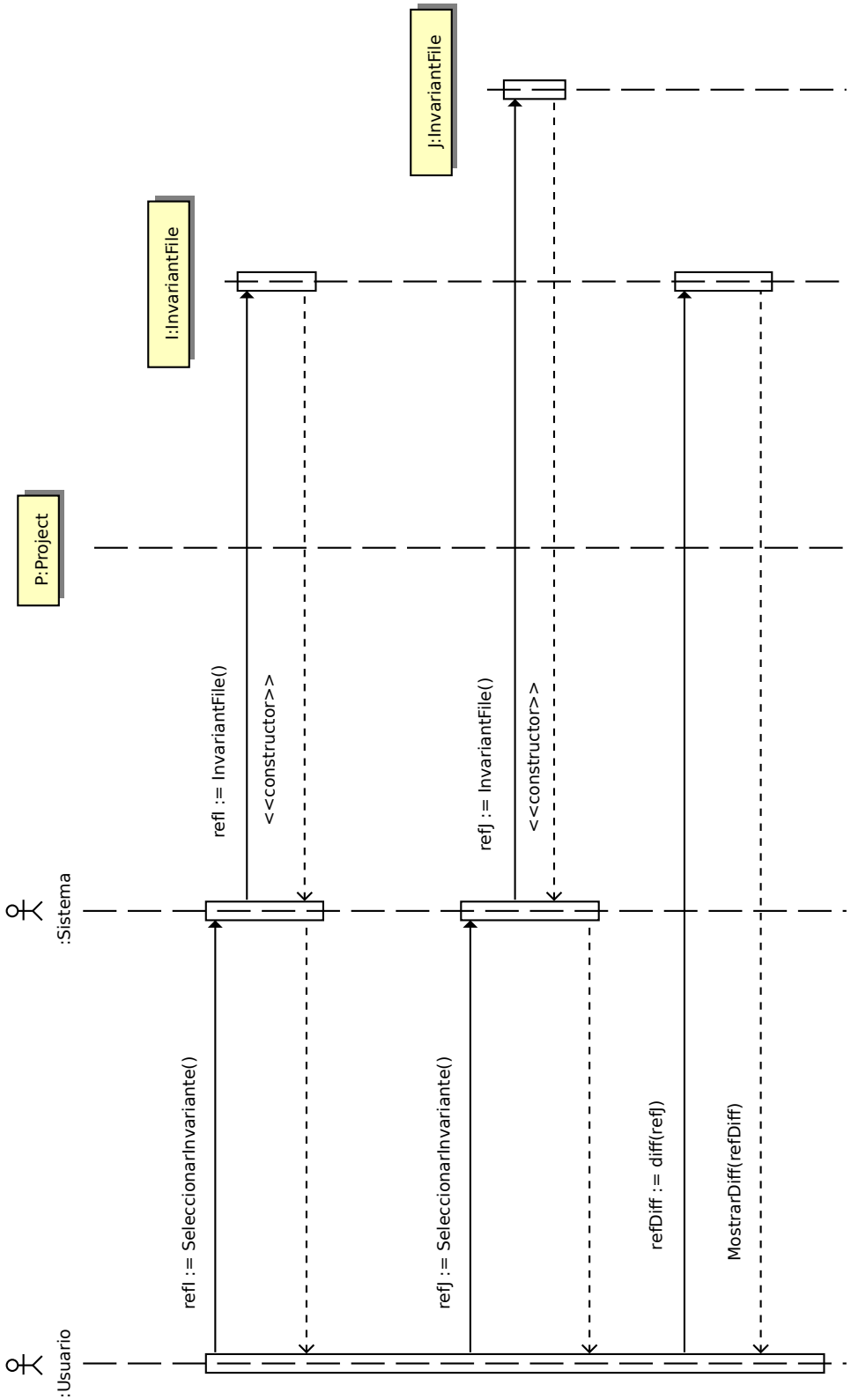


Figura 4.8: Diagrama de secuencia del sistema para el flujo alternativo *Comparar Invariantes* del caso de uso *Gestionar Invariantes*

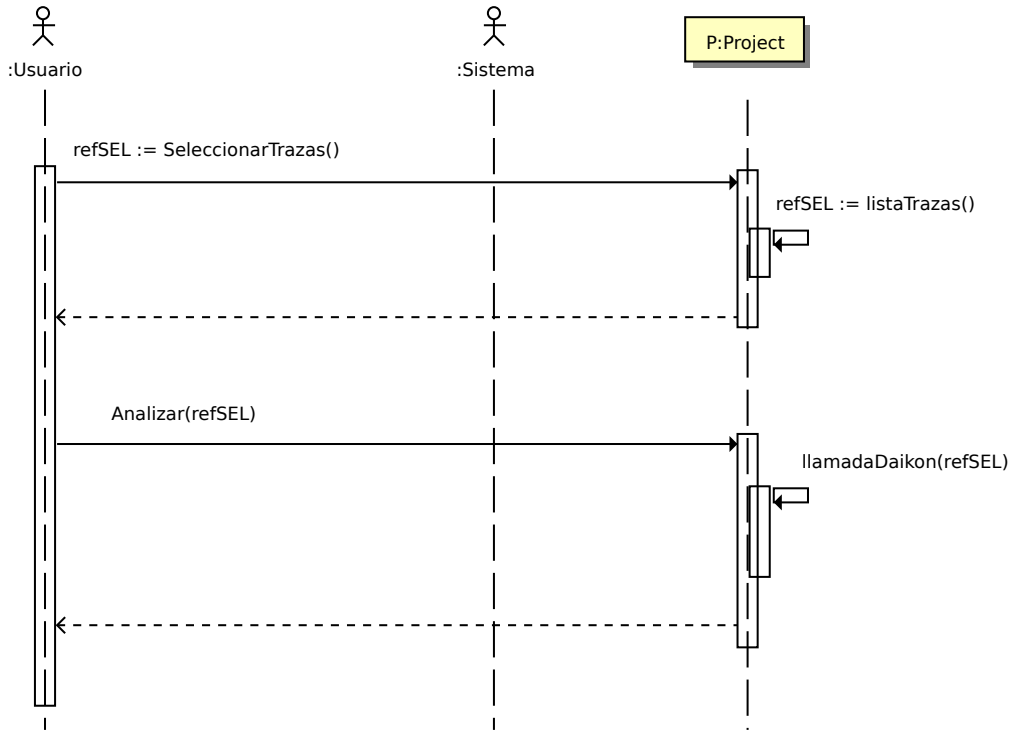


Figura 4.9: Diagrama de secuencia del sistema para el caso de uso *Analizar Trazas*

el estándar *xml*. Estas clases evitan la repetición de código muy prolijo, que requiere de la comprobación de muchos errores y captura de diversas excepciones, y muy proclive a fallos, como es la apertura y parseo de ficheros *xml*.

La operación con ficheros de tipo suite de prueba *BPELUnit* y el uso y manejo de casos de prueba unitarios a nivel individual, hizo necesaria la creación de un sistema de clases propio que hiciese posible su manejo de forma genérica. Esto se ve reflejado en la creación de la clase `BPTSFile` la cual representa a una suite y realiza operaciones genéricas con los casos de prueba que contiene, dotándolos de identificación individual y manejando sus adjuntos o *Attachments*.

La operación con ficheros de invariantes, esto es, la salida del analizador estadístico *Daikon*, está representada por la clase `InvFile`. La salida de *Daikon* no se realiza en ningún formato de serialización como *xml*, de forma que la clase adquiere la responsabilidad de parsear la información y obtener los invariantes generados en una estructura de fácil manejo. Aparte, siendo esta clase la que mantiene la información de los invariantes de un fichero, adquiere también la operación `diff`, la cual permite la comparación de dos ficheros de invariantes entre si, generando una estructura intermedia, con los datos de las diferencias.

El paquete `idgui` el cual implementa la interfaz gráfica, sigue un patrón *Observador* (*Observer* [Gam94]). La responsabilidad en gran parte de la clase `ProyectoUI` es la de observar cambios en la sección `idg` y mostrarlos en la interfaz de forma que el usuario pueda acceder a la información.

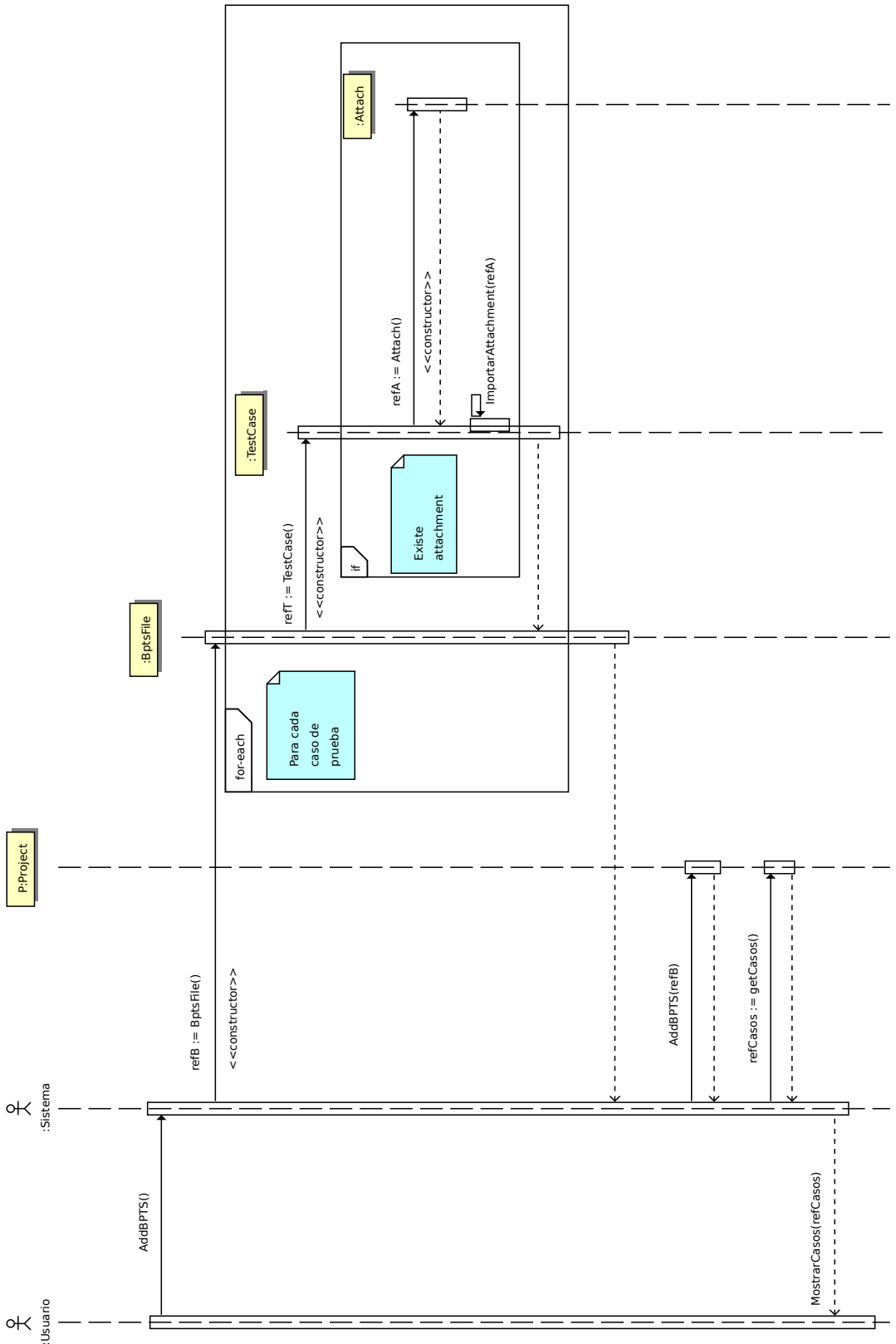


Figura 4.10: Diagrama de secuencia del sistema para *Añadir fichero de caso de prueba* del caso de uso *Gestionar Casos de Prueba*

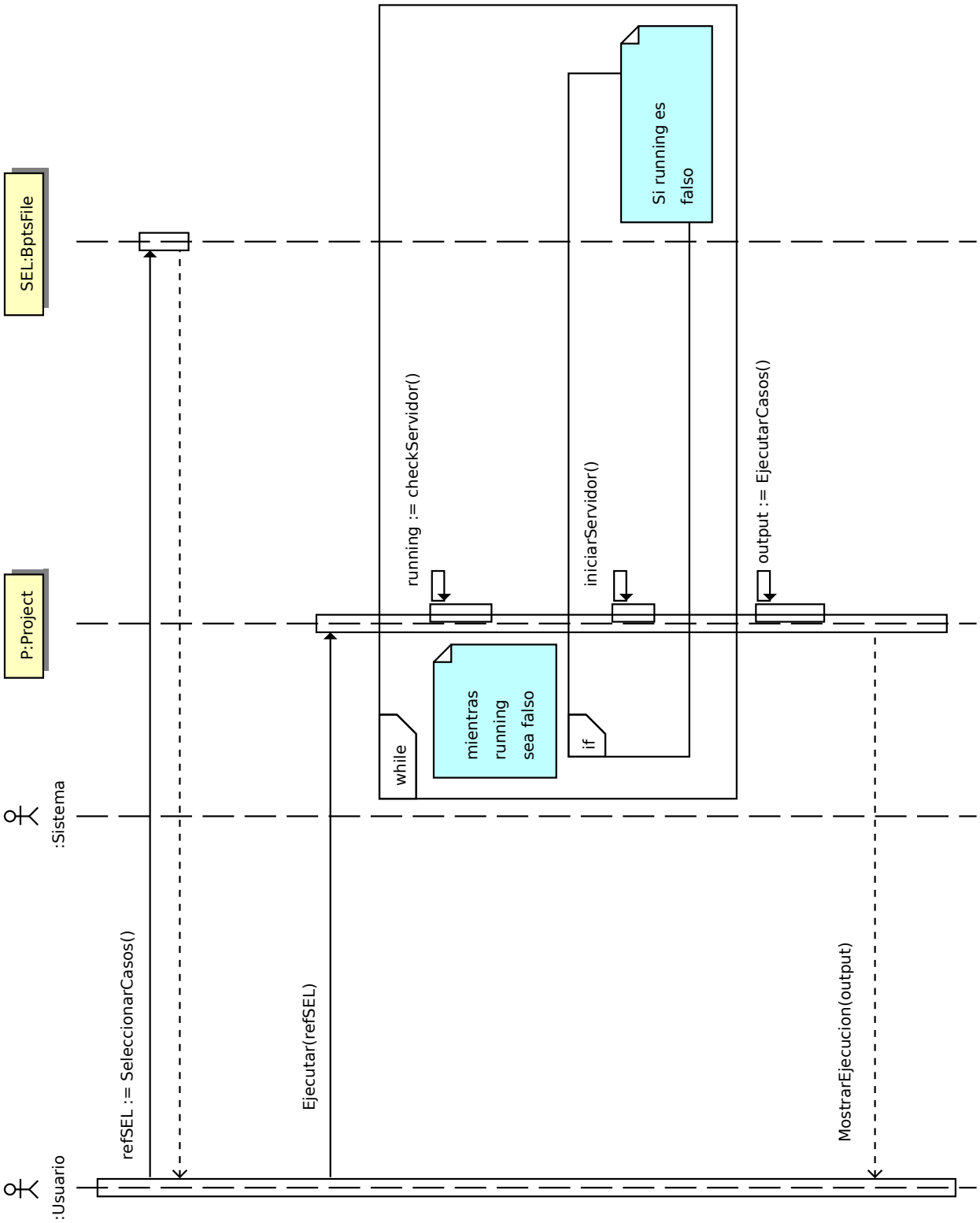


Figura 4.11: Diagrama de secuencia del sistema para el caso de uso *Ejecutar casos de Prueba*

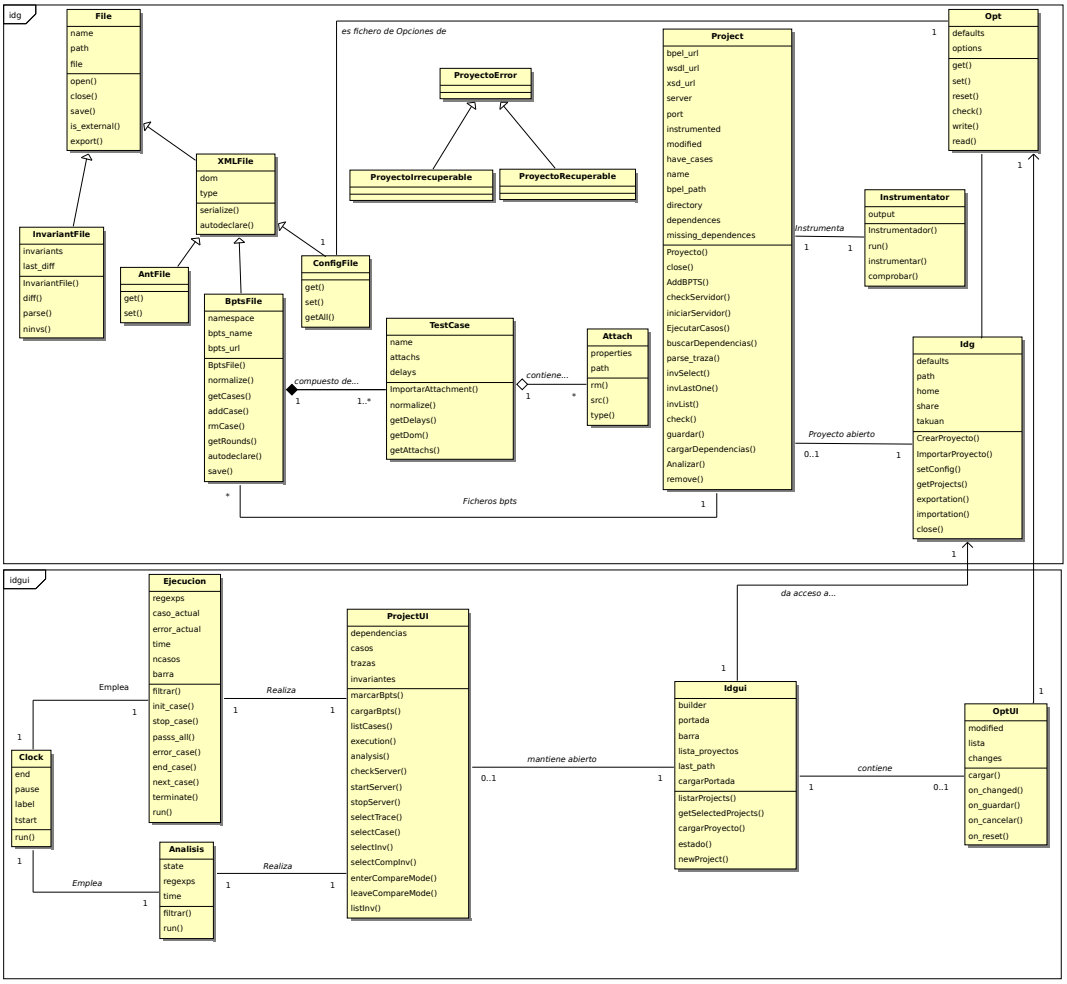


Figura 4.12: Diagrama de datos de diseño completo. En el pueden observarse las dos capas de lógica e interfaz distribuidas en dos paquetes diferentes, `idg` e `idgui`. Para mayor detalle puede resultar conveniente consultar las ampliaciones de ambas secciones que pueden verse en los subdiagramas disponibles en la figura 4.13 de la página 69 para la sección `idg` y la figura 4.14 situada en la página 70 para el paquete `idgui`

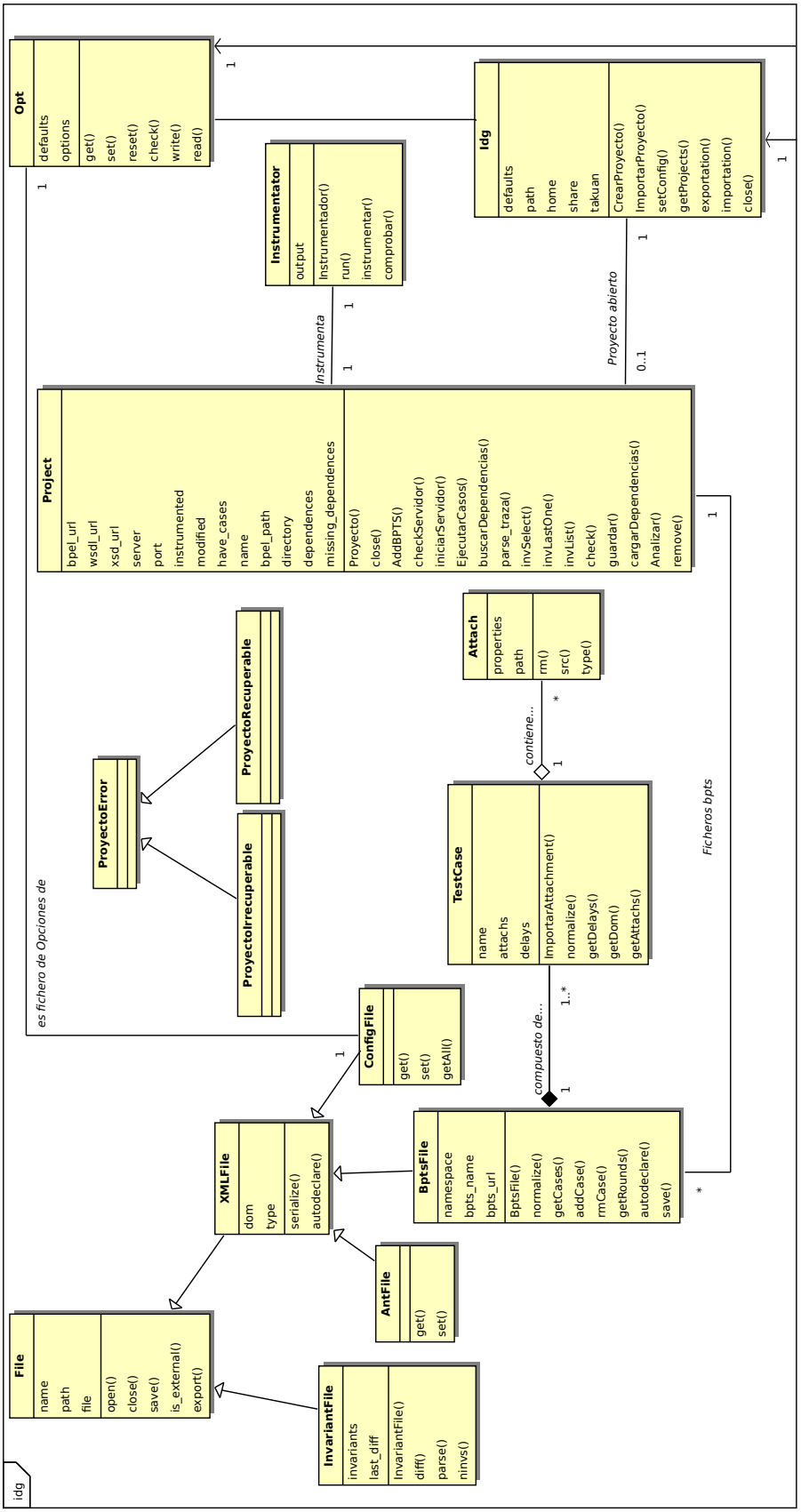


Figura 4.13: Ampliación del diagrama de datos de diseño que puede verse en la figura 4.12(página 68). Amplía el diagrama del paquete `idg` para mayor detalle

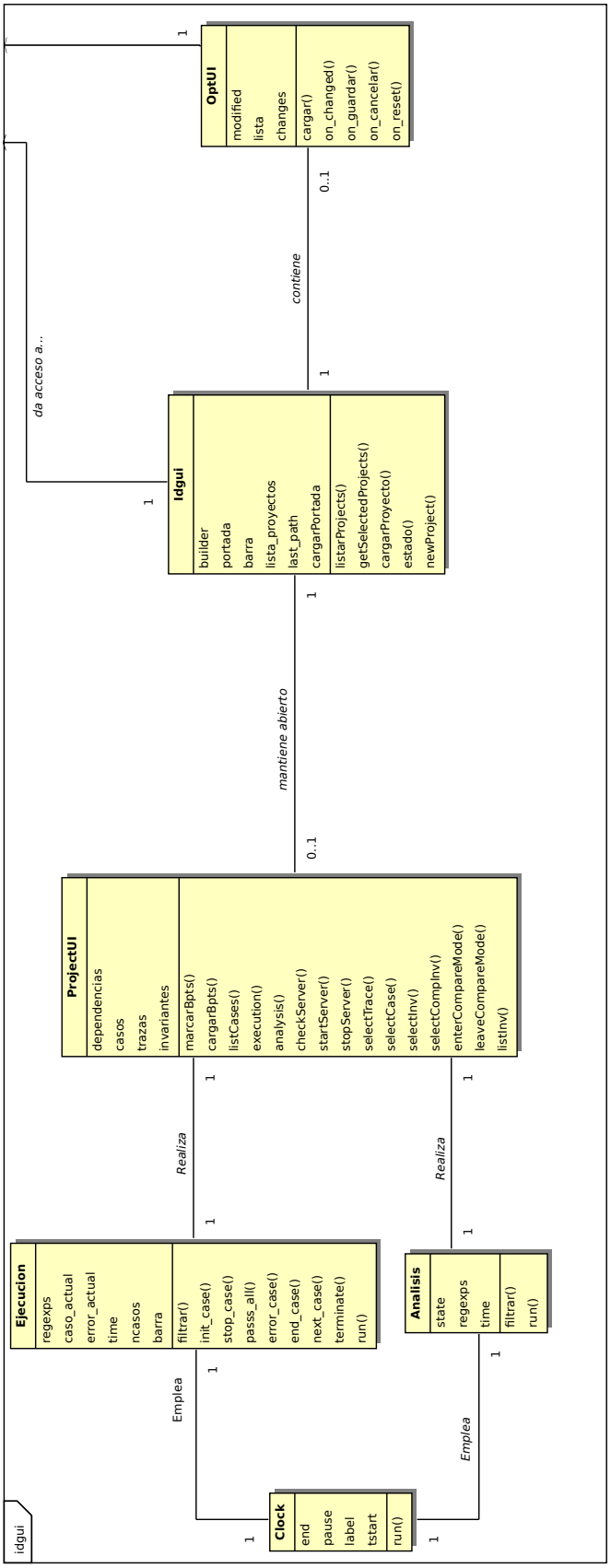


Figura 4.14: Ampliación del diagrama de datos de diseño que puede verse en la figura 4.12 (página 68). Amplía el diagrama del paquete `idgui` para mayor detalle

Las clases `Clock`, `Ejecucion`, `Analisis` y `Instrumentador` son clases trabajadoras, esto es threads o hilos, que pueden ser puestos a funcionar de forma concurrente con el resto de la aplicación y que o bien mantienen la información de su trabajo de forma interna pero consultable por la interfaz, o se comunican directamente con la clase `Proyecto` o `ProyectoUI` a fin de mantener actualizada la interfaz gráfica el progreso de su trabajo.

La clase `ProyectoUI` mantiene la responsabilidad de mostrar la información relativa a la clase de `idg` `Proyecto`. Esto incluye la muestra de todos los ficheros asociados (de invariantes, de casos de prueba unitarios *BPELUnit*, dependencias de los ficheros *WS-BPEL 2.0*, registros de ejecución producidos por el servidor *ActiveBPEL...*), así como manejar las órdenes del usuario con respecto al inicio de las distintas operaciones disponibles en la clase `Proyecto`. El manejo y la gestión gráfica de estas operaciones se encuentran delegadas en las clases trabajadoras `Ejecucion` y `Analisis`, a las cuales la clase `ProyectoUI` controla y presta un marco y un contexto gráfico.

La clase `Clock` es un simple cronómetro o *timer* que puede ser iniciado o pausado y que es capaz de mantener cuenta del tiempo transcurrido desde que se inicia hasta que se consulta, en cualquier momento. Adicionalmente, puede generar texto en un formato adecuado fácilmente entendible y actualizar un componente cualquiera de la interfaz con el de forma periódica.

Las clases `Ejecucion` y `Analisis` mantienen la responsabilidad de mostrar el progreso de las tareas del mismo nombre realizadas por la clase `idg` `Proyecto`. Esto incluye el parseo de la salida de los comandos de *Takuan* invocados y suministrados por `Proyecto` y de

La clase `OptUI` provee a la clase `Opt` de relación con el usuario. En ella se muestra un editor de configuración gráfico que lista las opciones disponibles en la clase `Opt`, redireccionando todas sus operaciones a la capa gráfica, de forma que delega cada operación a realizar sobre su contrapartida lógica del paquete `idg`. Estas acciones incluyen la modificación de opciones y el reseteo a los valores por defecto de todas estas.

Prácticamente cada clase lógica `idg` mantiene su equivalente en la interfaz gráfica `idgui` de forma que la responsabilidad de mostrar al usuario los resultados y el progreso de cada proceso queda así evidenciado de manera sencilla y ordenada.

CAPÍTULO 5

Implementación

5.1. Implementación

En esta sección se tratará de exponer los detalles internos de implementación de *IdiginBPEL*. En ella se mostrarán las estructuras, algoritmos y nuevas herramientas creadas durante el desarrollo de la aplicación.

- **Estructura y configuración**: Distribución del sistema y descripción de la configuración de la aplicación.
- **Proceso de *IdiginBPEL***: Descripción del proceso que realiza *IdiginBPEL* durante su ejecución y de tareas, operaciones, fases y etapas que atraviesa.
- **Formato de proyecto**: Especificación de formato `.idg` y de la estructura interna de un proyecto de la aplicación.
- **Búsqueda de dependencias**: Descripción de la implementación de la funcionalidad de búsqueda de dependencias en ficheros fuente *WS-BPEL 2.0*.
- **Manejo y ejecución de casos de prueba**: Generalización y gestión de los ficheros de suite de casos de prueba unitarios *BPELUnit* y del manejo de su ejecución en un servidor *ActiveBPEL*.
- **Manejo y análisis *Daikon* de registros de ejecución**: Procesamiento de registros de ejecución tanto por parte de *IdiginBPEL* como de *Takuan* y manejo de la ejecución de la herramienta de análisis estadístico *Daikon* sobre los mismos.

- **Manejo y comparación de invariantes**: Manejo y gestión de los invariantes generados por *Daikon* a partir de su análisis y sobre la implementación de la herramienta gráfica de comparación de invariantes.
- **Internacionalización**: Implementación del soporte de internacionalización para la aplicación.
- **Mejoras a *Takuan***: Pequeñas mejoras y colaboraciones con el desarrollo de la herramienta *Takuan*.

5.1.1. Estructura y configuración de *IdiginBPEL*

Estructura

A la hora de dotar de funcionalidad a la interfaz definida, se decidió emplear un diseño dos capas, esto es una capa de *dominio* y de *presentación* correspondientes a dos paquetes *Python* separados llamados `idg` e `idgui`. El paquete `idg` implementaba la parte lógica de las operaciones a realizar, mientras que `idgui` implementaba la funcionalidad que interaccionaba con la biblioteca *PyGTK* y con el usuario directamente. El paquete y la funcionalidad de `idgui` incluía y usaba la funcionalidad disponible en `idg`, la cual estaba completamente desacoplada de esta última, creando así métodos genéricos que podían ser usados desde cualquier parte del programa.

Esta separación permitía que una parte de la aplicación se ocupase de la interfaz gráfica y el aspecto, y otra de las operaciones lógicas. De esta manera el código se simplificaba de forma significativa, ya que lo único que tenía que realizar la sección de `idgui` era emplear las estructuras de datos y métodos que la sección lógica `idg` proveía, sin tener que tratar directamente con los detalles de implementación de *Takuan*, los cuales, unidos a las particularidades del código que manejaba la biblioteca *PyGTK*y los ficheros *glade* hubiesen sido sencillamente demasiado complejos.

Durante el desarrollo de la aplicación, durante la tercera iteración dentro del ciclo iterativo de desarrollo de la misma, se realizó una refactorización general del paquete. La estructura mantenida hasta entonces empleaba dos clases `Proyecto` y `ProyectoUI`. Una de ellas contenía todas las operaciones lógicas que un proyecto realizaba y mantenía disponibles datos aprovechables en operaciones externas que ofrecía públicamente en diversas estructuras; esta era la clase perteneciente a la sección lógica `idg`. La sección del programa responsabilizada de la capa gráfica, `idgui`, empleaba también otra clase `Proyecto`, cuya labor consistía en realizar las operaciones del proyecto empleando como base la funcionalidad lógica provista por `idg`. El principal problema de esta disposición era la acumulación de funcionalidad en las clases `Proyecto` las cuales crecían continuamente, haciéndolas difíciles de manejar y ampliar. El añadido de funcionalidades adicionales requería tener en cuenta una larga serie de recursos y convenciones que se mantenían dentro de la clase convirtiendo esta tarea en cada vez más trabajosa. Además, ciertos problemas ya resueltos anteriormente, sobre todo los relacionados

con ficheros fuente *xml*, se repetían de continuo, incluyendo la escritura de código muy prolijo, convirtiendo en tediosa y propensa a errores la tarea de realizar operaciones con ellos y mantener una gestión de excepciones uniforme.

La reestructuración se realizó reubicando toda la funcionalidad común que se encontraba en la clase `Proyecto`, a veces repetida en varias ocasiones, dentro de un nuevo sistema de clases. Resultó muy interesante el estudio del código fuente de la herramienta *BPELUnit*, la cual se encuentra escrita en *Java*. Tomando como base la misma idea de modelo que sigue el código de *BPELUnit*, en el cual cada archivo de código fuente a manejar por el sistema tiene su propia clase representativa y métodos concretos que lo manejan, en lugar de ser abierto, leído y manejado como un fichero *xml* cualquiera, exponiendo así las particularidades del manejo de cada uno al usuario cada vez que lo emplea.

Configuración

La configuración interna de *IdiginBPEL* incluye un fichero de configuración en el cual se encuentran rutas a componentes importantes, direcciones urls de conexión y otras propiedades. Puede verse una muestra de este fichero en el listado 5.1. El editor carga la configuración del proyecto abierto, acompañada por un texto de ayuda si existe, y permite editar cada valor. En el caso de editar valores incorrectos, siempre es posible volver todos ellos a su valor inicial por defecto, el cual se encuentra almacenado dentro de la aplicación. Puede verse la interfaz gráfica del editor de configuración en la imagen de la figura 5.1.

Listado 5.1: Fichero *xml* de configuración de *IdiginBPEL*.

```
<!-- Default configuration file

     The application looks for this file in the following order:
     1. ~/.idiginbpel/config.xml
     2. ./home/config.xml
     3. /usr/share/idiginbpel/config.xml -->
<config>
    <!-- User directory -->
    <home src="~/.idiginbpel">"idg.help.home"</home>
    <!-- Runtime directory -->
    <share src="~/IdiginBPEL/share">"idg.help.share"</share>
    <!-- takuan instalation -->
    <takuan src="~/takuan">"idg.help.takuan"</takuan>
    <!-- ActiveBpel instalation -->
    <activebpeldir src="~/AeBpelEngine">"idg.help.bpelunit"</bpelunit>
    <!-- ActiveBpel Daemon -->
    <activebpel src="~/bin/ActiveBPEL.sh">"idg.help.activebpeldaemon"</
       bpelunit>
    <!-- Server url -->
    <svr value="localhost">"idg.help.svr"</svr>
    <!-- Server port -->
    <port value="7777">"idg.help.port"</port>
</config>
```

Los valores del fichero de configuración quedan explicados en la siguiente tabla 5.1.

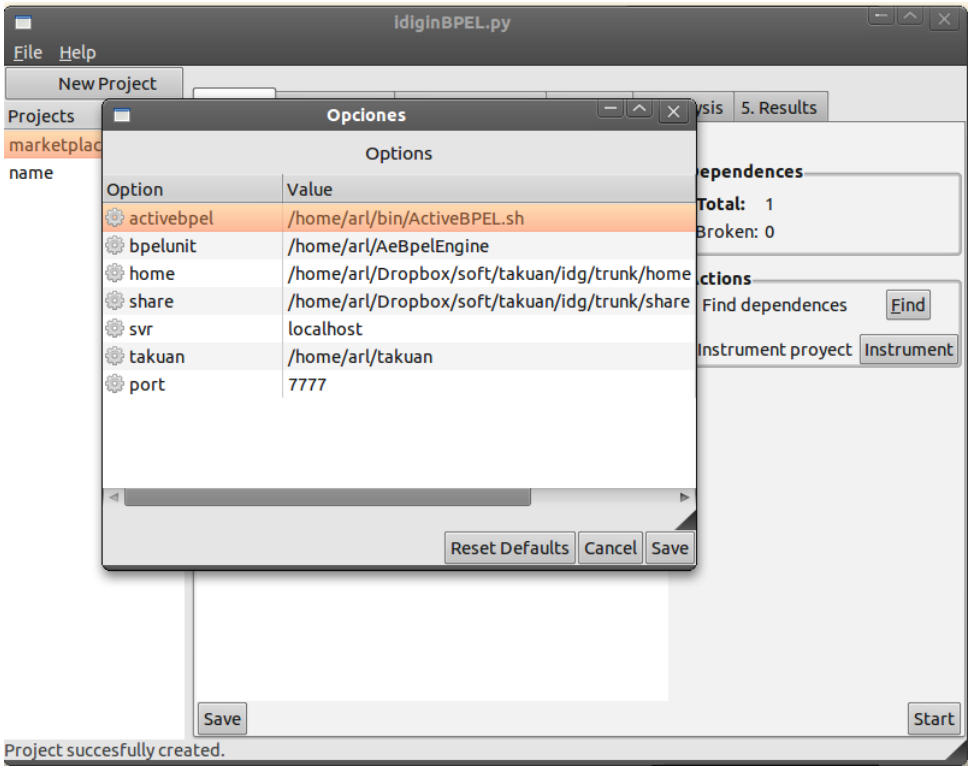


Figura 5.1: Editor gráfico de configuración incluido en *IdiginBPEL*

home	Directorio ”casa” de la aplicación. Allí se encuentra la configuración de usuario y los proyectos del mismo.
share	Directorio de datos de solo lectura compartidos por todas las instancias de la aplicación en el sistema. Usualmente se dispondrá en `/usr/share`.
takuan	Directorio de instalación de *Takuan*.
activebpeldir	Directorio de instalación de *ActiveBPEL*. Necesario para la recolección de registros de ejecución producidos por este.
activebpel	Ruta al guión de manejo de *ActiveBPEL* incluido en la instalación de *Takuan*. Necesario para el control del demonio del servidor.
svr	Dirección url del servidor donde se ejecuta el servidor *ActiveBPEL*.
port	Puerto donde escucha el servidor *ActiveBPEL*.

Tabla 5.1: Información contenida en el fichero de configuración de *IdiginBPEL*.

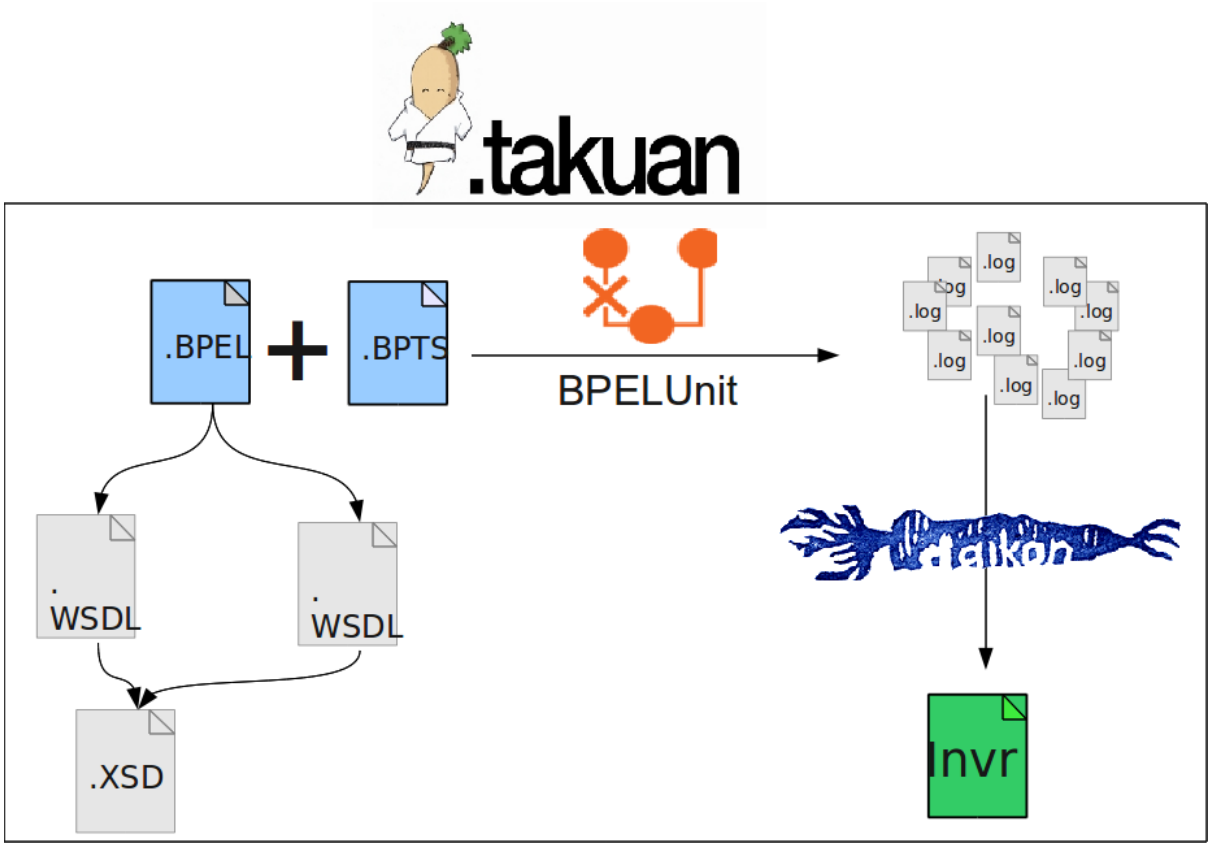


Figura 5.2: Proceso de *IdiginBPEL* y *Takuan* para la generación de invariantes potenciales

5.1.2. Proceso de *IdiginBPEL*

Como capa gráfica para *Takuan*, *IdiginBPEL* debe emplear la mayor funcionalidad posible del sistema ya en marcha, probado y depurado, reimplementando lo mínimo imprescindible. De esta forma el proceso de *IdiginBPEL* es muy similar en términos generales al proceso que realiza *Takuan*, si bien con salida gráfica agradable al usuario y múltiples opciones para dotar de mayor control sobre el proceso así como favorecer una mayor flexibilidad a la hora de hacer pruebas y compartirlas.

El proceso seguía estando dividido, al igual que en *Takuan*, en *Instrumentación*, *Ejecución* y *Análisis*, como puede verse en el esquema situado en la figura 5.2 en la página 77. En el mencionado esquema se muestra el proceso completo sobre un fichero *WS-BPEL 2.0* en *IdiginBPEL*. En el diagrama puede apreciarse como a partir de un fichero fuente y una serie de casos de prueba, pueden generarse trazas mediante la ejecución de los casos en un servidor *ActiveBPEL* para después tratar esas trazas y enviarlas a *Daikon* para su análisis final cuyo resultado es generación de los invariantes, que son el objetivo del proceso.

La ejecución y el análisis podían realizarse tantas veces como hiciesen falta por separado, sin que la ejecución de uno implicase la del otro. Se hizo también posible el añadir múltiples suites de casos de prueba unitarios y más tarde elegir individualmente que casos eran los destinados

a ejecutarse contra el servidor *ActiveBPEL*. Para ello, se serializaba el código de estos casos de forma que podrían tratarse como código xml cualquiera y de manera programática se generaba el código de pruebas unitarias candidato a la ejecución, compuesto con casos de prueba provinientes de diferentes suites, si el usuario lo requería.

El proceso completo puede desglosarse de la siguiente manera:

1. Añadido de un fichero *WS-BPEL 2.0*.
2. Búsqueda e inclusión de sus dependencias.
3. Instrumentación del código fuente.
4. Adición de casos de prueba unitarios *BPELUnit*.
5. Ejecución de la composición contra esos casos en un servidor *ActiveBPEL*.
6. Recolección de trazas desde el servidor *ActiveBPEL*.
7. Procesado de trazas al formato específico de *Daikon*.
8. Análisis en *Daikon* de las trazas transformadas.
9. Recolección de invariantes generados.
10. Visualización y/o comparación de invariantes generados.

Los primeros tres puntos pertenecen al caso de uso *Creación de proyecto*, donde se obtiene el código fuente *WS-BPEL 2.0* listo para su ejecución. Esto trae consigo el procesado genérico de ficheros fuente de tipo *WS-BPEL 2.0* y el control del proceso de instrumentación de *Takuan*. Todo esto se explica en los puntos 5.1.3 (página 80) y 5.1.4 (página 85). La adición de casos de prueba *BPELUnit* incluye también el manejo genérico de ficheros de suite de casos de prueba unitarios *BPELUnit* y el control de las tareas de *Takuan* durante la ejecución de los mismos en el servidor *ActiveBPEL*, así como cierto control e interactuación con dicho servidor; esto se explica en la sección 5.1.5 (página 89). El procesado de registros de ejecución, conversión a trazas *Daikon* y su análisis requiere del manejo de ficheros externos al proyecto y de control sobre las tareas de *Takuan* en su comunicación con *Daikon*, lo cual puede verse explicado en la sección 5.1.6 (página 92). Finalmente, el manejo de invariantes consiste en el parseo e interpretación de los mismos por parte de la aplicación y de la visualización gráfica de ellos, sección 5.1.7 (página 92).

La ejecución de *Takuan* se encontraba convenientemente controlada mediante guiones *ANT* que realizaban todas las operaciones correspondientes. En el sistema nuevo se propuso mantener una versión modificada de estos guiones, que sería modificada programáticamente para ser adaptada a cada proyecto. De esta manera, cada proyecto tendría un fichero `build.xml` que serviría para ejecutarlo.

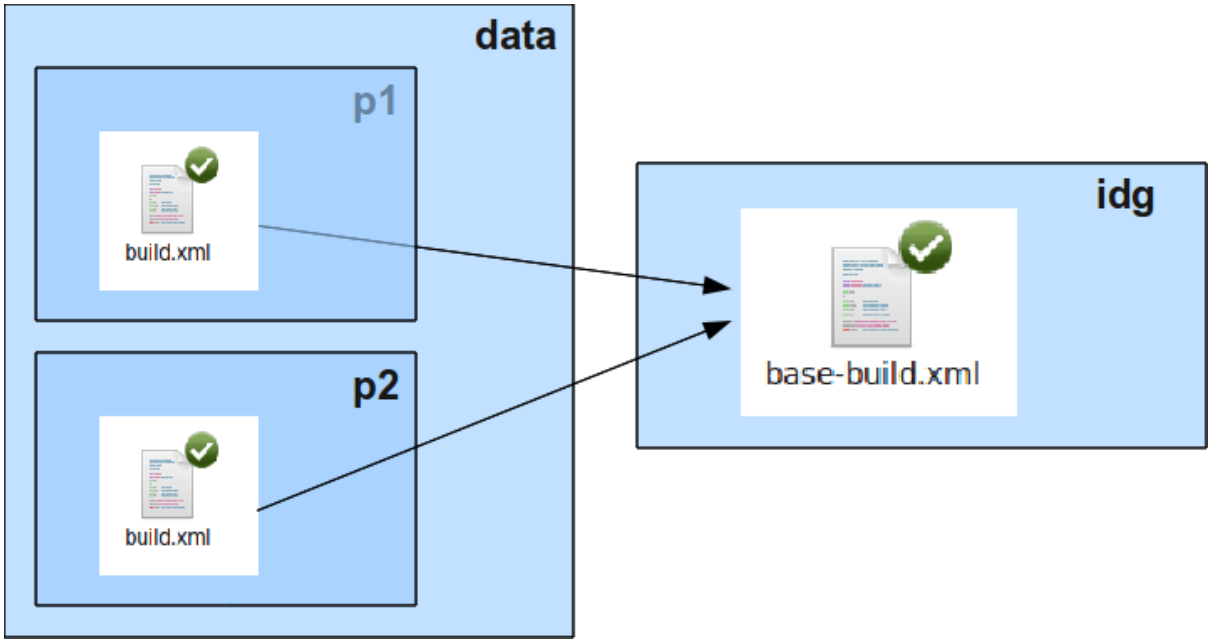


Figura 5.3: Relación de inclusión entre los ficheros *ANT* dentro de los proyectos

Listado 5.2: Guión *ANT* definiendo variables de proyecto

```
1  <?xml version="1.0" ?>
2  <project name="ServicioPrestamo" default="test-and-analyze" basedir=".">
3
4      <!-- Este fichero se encuentra en el directorio del proyecto -->
5      <dirname property="proy" file="${ant.file.BaseBPELBuildFile}"/>
6
7      <!-- OPCIONES -->
8
9      <!-- Nombre del fichero BPR a generar -->
10     <property name="bprfile"          value="${proy}/bpr_file.bpr"/>
11
12     <!-- Nombre del fichero BPTS a emplear para dirigir el proceso de
           prueba -->
13     <property name="bptsfile"         value="${proy}/test.bpts"/>
14
15     <!-- Nombre del fichero BPEL original -->
16     <property name="main.bpel"        value="${proy}/bpel_original.bpel"/>
17
18     <!-- IMPORTACION -->
19
20     <!-- Importamos el fichero Ant base -->
21     <import file="${proy}/base-build.xml"/>
22
23     <!-- Indicamos que hemos importado -->
24     <target name="abstract"/>
25
26 </project>
```

Este fichero `build.xml`, que puede verse en el listado 5.2. El fichero importa otro mayor llamado `base-build.xml` que contiene la parte común a todas las ejecuciones. El esquema de inclusión puede verse en la figura 5.3. Este último fichero acepta diversos argumentos como:

- **build-bpr**: Instrumentación del fichero *BPEL*. Crea un fichero *.bpel* nuevo ”anotado” que incluye instrucciones para mostrar valores de variables y comienzos y finales de secuencia.
- **test**: Ejecución en el servidor *ActiveBPEL*. Toma el el *BPEL* instrumentado, lo envía al servidor *ActiveBPEL*y ejecuta una suite de casos unitarios de prueba contra el servidor, produciendo un registro de ejecución por cada caso ejecutado con éxito.
- **analyze**: Análisis de las trazas generadas por la ejecución a partir del analizador estadístico *Daikon*. Este último recibe los registros de ejecución previamente transformados a su formato, y genera un fichero final con invariantes.

Esta reutilización de guiones *ANT* mejoraba la interacción con el sistema subyacente, *Takuan*, al ser más sencillo adaptar las (probables) modificaciones a su sistema, eliminando así posibles problemas de compatibilidad, debido al uso del mismo esquema para invocar cada orden y estructurar el flujo.

5.1.3. Formato de proyecto

El proceso de *Takuan* toma una serie de ficheros dispuestos según cierta convención y genera un directorio nuevo con todos los resultados y subproductos del proceso en cada ejecución. Esto hace tediosa y dificulta la comparación entre diversas ejecuciones y desanima a la experimentación y a los cambios, al verse en la necesidad de modificar ficheros, cambiar entre directorios y manejar archivos sin saber de forma cómoda a que ejecución pertenecen. Con el fin de poder mantener un historial de ejecuciones, poder realizar diversas pruebas comprobando su influencia sobre los invariantes resultantes y facilitar todo el proceso, se decidió hacer girar todo el formato subyacente del sistema sobre la unidad de código *WS-BPEL 2.0*. Esto es, se decidió seguir una estructura de proyectos, donde la unidad principal de estudio fuese una composición, la cual se procesaría, añadiéndose casos de prueba y otros complementos variables. De esta manera, la única pieza imprescindible para la creación de un proyecto y el diferenciador entre los mismos será el fichero *WS-BPEL 2.0*.

El diseño de un proyecto *WS-BPEL 2.0*, debía pues almacenar el código fuente a analizar, sus dependencias, los casos de prueba unitarios *BPELUnit* añadidos, los registros de ejecución o trazas generadas por la ejecución y los invariantes resultantes del análisis. Se decidió estructurar el proyecto como puede verse en listado 5.3.

Podemos ver la estructura interna de un proyecto *IdiginBPEL* así como los datos contenidos en el en el listado 5.3 y la tabla 5.2. Pasaremos a describir cada los pormenores de cada apartado.

Configuración	Fichero con la configuración del proyecto.
Fichero *bpel* original y dependencias	El fichero de código *WS-BPEL 2.0* originalmente empleado para crear el proyecto así como todas sus dependencias.
Casos de prueba	Los ficheros de casos de prueba *BPELUnit* añadidos al proyecto.
Trazas de ejecución	Los ficheros de trazas resultantes de la ejecución de los casos de prueba *BPELUnit* en el servidor *ActiveBPEL*.
Invariantes resultantes	Los invariantes resultantes de los distintos análisis de trazas realizados.

Tabla 5.2: Información contenida en un proyecto de *IdiginBPEL*.

Listado 5.3: Directorios y ficheros que conforman el "esqueleto" de un proyecto *IdiginBPEL*. (Directorios marcados con *d*)

```
proyecto
    dependencias  [d]
    trazas        [d]
    anl_trazas    [d]
    casos         [d]
    invariantes   [d]

    proyecto.xml
    base-build.xml
    test.bpts
    bpel_original.bpel
```

Un proyecto *IdiginBPEL* consiste en un directorio cuyo nombre será el mismo que el del proyecto de esta decisión vienen las restricciones a los posibles identificadores de proyecto, este debe ser un nombre válido y no conflictivo con la «shell». Dentro de este directorio existen una serie de ficheros así como de otros subdirectorios. Los ficheros en la raiz son ficheros importantes que son únicos, bien código *WS-BPEL 2.0* o configuraciones; en cada subdirectorio se almacenará una clase concreta de ficheros cuyo número ignoraremos a priori. Estos directorios se crean completamente vacíos.

Código original y sus dependencias

Como hemos dicho, el centro de un proyecto es el fichero *WS-BPEL 2.0* que almacena y sus dependencias, estas son importadas durante la creación del proyecto y será tratado con mayor profundidad en la siguiente sección 5.1.4. Dentro del proyecto, el fichero con el código original, y posteriormente su versión procesada e instrumentada, será guardada en la raíz del directorio. Las dependencias importadas serán almacenadas en el directorio `dependencias`.

Guión *ANT* `base-build.xml`

Este fichero contiene las ordenes de ejecución basadas en el fichero del mismo nombre incluido en la instalación de *Takuan*. Es empleado de forma común por todos los proyectos. Puede verse un ejemplo de guión *ANT* (no este) en esta misma sección en el listado 5.2 (página 79).

Directorio de invariantes

Los invariantes generados por cada análisis de *Daikon* son incluidos en este directorio. Cada fichero de invariantes se renombra con una nomenclatura que indica su «timestamp» o marca de tiempo de creación.

Casos de prueba unitarios de *BPELUnit*

Los casos de prueba de *BPELUnit* vienen en ficheros *.bpts* que incluyen una suite completa de casos, como puede verse en el listado 5.4. En el ejemplo, que decide si conceder o no un préstamo a partir de su importe, puede verse la especificación de un cliente (`<tes:clientTrack>`) y de un servicio externo («partner»), el cual intervendrá en la composición (`<tes:partnerTrack>`). El centro del caso se basa en la realización del cliente de una operación *send-receive* para garantizar el préstamo, enviando el importe y recibiendo la respuesta.

Estos ficheros de suite de casos de prueba *BPELUnit* (`.bpts`) son incluidos en el directorio `casos`, para su procesamiento posterior. Al ser incluidos en el directorio, se busca un nombre único para ellos, conservando en su identificador el nombre original para facilitar la identificación por parte del usuario al ser mostrados estos más tarde en la interfaz gráfica.

El fichero `test.bpts` es el ”fichero de casos de prueba general”. Es decir, es usado para aglutinar todos los casos en una sola suite de forma que puedan ejecutarse todos los casos requeridos al mismo tiempo. Este fichero es reescrito programáticamente cuando se prepara una ejecución de casos de prueba en un servidor *ActiveBPEL* y a él son añadidos los casos de prueba que deben entrar en el proceso.

Listado 5.4: Fichero tipo `.bpts` de suite de casos de prueba unitarios *BPELUnit*.

```
1 <?xml version="1.0" encoding="utf-8"?><tes:testSuite xmlns:ap="http://
    j2ee.netbeans.org/wsdl/ApprovalService" xmlns:as="http://j2ee.
    netbeans.org/wsdl/AssessorService" xmlns:gen="http://xml.netbeans.org
    /schema/Loans" xmlns:sp="http://j2ee.netbeans.org/wsdl/LoanService"
    xmlns:tes="http://www.bpelunit.org/schema/testSuite">
2
3   <tes:name></tes:name>
4   <tes:baseURL>http://localhost:7777/ws</tes:baseURL>
5   <tes:deployment>
6     <tes:put name="loanApprovalProcess" type="activebpel">
7       <tes:wsdl>LoanService.wsdl</tes:wsdl>
8       <tes:property name="BPRFile">LoanApprovalDoc.bpr</tes:property>
```

```
9       </tes:put>
10      <tes:partner name="assessor" wsdl="AssessorService.wsdl"/>
11      <tes:partner name="approver" wsdl="ApprovalService.wsdl"/>
12    </tes:deployment>
13
14    <tes:testCases xmlns:tes="http://www.bpelunit.org/schema/testSuite">
15      <tes:testCase abstract="false" basedOn="" name="
          loanApprovalCasosAntonia8.bpts1313424719.55:SmallAmountLowRisk"
          vary="false" xmlns:tes="http://www.bpelunit.org/schema/testSuite"
          >
16        <tes:clientTrack xmlns:tes="http://www.bpelunit.org/schema/
            testSuite">
17          <tes:sendReceive operation="grantLoan" port="LoanServicePort"
              service="sp:LoanServiceService" xmlns:tes="http://www.
              bpelunit.org/schema/testSuite">
18            <tes:send fault="false" xmlns:tes="http://www.bpelunit.org/
                schema/testSuite">
19              <tes:data xmlns:tes="http://www.bpelunit.org/schema/
                    testSuite">
20                <gen:ApprovalRequest xmlns:gen="http://xml.netbeans.org/
                      schema/Loans">
21                  <gen:amount xmlns:gen="http://xml.netbeans.org/schema/
                        Loans">9999</gen:amount>
22                </gen:ApprovalRequest>
23              </tes:data>
24            </tes:send>
25
26            <tes:receive fault="false" xmlns:tes="http://www.bpelunit.org/
                  schema/testSuite">
27              <tes:condition xmlns:tes="http://www.bpelunit.org/schema/
                    testSuite">
28                <tes:expression xmlns:tes="http://www.bpelunit.org/schema/
                      testSuite">//gen:accept</tes:expression>
29                <tes:value xmlns:tes="http://www.bpelunit.org/schema/
                      testSuite">'true'</tes:value>
30              </tes:condition>
31            </tes:receive>
32          </tes:sendReceive>
33        </tes:clientTrack>
34
35        <tes:partnerTrack name="assessor" xmlns:tes="http://www.bpelunit.
            org/schema/testSuite">
36          <tes:receiveSend operation="assessLoan" port="
                AssessorServicePort" service="as:AssessorServiceService"
                xmlns:tes="http://www.bpelunit.org/schema/testSuite">
37            <tes:receive fault="false" xmlns:tes="http://www.bpelunit.org/
                  schema/testSuite"/>
38            <tes:send fault="false" xmlns:tes="http://www.bpelunit.org/
                  schema/testSuite">
39              <tes:data xmlns:tes="http://www.bpelunit.org/schema/
                    testSuite">
40                <gen:AssessorResponse xmlns:gen="http://xml.netbeans.org/
                      schema/Loans">
41                  <gen:risk xmlns:gen="http://xml.netbeans.org/schema/
                        Loans">low</gen:risk>
```

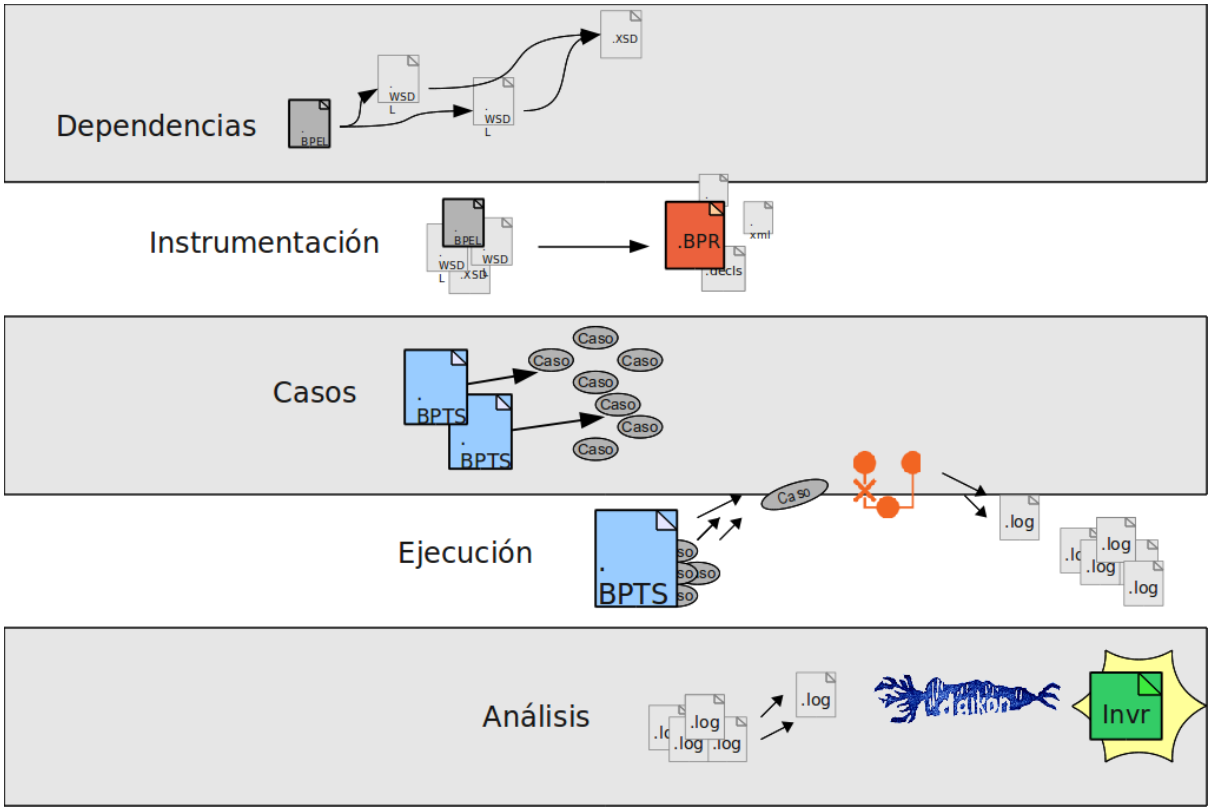


Figura 5.4: Esquema de los productos intermedios generados por *IdiginBPEL*

```
42                </gen:AssessorResponse>
43              </tes:data>
44            </tes:send>
45          </tes:receiveSend>
46        </tes:partnerTrack>
47
48        <tes:partnerTrack name="approver" xmlns:tes="http://www.bpelunit.
              org/schema/testSuite"/>
49      </tes:testCase>
50  </tes:testCases>
```

Registros de ejecución y trazas

El directorio `trazas` es el destino de los registros de ejecución generados por los casos de prueba unitarios de *BPELUnit* en un servidor *ActiveBPEL*. La ejecución de cada caso genera una o más ficheros `.log` de registro de ejecución. Durante la el proceso son importados al mencionado directorio, adquiriendo también un nombre identificativo conformado por `suite-caso-timestamp`. El directorio `anl_trazas` contiene la transformación de registros de ejecución en trazas, las cuales se encuentran en un formato compatible con *Daikon* y aptas para el proceso de análisis estadístico que este realiza sobre ellas.

Fichero de configuración `proyecto.xml`

Este fichero describe el estado interno del proyecto y se emplea para poder acceder a los listados de ficheros y la configuración propia sin necesidad de realizar operaciones de listado de ficheros del sistema operativo. Un ejemplo de este fichero puede encontrarse en el listado disponible en la figura 5.5.

Listado 5.5: Ejemplo de fichero de configuración de proyecto.

```
<proyecto nombre="" creado="" guardado="">
    <!-- bpel -->
    <bpel_o src="" timestamp=""/>
    <bpel src="bpel_original.bpel" timestamp=""/>
    <bpr  src="bpr_file.bpr" timestamp=""/>

    <svr url="localhost" port="7777"/>

    <!-- dependencias -->
    <dependencias>
        <!-- <dependencia src="" tipo="" rota=""/> -->
    </dependencias>

    <invariantes>
        <!-- <invariante src="" timestamp="">
            <traza nombre="" src="" timestamp="" actual=""/>
        </invariante> -->
    </invariantes>
</proyecto>
```

Los datos incluidos hacen posible seguir rastro a todos los ficheros internos que componen un proyecto de forma efectiva. La aplicación guarda este fichero y lo sincroniza mediante las operaciones `guardar` de la clase `Proyecto`.

5.1.4. Dependencias

Un fichero *WS-BPEL 2.0* normalmente viene acompañado de sus declaraciones *WSDL* y estas, de declaraciones de tipo *XML Schema*. Por tanto, a la hora de crear un proyecto, no solo es necesario copiar el fichero *WS-BPEL 2.0* e introducirlo en el directorio de proyecto, sino que también es necesario incluir sus dependencias. Esto puede hacerse de forma automática analizando el código fuente y siguiendo la ruta indicada en las directivas `import`, de los ficheros *bpel* y *WSDL*. Debido a que los ficheros *WSDL* pueden incluir a otros ficheros *WSDL* y ficheros de tipo *XML Schema*, es necesario hacer esta búsqueda de forma recursiva.

Las rutas coleccionadas a partir de las directivas **import** de los ficheros fuente, suelen ser rutas relativas que emplean como base la propia localización del fichero, de forma que cada ruta leída debe ser unida a la localización en disco del fichero fuente original. No basta con coleccionar las rutas y copiar más tarde todos esos ficheros dentro del proyecto, debido a que la estructura

de directorios y la situación del fichero de código fuente original *WS-BPEL 2.0* y la localización de sus dependencias puede (suele) ser muy diferente a la interna del proyecto; esto significa que se han de reescribir todos los ficheros de código fuente a introducir en este, reapuntando cada ruta indicada en sus directivas `import` de forma que se adapten a la nueva estructura interna del proyecto[1].

El algoritmo empleado para la extracción de estas dependencias puede verse en el listado incluido en la figura 5.6 (página 86). El listado muestra como, inicialmente a partir del fichero *WS-BPEL 2.0* que es pasado como argumento a la función `buscar_dependencias` (línea 4), se analiza el código en busca de directivas `import` (línea 35) y para cada directiva `<import>` encontrada, se añaden las rutas a nuevas dependencias encontradas (línea 42) para a continuación reescribir la dirección de la ruta al lugar que vaya a ocupar dentro del proyecto (línea 52). Una vez hecho esto se escribe el fichero modificado dentro del proyecto (línea 63). El algoritmo continua iterando recursivamente sobre el conjunto total de dependencias encontradas hasta haberlas procesado todas (línea 73). Si alguna dependencia no pudo leerse o importarse, será guardada como ”rota” (línea 30). En el momento en que recursivamente se vuelva a la llamada raíz, esto es, la llamada que analizaba el código *WS-BPEL 2.0* original, la recursión se rompe y se devuelve el resultado.

Listado 5.6: Algoritmo de búsqueda e importación de dependencias de *IdiginBPEL*. Simplificado por claridad

```
1       def buscar_dependencias(self, bpel):
2           # Buscar las dependencias del bpel recursivamente
3           ## Lista de rutas con las dependencias del bpel
4           self.deps, self.dep_miss = self.__buscar_dependencias([bpel])
5
6       def __buscar_dependencias(self,files,deps=set(),miss=set(),first=
            True):
7           if first:
8               deps = set()
9               miss = set()
10
11          # Caso de parada de la funcion
12          if len(files) == 0 :
13              return []
14
15          local_deps = set()
16
17          # Buscamos en todas las rutas de ficheros que recibamos
18          for f in files:
19              if path.exists(proy):   # Si ya existe en el proyecto
20                  miss.discard(dir)  # Quitamos de las dependencias rotas
21
22              # Abrimos el fichero, obtenemos los imports, modificamos las
                    rutas
23              # y lo serializamos de nuevo pero dentro del proyecto.
24
25              # Cargar fichero en memoria
26              try:
```

[1]La estructura interna del proyecto para el almacenamiento de dependencias, y del fichero fuente *WS-BPEL 2.0* original, puede verse con más detalle en la sección anterior 5.1.3

```
27                  xml = md.parse(f)
28              except:
29                  # a las dependencias rotas
30                  miss.add(f)
31                  continue
32
33              # Buscar los imports en el fichero
34              # empleando los distintos namespaces posibles
35              imps_l = xml.getElementsByTagName('import')
36
37              # Obtener las rutas absolutas de los import
38              # Modificar las rutas de los import
39              for i in imps:
40                  attr = 'location' if i.hasAttribute('location') else '
                        schemaLocation'
41                  ruta = i.getAttribute(attr)
42                  deps.add(ruta)
43
44                  # Los bpel (first) apuntan a dentro del directorio
                        dependencias
45                  # El resto de dependencias, al mismo directorio en el
                        que estan.
46                  if first :
47                      ruta = path.join(self.dep_nom, path.basename(ruta))
48                  else:
49                      ruta = path.basename(ruta)
50
51                  # Reescribir el codigo con la ruta correcta
52                  i.setAttribute(attr, ruta)
53
54              # Copiar el fichero en el proyecto
55              try:
56                  # Serializar el xml a un fichero en el directorio self.
                        dep_dir
57                  # Con la ruta adecuada si es el bpel original.
58                  if first :
59                      file = open(self.bpel,'w')
60                  else:
61                      file = open(path.join(self.dep_dir, nom), 'w')
62
63                      file.write(xml.toxml('utf-8'))
64              except:
65                  miss.add(ruta)
66              finally:
67                  file.close()
68
69                  # fin del for
70          # fin del for
71
72          # Llamada recursiva, false porque ya no es la primera llamada.
73          self.__buscar_dependencias(list(local_deps), deps, miss, False)
74
75          # Si es la llamada del fichero bpel, es la llamada final,
76          # devolvemos todas las dependencias.
77          if first:
```

```
78                 return list(deps), list(miss)
79             # Si es una llamada normal, devolvemos los ficheros a mirar en
80             # la siguiente iteracion.
81             else:
82                 return files
```

Una posible solución que se valoró de forma que no implicase la reescritura del código de forma automática, hubiese pasado por realizar presunciones sobre la localización de estas dependencias y mantener una convención[2] que el usuario debería cumplir si deseaba importar el fichero fuente de forma exitosa. Esto se juzgó podía ser contraproducente, limitando la flexibilidad de la herramienta y eliminando gran parte de la ventaja obtenida de la extracción automática de dependencias.

El hecho de que se haya valorado la posibilidad de no reescribir el código fuente de los ficheros a importar deriva de la problemática creada por las deficientes características en la serialización de las bibliotecas de manejo *xml* de las que se disponía. En ellas, un fichero *xml* cargado en memoria, **no** modificado y vuelto a serializar, **mantenían diferencias importantes**, en algunos casos, tan grandes que el código del lenguaje basado en *xml* en cuestión se volvía en imposible de ejecutar, quedando así, roto.

La reescritura de código basado en *xml* no resultó tan sencilla como se prometía debida a la adaptación particular de cada estándar basado en *xml* del propio estándar *xml*. Este estándar *xml* dicta que los espacios de nombres para los atributos de los elementos *xml*, no tienen significado alguno, y pueden ser cualquier cadena alfanumérica aleatoria, mientras el espacio de nombre se encuentre adecuadamente declarado al inicio del documento. Este comportamiento no es respetado por el estándar *XPath*, que si que incluye carga semántica en los espacios de nombre, no siendo posible variar la cadena identificativa, a costa de romper el código. Algunas bibliotecas de serialización *xml*se apegan demasiado estrictamente al estándar *xml*, reinventando los espacios de nombre a placer en cada serialización. Podemos ver un ejemplo de *xml*original y serializado respectivamente en los listados 5.7 y 5.8.

Listado 5.7: Fichero *WS-BPEL 2.0* original sin haber sido serializado

```
1 <process xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable
    "
2 xmlns:ns0="http://xml.netbeans.org/schema/Loans" (..) >
3 <if name="If1">
4 <condition> ( number(string($processInput.input/ns0:amount)) &lt;= 10000
    ) </condition>
5 <sequence name="SmallAmount">
6 <assign name="copyLoanInfoToAssessorInput">
7 <copy>
8 <from>$processInput.input/ns0:amount</from>
9 <to>$assessorInput.input/ns0:amount</to>
10 </copy>
11 (..)
```

[2]Mantener una convención sobre la localización de las dependencias es lo que hace *Takuan*, al exigir que se encuentren todas ellas en el mismo directorio que el fichero *.bpel*

Listado 5.8: Fichero *WS-BPEL 2.0* original tras haber sido serializado.

```
<ns0:process xmlns:ns0="http://docs.oasis-open.org/wsbpel/2.0/process/
    executable"
xmlns:ns1="http://xml.netbeans.org/schema/Loans" (..)>
<ns0:if name="If1">
<ns0:condition> ( number(string($processInput.input/ns0:amount)) &lt;=
    10000 ) </condition>
<ns0:sequence name="SmallAmount">
<ns0:assign name="copyLoanInfoToAssessorInput">
<ns0:copy>
<ns0:from>$processInput.input/ns0:amount</from>
<ns0:to>$assessorInput.input/ns0:amount</to>
</copy>
(..)
```

Esta molesta propiedad de algunas bibliotecas *Python* de manejo *xml* supuso un impedimento y retrasó el desarrollo hasta que se halló la manera de emplear una de las diferentes bibliotecas *xml* disponibles para *Python* de forma que respetase los espacios de nombre originales. Esta biblioteca resultó ser *minidom*, que serializa ignorando los espacios de nombre, pero afortunadamente los mantiene en memoria, de forma que pudo encontrarse un método recursivo, disponible en el listado 5.9 que añadía el prefijo de los espacios de nombre al nombre del elemento original, proveyendo de una serialización que dejaba tal cual los espacios de nombres.

Listado 5.9: Declaración automática de prefijo de namespaces para la biblioteca *Python minidom*

```
def minidom_namespaces(elto):
    """@brief Declara inline namespaces no declarados
       @param elto Elemento padre"""

    # Utilizamos una cola y procesamos los elementos en orden de
       documento
    eltos = [elto]
    while eltos :
        e = eltos.pop(0)
        # Les damos la declaracion del namespace si tienen
        if e.namespaceURI :
            if not e.prefix :
                e.prefix = 'ns0' # Evitar prefijos vacios
            e.setAttribute('xmlns:' + e.prefix, e.namespaceURI)
        # Metemos sus hijos en la cola
        eltos.extend(e.childNodes)

    return elto
```

5.1.5. Casos de prueba

Los los casos en una suite de casos de prueba unitarios *BPELUnit* no se ejecutan de la misma forma en la que el usuario los importa y selecciona. Esto es debido a que los casos de prueba

son considerados de forma individual. Significa esto que para cada ejecución, el usuario puede elegir diferentes casos pertenecientes a diferentes suites; aparte, sólo una suite puede ser incluida en una ejecución, de forma que el procedimiento empleado es mantener los casos en su entorno original (la suite que se importó, esto es, su fichero `.bpts`) y se extraen sus elementos `<testCase>` con los casos de prueba seleccionados por el usuario, justo antes de la misma ejecución, componiendo otra suite general de casos de prueba llamada `test.bpts`[3]. El procedimiento de abrir un fichero `.bpts`, parsearlo y extraer o modificar en el sus casos es largo, tedioso y propenso a errores, los cuales también deben ser manejados, de forma que se extrajo su funcionalidad en la clase `BPTSFile`, los cuales describimos a continuación en la siguiente sección.

Clase `BPTSFile`

Dentro de la jerarquía de clases que representan los ficheros a manejar por *IdiginBPEL*, la clase `BPTSFile` es la más grande y compleja. También aglutina la mayor funcionalidad. Es responsabilidad de la clase el importar los ficheros suite de casos de prueba unitarios *BPELUnit*(ficheros `.bpts`) incluyendo los adjuntos (*attachments*) que estos pudiesen incluir, nombrarlos de forma consistente, listar los casos individuales y extraerlos para reescribir otra suite de forma dinámica justo antes de las ejecuciones. Puede ver un ejemplo de fichero de suite de casos de prueba unitarios *BPELUnit* en el listado 5.4 en la página 82.

Una particularidad de los casos de prueba unitarios de *BPELUnit* son los «rounds» o rondas. Cada caso puede ejecutarse más de una vez, representando un numero de intentos durante su ejecución. Esto provoca más de un fichero de registro de ejecución posteriormente y estos casos deben ser marcados en la interfaz gráfica; Por ello también deben detectarse y ser manejados aparte.

Ejecución de casos de prueba unitarios de *BPELUnit*

El proceso de ejecución de casos se realiza mediante un hilo trabajador, que corre de forma paralela al hilo principal de la interfaz y actualiza la interfaz de forma independiente. Esta labor la realiza la clase del paquete `idgui` llamada `Ejecucion`. La clase obtiene la salida del proceso de ejecución de *BPELUnit* a través de una tubería y la va parseando mediante expresiones regulares, detectando de esta forma cuando empieza un caso, termina, se para, tiene éxito o falla, y los rounds realizados por cada uno.

El hilo actualiza la interfaz de forma concurrente. Para prevenir problemas típicos de concurrencia con la interfaz, se emplea el sistema de cerraduras que incluye la biblioteca *GTK*. Cuando se comienza una sección concurrente, se inicia una cerradura que bloquea el sistema completo de interfaz durante esa fracción de código. Esto evita actualizaciones concurrentes de variables y otros desajustes muy difíciles de depurar. Puede verse un ejemplo de código concurrente de la aplicación en el listado 5.10.

[3]Puede leer más sobre este fichero de pruebas general en la sección 5.1.3 en la página 82

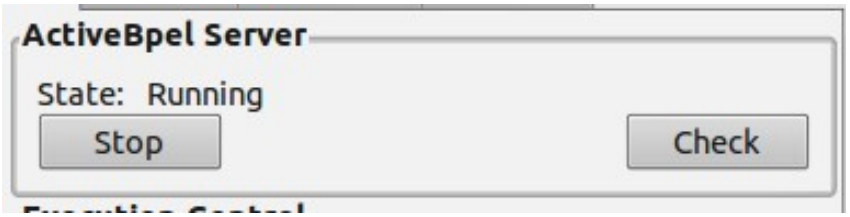


Figura 5.5: Botones de control del servidor *ActiveBPEL* situados en la interfaz gráfica de *IdiginBPEL*

Listado 5.10: Ejemplo de código concurrente en *PyGTK*.

```
            # Actualizar la gui
            gtk.gdk.threads_enter()

            # Si es el primer caso, ponemos 0.06 representando el
            # trabajo realizado por la conexion.
            if self.i_case == 0:
                self.barra.set_fraction(0.06)
            # Aumentar el contador de casos si es un caso normal o el primer
                round
            if not r or round == '1' :
                self.i_case += 1

            # Actualizamos el texto con el contador de casos
            # tenemos en cuenta que los round tienen un texto diferente
            #  Format:
            #  round: this_case / total_cases (this_round_case )
            #  no round: actual_case / total_cases
            text = "%i / %i (%s)" % (self.i_case, self.ncasos, round) if r
               else \
                    "%i / %i" % (self.i_case, self.ncasos)
            self.barra.set_text(text)

            # Activar el estado de que esta ejecutandose (estado 2)
            self.ui.activar_ejec_caso(name, 2)

            gtk.gdk.threads_leave()
```

Durante la etapa de ejecución, el servidor *ActiveBPEL* debe encontrarse activo. Esto puede ser controlado por el usuario a través de la interfaz gráfica. El control del servidor *ActiveBPEL*, se realizó mediante la consulta a la dirección url del servidor empleando las bibliotecas de resolución HTTP de *Python*. En base a la respuesta devuelta por el servidor en la dirección url facilitada, se comprueba si el servidor está activo o no. El servidor *ActiveBPEL* también puede arrancarse a partir de su guión de manejo, incluido en la instalación de *Takuan*. *IdiginBPEL* incluye en su interfaz gráfica, situados en la sección de ejecución dos botones que comprueban y arrancan/paran el servidor, como puede verse en la imagen de la figura 5.5

5.1.6. Registros de ejecución

Los registros de ejecución son generados por el servidor *ActiveBPEL* dotado de las convenientes extensiones de *XPath* desarrolladas como parte del código de *Takuan*. Durante la ejecución de casos de prueba unitarios de *BPELUnit*, el servidor *ActiveBPEL* va generando «logs» o registros de ejecución en su directorio. Estos son importados uno a uno por la clase `Ejecucion`[4] al proyecto tras el término de cada caso. Estos registros de ejecución son renombrados con un identificador único que de paso muestra su origen, el formato es `nombreSuite - nombreCaso - Round - timestamp` y son alojados en el directorio de proyecto `trazas`. Estas trazas almacenadas son dispuestas al usuario de forma que puede elegir cuales de ellas entrarán en el análisis posterior.

Tras haber recolectado al menos un registro de ejecución de caso de prueba unitario de *BPELUnit*, es posible analizarlas mediante *Daikon*. Este, sin embargo, no acepta los registros de ejecución tal y como los genera *ActiveBPEL*, de forma que es preciso transformarlos a un formato compatible[5]. Esta transformación se realiza mediante un componente intermedio de *Takuan* conformado por guiones en lenguaje *Perl*. Estos guiones toman como argumento los guiones a transformar[6] y realizan las operaciones adecuadas de forma en que a su término son generados trazas *Daikon* compatibles.

Estas trazas compatibles generadas, se almacenan siguiendo un formato adecuado en el directorio `anl_trazas`, y son más tarde pasadas como argumento a la ejecución de *Daikon*. El cual, actuando sobre las mismas, genera los correspondientes invariantes. Estas trazas son ficheros temporales que son eliminados tras la ejecución de *Daikon*.

El componente que controla la ejecución de *Daikon* y muestra el progreso del proceso en la interfaz es la clase de interfaz gráfica `idgui` *Analisis*. Esta realiza un proceso de parseo de la salida de *Daikon*, la cual recibe a través de una tubería, de manera similar a la clase *Ejecucion*, la cual se describió en la sección 5.1.5 (página 90). Esta clase parsea la salida de *Daikon* de forma que controla cuantos casos han sido

5.1.7. Invariantes

Los invariantes generados en el análisis estadístico realizado por *Daikon*, son almacenados en el directorio `invariantes`. También siguen un patrón, que muestra el «timestamp» en el cual fueron generados a modo de identificador único. Un fichero de invariantes contiene la salida de *Daikon* con ellos con el formato.

[4] Para más información sobre el proceso de ejecución consultar la sección 5.1.5 disponible en la página 90

[5] Para más información sobre la transformación de registros de ejecución en trazas *Daikon* compatibles, consulte la sección 2.6 (página 11).

[6] Estos guiones necesitaron de modificación para su buen funcionamiento en conjunto con *IdiginBPEL*. Para más información sobre la contribución a *Takuan* modificando estos guiones *Perl*, consulte la sección 5.1.9 (página 98).

```
nombre operador valor
```

Los invariantes son una serie de expresiones matemáticas agrupadas por secuencias, como puede verse en el listado de la figura 5.11. Los invariantes generados por una ejecución no son necesariamente iguales a los de otra, ya sea bien debido al empleo de diferentes casos de prueba o a diferentes resultados producidos por los mismos.

Listado 5.11: Invariantes producidos por *Daikon*.

```
1 LoanApprovalProcess.LargeAmount:::ENTER
2 processInput.input.ApprovalRequest[1].amount[] == [130000.0]
3 processInput.input.Request[0].amount[] == [0]
4 processOutput.output.ApprovalResponse[1].accept[] == [0]
5 processOutput.output.ApprovalResponse[1].accept[] elements == true
```

Es por tanto algo interesante el ver como el conjunto de invariantes producido por *IdiginBPEL* varía. La comparación de invariantes permite seleccionar dos invariantes anteriormente generados por la aplicación dentro de un mismo proyecto, y ver sus diferencias de forma visualmente efectiva. Esta funcionalidad permite ver cómo han cambiado los invariantes de una ejecución a otra, si estas incluyen componentes de tipo aleatorio o cambiante, o ante la presencia de trazas en el análisis realizado pertenecientes a nuevos casos de prueba unitarios. La utilidad principal del análisis de diferencias es ver como hay invariantes que desaparecen, otros que afinan o empeoran su valor, junto a algunos nuevos que antes no eran contemplados. Este proceso permite la depuración de composiciones, detectando invariantes que no deberían darse o con valores extraños y la mejora de casos de prueba, viendo cómo tras mejorar estos, los invariantes mejoran su acierto.

Para poder comparar dos invariantes entre sí, se diseñó una vista que permite cargar dos invariantes a la vez dos ventanas divididas por una división vertical, de forma que se sitúa el primer invariante seleccionado a la izquierda, y el segundo a la derecha. Se desarrolló un algoritmo de búsqueda de diferencias, que puede verse simplificado en el listado 5.12 (página 94), que compara invariante a invariante el invariante situado en la ventana izquierda con el situado en la derecha y que es descrito a continuación.

En este listado 5.12, se construye una estructura (documentada en la línea 8) que contiene unidos los dos conjuntos de invariantes agrupados por secuencias junto a información adicional que indica como reconstruir las dos vistas indicando las diferencias. Esta información se encuentra en unos marcadores adicionales (+, -, c, n) que indican cuando un invariante (o secuencia) es:

- **nuevo**: +, no estaba en el antiguo conjunto de invariantes pero aparece en el nuevo.
- **eliminado**: -, estaba en el antiguo pero no aparece en el nuevo
- **cambiado**: *c*, está en ambos pero no es igual.

- **sin cambios**: *n*, idéntico en ambos conjuntos de invariantes.

La primera operación consiste en encontrar secuencias completamente nuevas o eliminadas, las cuales serán marcadas en verde por completo, evitando así tener que analizar sus invariantes uno a uno (línea 27). Mediante operaciones de conjuntos (`set` en *Python*) tomamos solo las secuencias comunes a ambos ficheros en proceso y procedemos a evaluar sus invariantes (línea 38). Los que no han cambiado y son exactamente iguales en ambos grupos son detectados mediante las mismas operaciones de conjuntos empleadas con las secuencias, siendo estos marcados como no cambiados y eliminados del proceso (línea 47), que prosigue con los que invariantes que son exactamente iguales, los cuales pueden o bien haber cambiado, ser nuevos o haber sido eliminados. Para saberlo, comprobamos los invariantes cambiados del conjunto más antiguo contra los del conjunto más nuevo, y posteriormente realizamos lo contrario (línea 65). Los invariantes se miran todos con todos uno a uno, salvo los ya evaluados, los cuales son marcados para evitar que aparezcan múltiples veces. Una vez determinados los invariantes que han cambiado, y detectados los nuevos y los eliminados, se pasa a la siguiente secuencia hasta terminar el conjunto.

Listado 5.12: Algoritmo de *diff* que construye las diferencias entre dos conjuntos invariantes.

```
1  def diff(self, inv_file):
2          """@brief Computes the differences against another invariant
               file.
3
4              @param inv_file Another invariant file.
5              @returns A dictionary-like structure with instructions to
                   recompose
6              the two views of the diff.
7              Eg:
8                  {
9                    sequence: {
10                       name:'seq',
11                       mode: '+|-|n|c',
12                       list: [(+, inv), (c, inv), ..]
13                     }
14                 }
15                 """
16         diff = {}
17         left_seqs = set(self.invs.keys())
18         right_seqs = set(inv_file.invs.keys())
19
20         new_seqs = right_seqs - left_seqs
21         removed_seqs = left_seqs - right_seqs
22         common_seqs = left_seqs.intersection(right_seqs)
23
24         for seq in left_seqs.union(right_seqs):
25             diff[seq] = {}
26
27         # Sequence in right but not in left. (+, new)
28         for seq in new_seqs:
29             diff[seq]['mode'] = '+'
30             diff[seq]['list'] = [('+', inv) for inv in inv_file.invs[seq
                   ]]
```

```
31
32          # Sequence in left but not in right. (-, removed)
33          for seq in removed_seqs:
34              diff[seq]['mode'] = '-'
35              diff[seq]['list'] = [('-', inv) for inv in self.invs[seq]]
36
37          # Sequence in left and in right. (c, changed)
38          for seq in common_seqs:
39              diff[seq] = {'list': []}
40
41              # Detect unchanged invariants
42              left_invs = set(self.invs[seq])
43              right_invs = set(inv_file.invs[seq])
44              common_invs = left_invs.intersection(right_invs)
45
46              # Add unchanged invariants
47              diff[seq]['list'].extend([('n', inv) for inv in common_invs
                    ])
48
49              # If the complete sequence is unchanged, we're done.
50              if len(common_invs) == len(left_invs) == len(right_invs):
51                  diff[seq]['mode'] = 'n'
52                  continue
53
54              # Otherwise, the sequence is changed.
55              diff[seq]['mode'] = 'c'
56
57              # Avoid compare again unchanged invariants
58              left_invs -= common_invs
59              right_invs -= common_invs
60
61              repeated_invs = set()
62
63              # Find each different invariant by name an separator.
64              # left vs right
65              for self_inv in left_invs:
66                  found = False
67                  for other_inv in right_invs:
68
69                      found = (self_inv[:2] == other_inv[:2])
70                      found = found and self_inv not in repeated_invs
71                      found = found and other_inv not in repeated_invs
72
73                      # Same name and sep but different value (c, change)
74                      if found:
75                          diff[seq]['list'].append(('c', self_inv,
                                other_inv))
76                          repeated_invs.add(self_inv)
77                          repeated_invs.add(other_inv)
78                          break
79
80                  # Not found (-, deleted)
81                  if not found and self_inv not in repeated_invs:
82                      diff[seq]['list'].append(('-', self_inv))
83
```

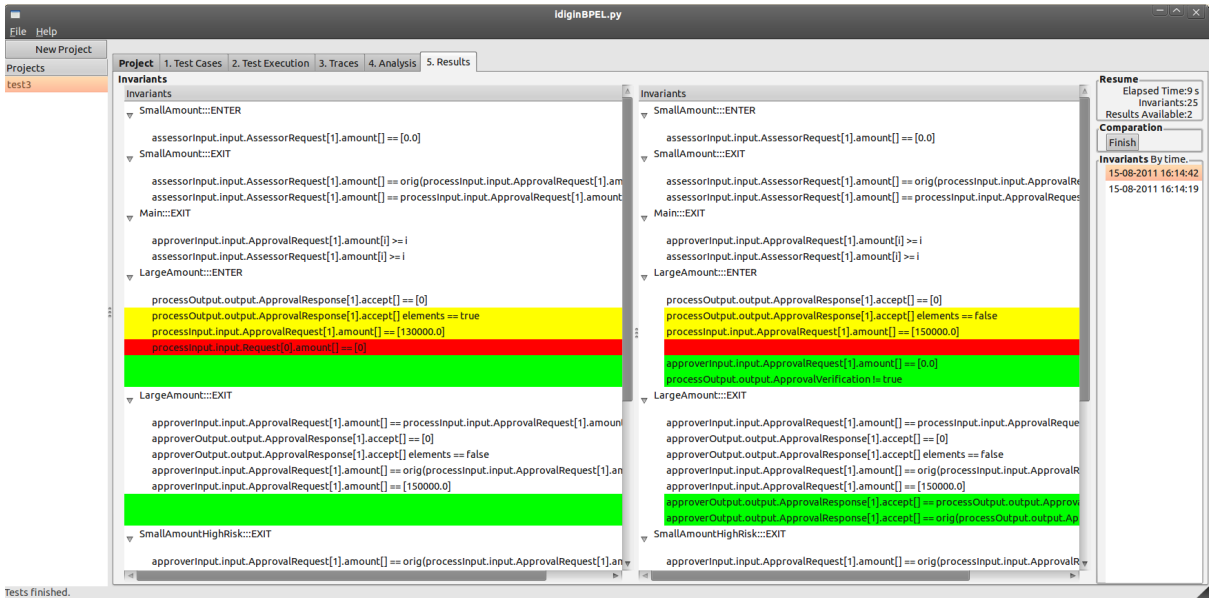


Figura 5.6: Muestra de la interfaz de comparación de invariantes de *IdiginBPEL*

```
84                # Second pass for new invariants.
85                # other vs self
86                for other_inv in right_invs:
87                    found = False
88
89                    for self_inv in left_invs:
90                        found = self_inv[:2] == other_inv[:2]
91                        found = found and self_inv not in repeated_invs
92                        found = found and other_inv not in repeated_invs
93
94                        # Same name and sep but different value (c, change)
95                        if found:
96                            diff[seq]['list'].append(('c', self_inv,
                                other_inv))
97                            repeated_invs.add(self_inv)
98                            repeated_invs.add(other_inv)
99                            break
100
101                   # Not found (+, new)
102                   if not found and other_inv not in repeated_invs:
103                       diff[seq]['list'].append(('+', other_inv))
104
105           return diff
```

La comparación muestra las diferencias según un sencillo código de colores. Si un invariante es nuevo, es decir, no se encontraba a la izquierda pero si a la derecha, será marcado como verde. Si existía en el lado izquierdo pero ha cambiado, se marcará en amarillo, y si está a la izquierda pero no a la derecha, se considerará eliminado y se marcará en rojo. Puede verse un ejemplo de la interfaz en la imagen que se encuentra en la figura 5.6 (página 96).

5.1.8. Internacionalización

La siguiente funcionalidad a añadir se trata del soporte para multilenguaje e internacionalización de la aplicación. Esto se realizó mediante el empleo de la herramienta *GNU* gettext, el cual dispone de un módulo de *Python* a tal efecto. El uso de gettext es sencillo, únicamente hay que emplear la cadena de traducción a modo de identificador a la hora de llamar a una función, y esta se verá traducida si la configuración de la aplicación señala otro idioma como el vigente. Un ejemplo de esto puede verse en el listado de la figura 5.13.

Listado 5.13: Ejemplo de uso de gettext en *IdiginBPEL*.

```
1  try:
2      fich, caso, time = traza.split(':')
3      time = time.rsplit('.',1)[0]
4  except ValueError:
5      log.warning(_("Hay una traza que no sigue el formato: " + traza))
6      return "","",""
```

A la hora de obtener todas las cadenas a traducir de la aplicación y generar una plantilla que pueda ser manejada por humanos para añadir la correspondencia en otro idioma de cada frase, se emplea la herramienta `intltool-extract`, que genera ficheros `.h` a partir de ficheros de código fuente con llamadas a gettext o bien ficheros *Glade*. Tras obtener todos los ficheros `.h` necesarios, se utiliza el comando `xgettext` para generar un esqueleto de plantilla `.pot` con todas las cadenas y su posible traducción vacía a falta de ser escrita.

Una vez se dispone de una plantilla `.pot`, es necesario crear una traducción específica para cada idioma, que es un fichero de traducción `.po`. Si no es la primera vez que se genera la plantilla, sino que se trata de una actualización de la plantilla con nuevas cadenas que antes no estaban, es necesario actualizar las traducciones `.po` a fin de no perder el progreso ya realizado; para esta labor puede utilizarse la herramienta `msgmerge`, que añade las nuevas cadenas a un fichero de traducción `.po` que ya contiene pares de traducción, que de otro modo se perderían.

Estos ficheros `.po` es necesario que sean convertidos al formato binario propio de gettext a fin de que gettext maneje las traducciones durante la ejecución de la aplicación. Para convertir una plantilla `.po` a binario (`.mo`) se emplea la herramienta `msgfmt`.

Todo este proceso completo se halla automatizado de forma que no es necesario recordar y teclear sin errores una larga serie de órdenes de consola para sencillamente actualizar las traducciones. La automatización se realizó mediante la programación de un guión *bash* disponible bajo el nombre de `trad.sh` en el CD adjunto a esta memoria y que maneja la extracción de cadenas, la actualización de plantillas, su conversión a binario, su limpieza y la generación de un fichero de contexto de traducción, que busca en los ficheros fuente las cadenas a traducir y proporciona varias líneas de código que las rodean, dotando así de una guía al traductor, que puede ver esas cadenas de texto en su contexto original.

A diferencia de en *Takuan*, en el sistema ideado por *IdiginBPEL*, el proceso de instrumentación solo se realiza una única vez por proyecto, durante el proceso de creación del mismo. A la hora

de crear un proyecto, recordemos era necesario proveerle de un fichero *bpel*, y el sistema se haría cargo de sus dependencias[7]).

Como ya explicamos, a la hora de guardar estas dependencias importadas dentro del proyecto, estas se almacenaban en un directorio llamado `dependencias`, y las rutas de inclusión entre ellas eran reescritas de forma que todas encajasen. En este punto se detectó un error en el código del *Instrumentador* de *Takuan*; este empleaba siempre la misma ruta base a la hora de calcular las rutas a los diferentes componentes de las dependencias, fallando en los casos en los que el fichero `A` importaba al fichero `B` y este a su vez a un tercero `C`. La ruta relativa al fichero `C` era calculada incorrectamente en base al fichero `A`, en lugar de en base al `B` como hubiese sido correcto, incurriendo así en fallos a la hora de encontrar determinados ficheros.

Este error fue reportado a los desarrolladores de *Takuan* del grupo *SPI&FM* y Antonio García Domínguez resolvió el error reescribiendo el módulo que realizaba estas operaciones en el instrumentador.

5.1.9. Error en parámetros de preprocesado

Los guiones *Perl* resultaron tener deficiencias a la hora de tratar con ficheros de registro de ejecución en cuyo nombre había caracteres interpretables por la «shell» o espacios. De forma que se modificaron los ficheros *Perl* de forma que empleasen la orden `system` y evitasen ese tipo de problemas. Para controlar la salida del guión se les añadió un parámetro nuevo a todos los guiones de conversión con la ruta al archivo de salida donde debían realizar su salida. Estas modificaciones fueron enviadas en forma de una serie de parches, que fueron aplicados sobre el código fuente de *Takuan*.

Una vez solucionado el problema con los guiones de conversión, era ya posible obtener trazas en formato *Daikon* de forma sencilla a partir de los registros de ejecución obtenidos de las ejecuciones en el servidor *ActiveBPEL*. Tras eso, estas trazas convertidas son pasadas a *Daikon*, el cual realiza el estudio estadístico adecuado y genera un fichero final con los invariantes.

[7]Para más información sobre la búsqueda recursiva de dependencias, puede consultar la sección 5.1.4 (página 85.

CAPÍTULO 6

Conclusiones

El trabajo en este proyecto ha conseguido pues crear una herramienta gráfica, *IdiginBPEL*, la cual aporta la posibilidad de realizar la generación dinámica de invariantes basada en *Takuan* mediante el uso de una interfaz visual, sencilla y agradable, añadiendo funcionalidad extra que permite la ejecución de casos de forma masiva, la comparación de invariantes afinando de manera significativa el control que el usuario ejerce sobre el proceso general.

La colaboración del alumno con el grupo *SPI&FM* durante la realización de este *PFC*, dió lugar a la aportación de varios errores detectados y mejoras concretas al código fuente de *Takuan*, ayudando en cierta medida al perfeccionamiento de la misma.

Como experiencia personal, este desarrollo resulta el proyecto de mayor envergadura, más completo y exigente al cual el alumno se ha enfrentado, del cual se ha podido obtener una no despreciable experiencia. Este *PFC*, además, ha propiciado la inclusión del alumno como autor en su primer artículo de investigación, proporcionándole una mayor consciencia de la realidad del trabajo en el ámbito científico.

6.1. Publicaciones y premios

La aplicación fue merecedora del *Accésit al Mejor Proyecto Científico Libre* durante la fase local en la Universidad de Cádiz de la IV edición del Concurso Universitario de Software Libre en el año 2010 [Con11]. Este premio supuso un reconocimiento a la labor realizada trabajando sobre el desarrollo de la herramienta de código libre *IdiginBPEL*.

En cuanto a aportaciones científicas, *IdiginBPEL* se presentó en el 10th International Conference on Web Engineering en formato demo. Como resultado, el artículo *"A Tool for WS-BPEL Composition Testing Using Dynamic Invariant Generation"* se publicó en sus actas, dentro de la serie *Lecture Notes in Computer Science de Springer Verlag* [Pal10]. El congreso está indexado en diversos rankings bibliográficos, como *CORE*, *DBLP* o *Conference Proceedings Citation Index* de Thomson Reuters.

6.2. Trabajo futuro

Posibles ampliaciones de la herramienta y trabajos futuros que se proponen son:

- **Visualización de código**: Una posible extensión a *IdiginBPEL* podría incluir una herramienta de visualización y edición de código *WS-BPEL 2.0* que permitiese ver y editar la composición a analizar. De esta manera también se podría realizar un control mucho más fino de la instrumentación automática, ajustando manualmente los aspectos del código sobre los que se va a generar registros de ejecución así como editar los casos de prueba unitarios de *BPELUnit* sobre la marcha, mejorando mucho el proceso de mejora de los mismos, al agilizar el ciclo de probar un caso, editarlo y volver a analizar.
- **Cobertura de caminos**: *Takuan* incluye mejoras en este sentido [Á10] que no se ven reflejadas en *IdiginBPEL*. Podría ampliarse la herramienta para que mostrase de forma adicional la cobertura realizada a cada rama del código ejecutado, mejorando así la comprensión sobre los casos de prueba unitarios de *BPELUnit* en uso, permitiendo su mejora, así como una comprensión más profunda de los invariantes generados.
- **Nuevas tecnologías basadas en *WS-BPEL 2.0***: Existen ya una variada serie de tecnologías que, empleando *WS-BPEL 2.0* como base, lo han respaldado y lo amplían. Ejemplos son *BPMN* (*Business Process Modelling Notation*, una notación gráfica estandarizada para modelos de negocio) la cual ha sido ampliamente aceptada y permite la generación de código *WS-BPEL 2.0* ejecutable a partir de modelos gráficos [Bus10]. Junto a *BPMN* se encuentran *BPEL4People* y *BPELScript* que implementan interacciones humanas en procesos de negocio y facilitan la creación de composiciones mediante un lenguaje más flexible, respectivamente. A medida que la aceptación de cada herramienta mencionada fuese creciendo, podría estudiarse la posibilidad de *IdiginBPEL* de trabajar con ellas.
- **Clasificación de Invariantes**: Una ampliación interesante para *IdiginBPEL* podría mostrar los diferentes tipos de invariantes generados pero descartados, así como clasificar de forma visual los ya generados mostrando el nivel de confianza que se les estima.
- **Integración con el S.O.**: Tomando la fase de análisis estadístico realizada por *Daikon* en ocasiones un tiempo considerable, podría integrarse *IdiginBPEL* con las notificaciones del sistema, de forma que informe al usuario del mismo de las diferentes etapas por las que pasa el proceso y avisarle de ellas, así como de su término.

- **Integración con otras herramientas**: Se podría realizar un estudio sobre la posibilidad de comunicar o integrar de alguna forma *IdiginBPEL* con *Gamera* [Jim09], herramienta de pruebas de mutación desarrollada por el grupo *SPI&FM*.
- **Operatividad con un servidor *ActiveBPEL* remoto**: Actualmente el sistema interno de *IdiginBPEL* exige, al igual que *Takuan*, que el servidor *ActiveBPEL* contra el cual se realizan las ejecuciones de casos de prueba unitarios de *BPELUnit*, se encuentre accesible en la misma máquina en donde se encuentran realizando las pruebas. Sería conveniente ajustar *IdiginBPEL* y/o *Takuan* de forma que pudiesen operar con un servidor remoto.

APÉNDICE A

Herramientas

A.1. Python

A.1.1. Historia del lenguaje

Python fue inventado en 1990 por Guido van Rossum, cuando se encontraba en el *CWI* (*Centrum Wiskunde & Informatica*) en Amsterdam. Su nombre proviene de la serie de comedia de la *BBC Monty Python's Flying Circus* [Wik11c] de la cual Guido es seguidor. De acuerdo con la historia, Guido se encontraba viendo reposiciones del programa más o menos al mismo tiempo en que el necesitaba un nombre para el nuevo lenguaje que estaba desarrollando.

Debido a esta herencia recibida de los *Monty Python*, ciertas referencias al grupo de cómicos suelen aparecer dentro de la comunidad de desarrolladores. Palabras como «spam», «lumberjack» y «shrubbery» tienen connotaciones especiales para los usuarios de *Python*, y por ejemplo, los debates y enfrentamientos son normalmente denominados como «The Spanish Inquisition». Como regla general, si un usuario de *Python* comienza a usar frases que no tienen relación alguna con la realidad, probablemente las esté sacando de la series de los *Monty Python* o de sus películas.

Guido, en el momento de la creación del lenguaje, también se encontraba implicado en el desarrollo del sistema operativo distribuido *Amoeba* y el lenguaje *ABC*. De hecho, su motivación original para la creación de *Python* era la creación de un lenguaje de «scripting» lo suficientemente avanzado para el sistema *Amoeba*.

Tras la creación original del lenguaje, el ”efecto red” resultó en la distribución de las primeras versiones de *Python* por todo el mundo y fué adoptado progresivamente en multitud de labores diferentes. Desde que apareció al dominio público en 1991, *Python* ha conseguido atraer una gran cantidad de leales seguidores, inicialmente organizados en un grupo de noticias en Internet, *complang.lang.python* en 1994, actualmente la *PSP* (*Python Software Foundation*) posee la propiedad intelectual de *Python* y coordina las actividades de la comunidad así como el *PBF* (*Python Business Forum*).

Una buena muestra de la filosofía de *Python* y sus principales características viene en el denominado *Zen of Python*, escrito por Tim Peters, y que puede ser visualizado ejecutando la siguiente orden en una consola de *Python*:

```
>>> import this
```

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
Algthough that way may not be obious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explan, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!

Zen of Python, por Tim Peters

A.1.2. El lenguaje de programación *Python*

Python [Lut09] es un lenguaje de programación de propósito general licenciado como software libre y optimizado para ser productivo, portable, proveer calidad al software y facilitar la integración. Es usado por cientos de miles de desarrolladores en el mundo en áreas como «scripting» para sistemas, programación de servidores, interfaces de usuario...

Como lenguaje cuyos principales esfuerzos van en la dirección de tratar de reducir todo lo posible el tiempo de desarrollo, es usado en todo tipo de roles: Administración de sistemas, desarrollo de sitios web, «scripting» para teléfonos móviles, educación, «testing» de «hardware», análisis de inversiones, videojuegos y guiado de naves espaciales.

Entre otras cosas, *Python* ofrece Programación Orientada a Objetos, una sintaxis extremadamente simple, legible y mantenible. Integración con componentes en *C* y una enorme colección de utilidades e interfaces ya hechas y probadas. Su conjunto de herramientas integrado lo hace ser un lenguaje flexible y ágil, ideal tanto para tareas rápidas como para largos desarrollos de mayor calado. El lenguaje favorece:

- **Calidad del software**: Haciendo que el software creado con el pueda ser entendido, reutilizado y modificado de forma sencilla.
- **Productividad**: *Python* se encuentra optimizado para mejorar la velocidad de desarrollo, es fácil escribir programas de forma rápida; esto lo hace flexible y ágil.
- **Portabilidad**: Los programas en *Python* se encuentran disponibles en prácticamente cualquier plataforma (*Mac*, *Linux*, *Windows*) y arquitectura disponible hoy en día y pueden ejecutarse en ellas sin (prácticamente) modificaciones ni cambios.
- **Integración de Componentes**: Está diseñado para ser integrado con otras herramientas; los programas escritos en *Python* pueden ser fácilmente mezclados y se puede guionizar y dirigir de forma sencilla a otros componentes del sistema.

A pesar de su carácter de propósito general, *Python* es usualmente llamado un lenguaje de «scripting», porque hace fácil el reutilizar directamente otros componentes de «software». Quizás la virtud más destacable de *Python* es sencillamente que ofrece la posibilidad de realizar desarrollar software de forma fácil y agradable.

A.2. Git-Svn

Git y *Subversion* son dos herramientas con una orientación muy diferente entre si que las hace incompatibles en muchos aspectos. Trabajar con ambas herramientas al mismo tiempo puede no ser un proceso directo y estar lleno de impedimentos. Sin embargo, a veces es algo completamente necesario, de forma que en las siguientes secciones se mostrará como transportar un repositorio *svn* a *Git* y como mantener actualizado un repositorio *svn* a partir del trabajo realizado en otro proyecto *Git*.

A.2.1. De *svn* a *git*

En esta sección se listan los pasos a seguir para conseguir exportar un repositorio manejado bajo *Subversion* a un repositorio vacío en *Git*.

1. Instalar el paquete *git-svn*

   ```
   $ sudo apt-get install git-svn
   ```

2. Clonar el repositorio *svn* creando uno nuevo en *git*.

   ```
   $ git svn clone <url-repo-svn>
   ```

Esta secuencia de ordenes habrá creado un repositorio *Git* con todos los *commits* realizados en el repositorio svn. Si nos fijamos en el fichero de configuración situado en `.git/config` tendremos lo siguiente, como podemos ver:

```
[svn-remote "svn"]
    url = <url-repo-svn>
    fetch = :refs/remotes/git-svn
```

A.2.2. ¿Cómo trabajar con él?

Sencillamente podríamos decir que `svn update` es ahora `git svn rebase`. La orden es `git svn rebase` en lugar de `git svn merge` debido a que se sigue usando la lógica de svn aún estando en *Git*. En *Git* la orden `merge`, une diferentes ramas en una sola, creando un nuevo commit con la información de la mezcla realizada entre las dos o más ramas que intervienen. La mezcla se realiza buscando buscando en las ramas hasta encontrar un ancestro común a todas, y no reescribe los commits anteriores, solo añade información. Cuando no se necesita realizar una búsqueda de ancestros comunes ya que una rama se encuentra basada en otra, lo único que debe hacer *Git* es avanzar el puntero a la última revisión de una rama («HEAD»). Esto se denomina una mezcla «fast-forward».

Cuando se realiza `svn update` lo que se hace es aplicar los parches al trabajo actual situado en el repositorio. La orden `rebase` hace algo parecido, replantear una rama en base a otra significa calcular todos los cambios que se han hecho con respecto a esa otra rama y aplicarlos encima, , reescribiendo los commits como si siempre hubiesen estado así.

A la hora de realizar commits, `svn commit` ahora es `git svn dcommit` Este tipo de commit (`git svn dcommit`) tiene un inconveniente enorme, y es que NO se manejan commits de merge. Es decir, no es posible mandar al repositorio *svn* commits resultado de mezclar dos ramas, debe hacerse un rebase siempre de forma previa al commit.

Este tipo de commits son peligrosos, falsean el historial de *svn* y pueden llegar a ocasionar problemas en el servidor *Subversion* si se envían commits de merge, llegando a romperlo. Se desaconseja el mantener ambos sistemas de control de versiones al mismo tiempo. Si aún así se desea realizar esta tarea, en la siguiente sección se provee de un flujo de trabajo bien probado para realizar la sincronización de forma segura.

A.2.3. Trabajando de forma efectiva con *svn* y *git*

El problema viene en el momento en que *svn* maneja los conflictos y los merges de manera diferente, de forma que enviar un *merge commit* a un repositorio *svn*, es una fuente de problemas. La solución alternativa a `merge` en *Git*, es rebase. El funcionamiento de `rebase` es, fusionar dos branches, de manera que al final no es que se unan mediante un commit que resuelve las diferencias, sino que a cada commit anterior de uno de los branches, se le aplican todas las diferencias que tiene con la otra, reescribiendo su historia. Al final de un rebase, se termina con el branch desde el que se inició, con sus commits reescritos añadiendo lo que tenía el otro branch, es como si se hubiese escrito así desde un principio.

Reescribir la historia, es decir aplicar parches hacia atrás, es peligroso cuando se trabaja también con un repositorio remoto, ya que si se modifican commits que ya han sido enviados, y alguien cambia la historia, se generarán muchos problemas y conflictos en commits antiguos. Por eso en definitiva no es buena idea y es mejor ir haciendo merge en master y otras ramas, y **separando de esta forma los rebases en una rama aparte específica para svn**.

La metodología planteada se compone de 3 ramas básicas, las cuales se usan para separar el desarrollo y las operaciones de commits:

- **master**: La normal. Solo actualizada con versiones estables.
- **work**: Se usa para trabajo diario.
- **svn**: Se usará para sincronizar con el repositorio *svn*.

El «workflow» consiste en trabajar en la rama `work`, realizando operaciones `merge` a `master` para actualizar esta última rama. Una vez que existe algo estable en `master`, se realiza un `rebase` hacia la rama `svn`, empleando posteriormente el comando `git svn dcommit` para enviar todos los commits a los que se les ha hecho rebase al servidor para sincronizarlo.

Un ejemplo real. Partiendo de unas ramas `master`, `work` y `svn` limpias, sincronizadas, todas iguales en su último commit (salvo la rama `svn`, la cual debido a las operaciones `rebase` mantiene un historial diferente). En este contexto, se entra en la rama de trabajo, `work` y añadimos una pequeña «feature».

```
$ git checkout work
$ vim idgui/proyecto.py
```

Se realizan algunos cambios...

```
$ git diff
(.. 20 o 30 líneas de cambios ..)

$ git status
# On branch work
# Changed but not updated:
#   (use "git add ..." to update what will be committed)
#   (use "git checkout -- ..." to discard
        changes in working directory)
#
#       modified:   idgui/proyecto.py
```

Para posteriormente hacer `commit` de ellos.

```
$ git commit -am "Añadido contador de casos
seleccionados a la pantalla de casos"
[work f70e216] Añadido contador de casos
seleccionados a la pantalla de casos
 1 files changed, 10 insertions(+), 5 deletions(-)
```

Nos encontramos así 1 commit por delante de la rama `master`. Vamos a sincronizarlos. Para ello cambiamos a la rama `master`, y ordenamos mezclarla con la rama `work`.

```
$ git checkout master
Switched to branch 'master'
$ git merge work
Updating 6832ef9..f70e216
Fast forward
 trunk/idgui/proyecto.py |   15 ++++++++++-----
 1 files changed, 10 insertions(+), 5 deletions(-)
```

Operación sencilla, pues ha sido «fast-forward», *Git* solo tenido que mover el `HEAD`. En este estado encontramos código estable en la rama `master`, y procederemos a hacer `commit` también hacia el repositorio *Subversion*. Para ello, se necesita sincronizar también la rama `svn`, pero como no es apropiado realizar una operación `merge`, se realizará un `rebase`:

```
$ git checkout svn
Switched to branch 'svn'
$ git rebase master
First, rewinding head to replay your work on top of it...
Fast-forwarded svn to master.
```

Hemos obtenido en este momento unas ramas `work`, `master` y `svn` con HEADS iguales y (En la rama `svn` el código se encuentra actualizado pero el HEAD no es el mismo debido a la operación de `rebase`). Procedemos a mandar un el commit al repositorio *Subversion*.

```
git svn dcommit
Committing to https://forja.rediris.es/svn/cusl4-idigin ...
Authentication realm:  Subversion User Authentication
Password for 'frajasal':
        M       trunk/idgui/proyecto.py
Committed r245
        M       trunk/idgui/proyecto.py
r245 = 1ded21df0c8c764605d9b505c9ca6738609aaf53 (git-svn)
```

¡Listo! Disponemos de ambos repositorios con el mismo código actualizado y bien sincronizados.

Manteniendo de forma estricta el «workflow» descrito, trabajando siempre sobre una rama aparte para después hacer `merge` contra `master` e inmediatamente `rebase` con la rama `svn`, no debería haber problemas de reescritura de commits anteriores en el repositorio *Subversion*, ya que las ramas `master` y `svn` irían avanzando al mismo tiempo, son colisiones.

A.3. *Python* y *Doxygen*

La herramienta de generación automática de documentación *Doxygen*, permite obtener documentación a partir del código. A través de comentarios que tienen una forma especial, se documenta directamente sobre el código que escribimos, y la herramienta extrae toda la información automáticamente de forma que la presenta en varios posibles formatos, aunque por defecto genera *html* y *pdf*.

Tiene muchos detalles. Referencias cruzadas, índices, páginas aparte, código de ejemplo, diagramas mediante *graphviz*... Sin apenas esfuerzo se obtiene una documentación amigable y si se han seguido buenas prácticas de programación dentro del mismo código de la aplicación, el código es mucho más legible y la documentación se obtiene sin esfuerzo.

Doxygen por defecto no se lleva demasiado bien con *Python*. Primero, porque está programado para lenguajes que utilizan bloques cerrados entre llaves, y siendo *Python* un lenguaje sin ese azúcar sintáctico, no funciona correctamente. Otro gran defecto, es que los comentarios *Doxygen* se sitúan justo encima de la declaración de la variable o función que se quiera documentar, mientras que *Python* provee de su propio sistema de autodocumentación integrado en el lenguaje mediante las llamadas «docstrings», las cuales se disponen justo debajo de la declaración.

Llegados a ese punto se decidió no llevarle la contraria al lenguaje. Viendo que el mismo ya provee de un sistema para que documentación en «docstrings» accesibles en tiempo de ejecución e integradas, ¿debemos dejar de usarlas y pasarnos al bloque de comentarios sobre las funciones habitual de *Java*, *Perl* o *C++*?. La ayuda necesaria para mantener el uso de las «docstrings» y *Python* la encontré en forma de un pequeño guión llamado *doxypy* [Gre11], consistente en un filtro que se invoca desde *Doxygen* para que modifique el código *Python*, extraiga comentarios y realice varios arreglos de manera que *Doxygen* lo trate de manera parecida a código *Java*.

A.3.1. Uso de *Doxypy*

Doxypy se encuentra disponible para descarga en su página [Gre11], los pasos para su uso son:

1. Descomprimir el directorio de *Doxyfile*
2. Modificar la siguiente sección del fichero de configuración de *Doxygen*:

   ```
   FILTER_SOURCE_FILES     = YES
   INPUT_FILTER            = "python doxypy/doxypy.py"
   ```

   teniendo en cuenta que las rutas de este fichero son relativas al propio fichero de configuración *Doxyfile*. Esta opción nos permite activar el filtro.
3. *Adicionalmente* se pueden activar opciones específicas para *Java*, que mejoran la salida resultante de la autodocumentación para *Python*.

   ```
   OPTIMIZE_OUTPUT_JAVA    = YES
   ```

Esta línea permite activar ciertas opciones de su representación típica en *C++* a otra en *Java* mucho más, similar a *Python*, como por ejemplo con respecto a la jerarquía de clases, cuya notación pasaría de ”Padre::Hijo” a ”Padre.Hijo”.

También se avisa que para las «docstrings» de los módulos, que no son detectadas por defecto, se debe introducir una orden `@namespace` en ellas con el nombre del módulo, y esta declaración hará el truco para que aparezcan.

APÉNDICE B

Manual de *IdiginBPEL*

B.1. Introducción

IdiginBPEL permite obtener invariantes dinámicos de una composición *WS-BPEL 2.0* de la ejecución masiva de pruebas sobre la misma. La herramienta provee de un entorno gráfico en el que pueden seleccionarse las pruebas a realizar, ejecutarlas contra un servicio real y analizar las trazas resultantes de la ejecución para obtener así los invariantes requeridos.

Como requisitos técnicos *IdiginBPEL* necesita para su ejecución de un entorno *Linux* y requiere *Takuan* instalado en el sistema, ya que este se emplea como base a partir del cual se automatizan los pasos.

Este documento tiene dos partes diferenciadas.

- *Guía Rápida*: Permite instalar y usar el programa en breves pasos.
- *Manual de Uso*: Muestra y explica en detalle la instalación, el formato y la utilización del programa.

Para una instalación y uso inmediato de la aplicación, la *Guía Rápida* resultará suficiente. En caso de problemas o de la necesidad de modificar detalles concretos en la aplicación, refiérase al *Manual de Uso* que incluye la descripción completa.

Figura B.1: Pantalla principal de *IdiginBPEL*

B.2. Guía rápida

B.2.1. Instalación

La instalación de *IdiginBPEL* se encuentra automatizada mediante el uso de un guión de instalación que comprueba las dependencias así como descarga e instala *IdiginBPEL* y su entorno necesario en el sistema.

El guión puede encontrarse en la forja del proyecto [Jav10]. Para instalar *IdiginBPEL* tan sólo es necesario ejecutar el guión en un directorio con permisos de escritura para el usuario, de la siguiente manera.

```
$ python install.py both
```

El guión se ocupará de iniciar el proceso de instalación de *Takuan*, y posteriormente, del de *IdiginBPEL*.

B.2.2. Uso

IdiginBPEL está basado en *proyectos*, consistentes en un único fichero de código fuente *WS-BPEL 2.0 .bpel* a analizar, junto a los ficheros con las pruebas necesarias.

Los pasos de la aplicación son:

1. Creación del *Proyecto*.
2. Añadido de *Casos de Prueba*.
3. *Ejecución* en servidor *ActiveBPEL*.
4. *Análisis* de las trazas generadas.

Para comenzar, arranque el programa desde una consola:

```
$ python idiginBPEL.py
```

Creación del proyecto

Cree un proyecto, para lo cual necesitará especificar un nombre para el mismo y la ruta a un archivo *WS-BPEL 2.0* como puede verse en la Figura B.2. Tras pulsar en el botón *Crear*, el proyecto quedará creado como puede verse en Figura B.3.

Casos de prueba

Una vez creado el proyecto, debe añadir los casos de prueba necesarios para ejecutar el fichero de código fuente *WS-BPEL 2.0*, como puede verse en la Figura B.4.

Los casos de prueba añadidos aparecen listados agrupados por fichero. Los casos marcados serán ejecutados al pulsar el botón de *Ejecutar*.

Ejecución

Para poder realizar la ejecución de los casos de prueba es necesario que el servidor *ActiveBPEL* se encuentre activo de forma que pueda desplegarse el fichero *WS-BPEL 2.0* en el. Esto puede hacerse pulsando en el botón «Start» como puede verse en la Figura B.6.

Una vez arrancado *ActiveBPEL*, se puede iniciar la ejecución pulsando en el botón *Ejecutar*. En ese momento los casos empezarán a probarse contra el servidor uno a uno, mostrando en cada caso si pasaron o no. Esto puede verse en B.7.

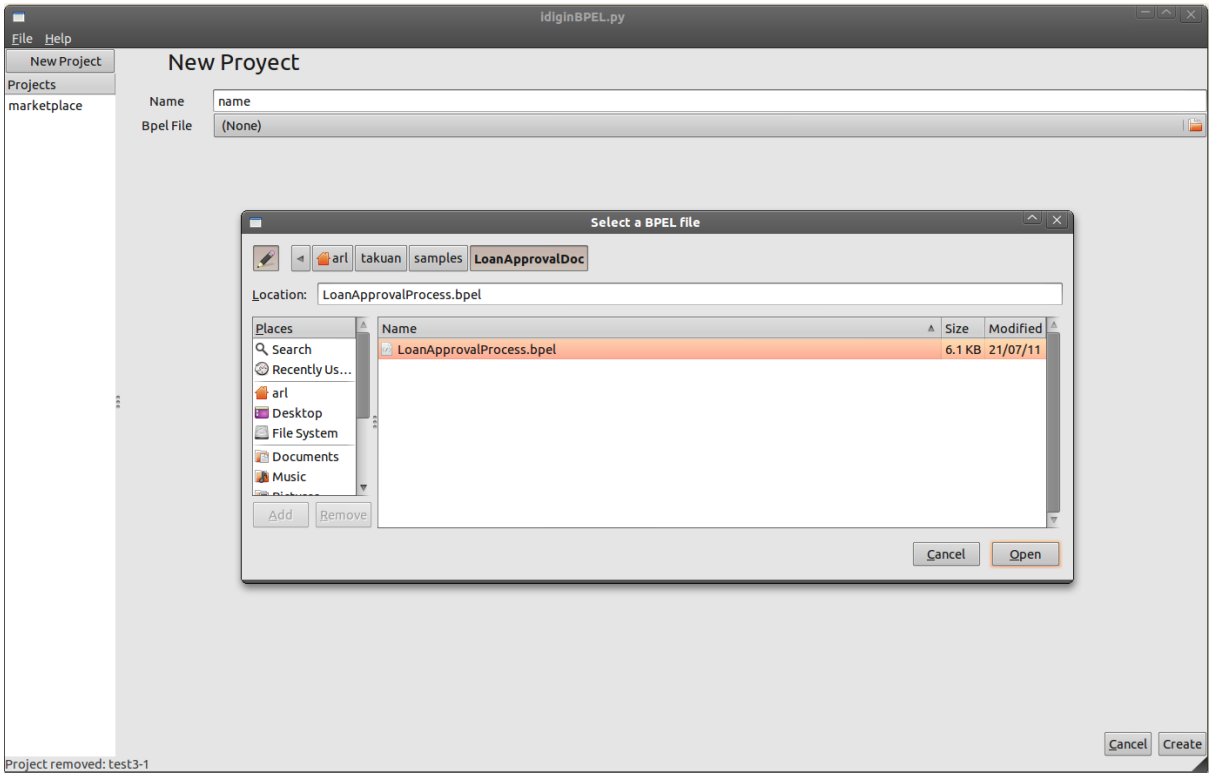


Figura B.2: Pantalla de creación de proyecto en *IdiginBPEL*

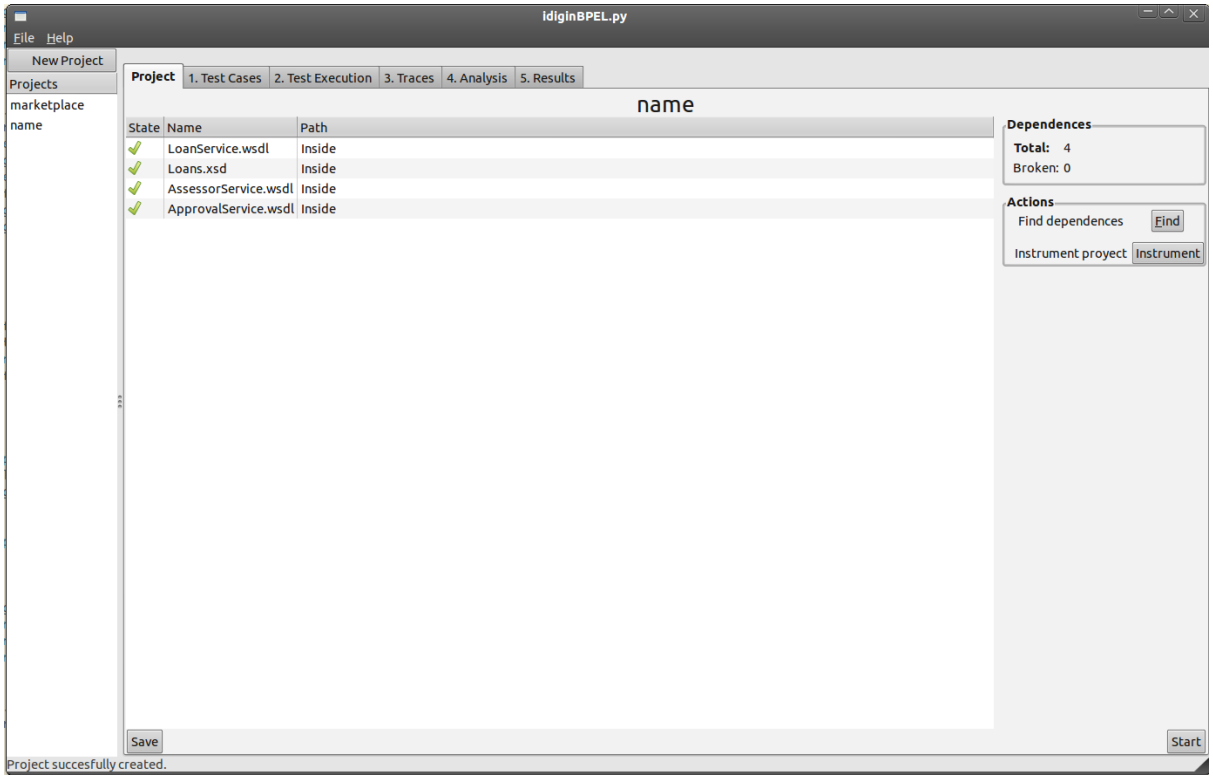


Figura B.3: Pantalla de proyecto creado en *IdiginBPEL*

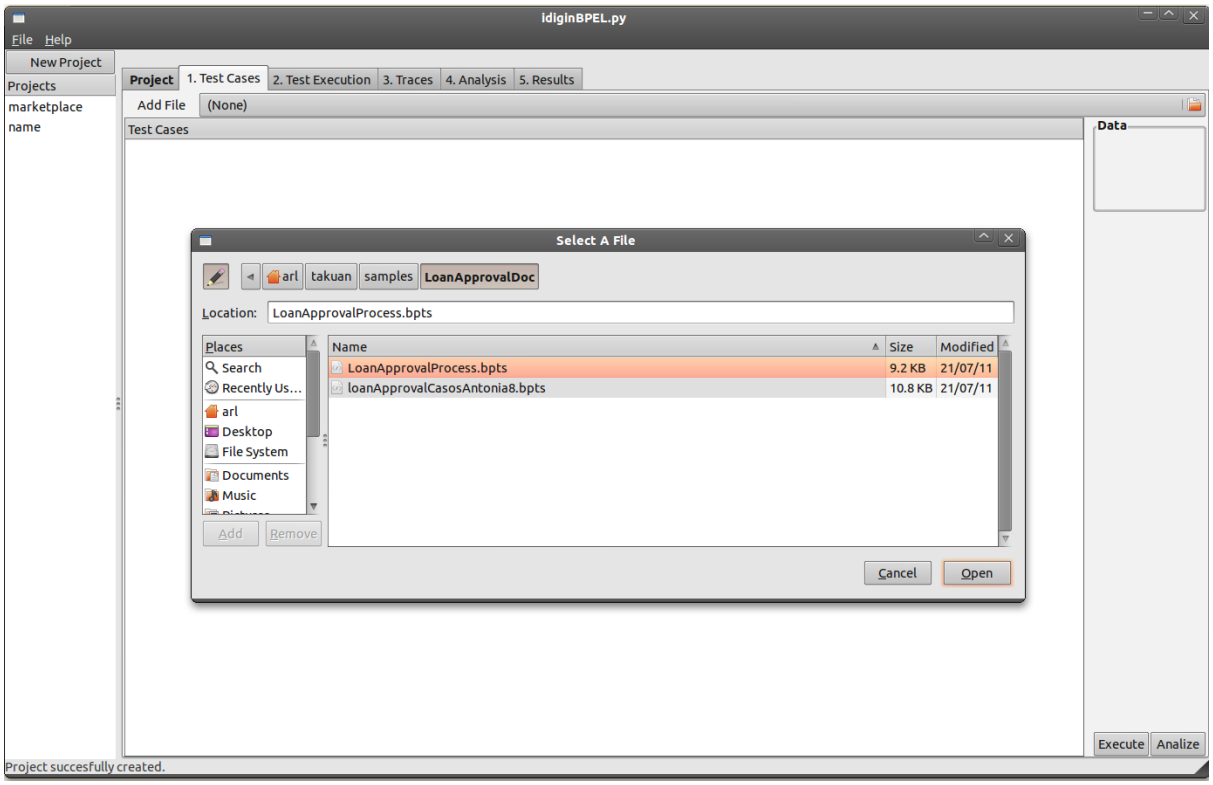


Figura B.4: Pantalla para añadir casos de prueba en *IdiginBPEL*

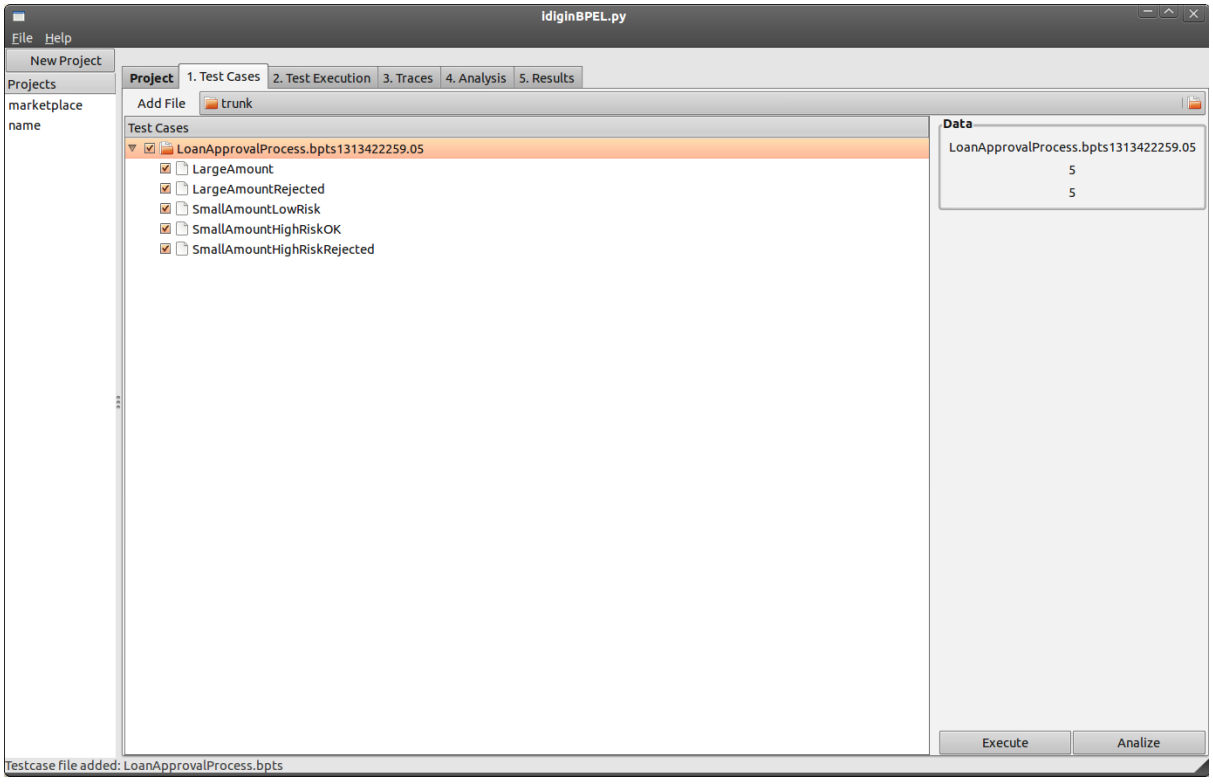


Figura B.5: Pantalla de selección de casos en *IdiginBPEL*

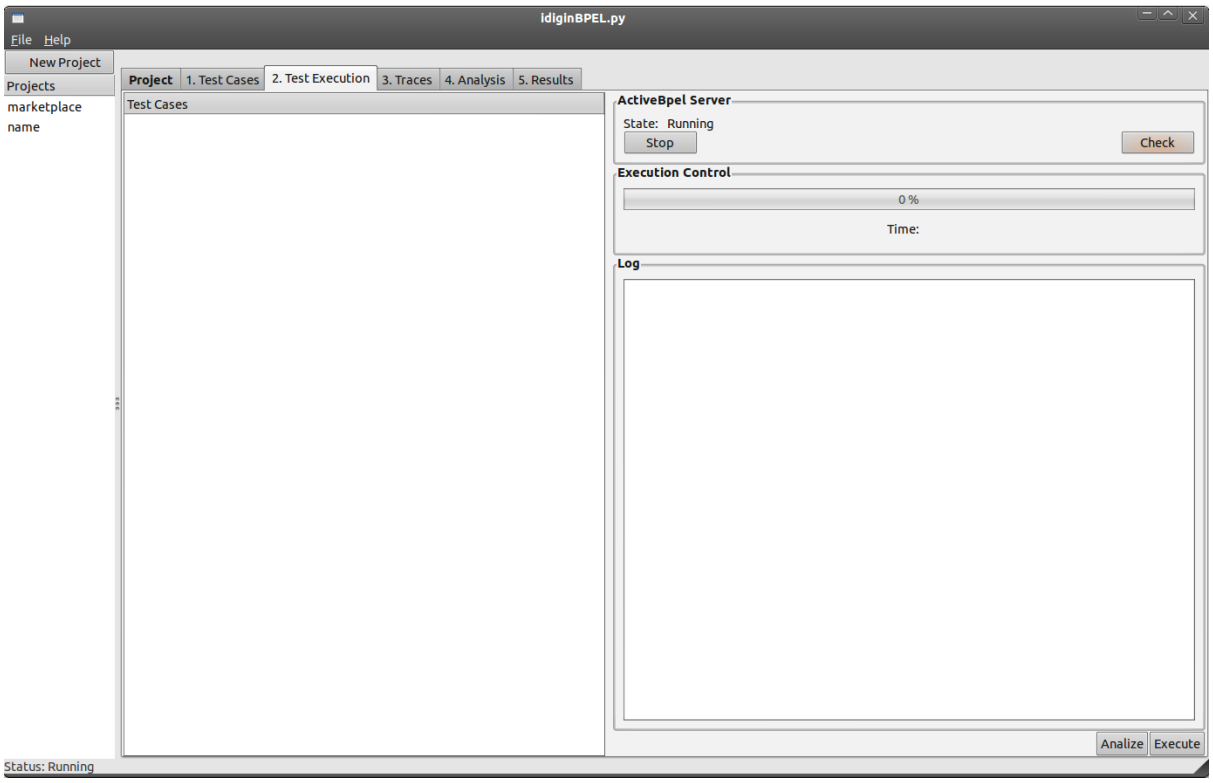


Figura B.6: Pantalla de activación del servidor *ActiveBPEL* en *IdiginBPEL*

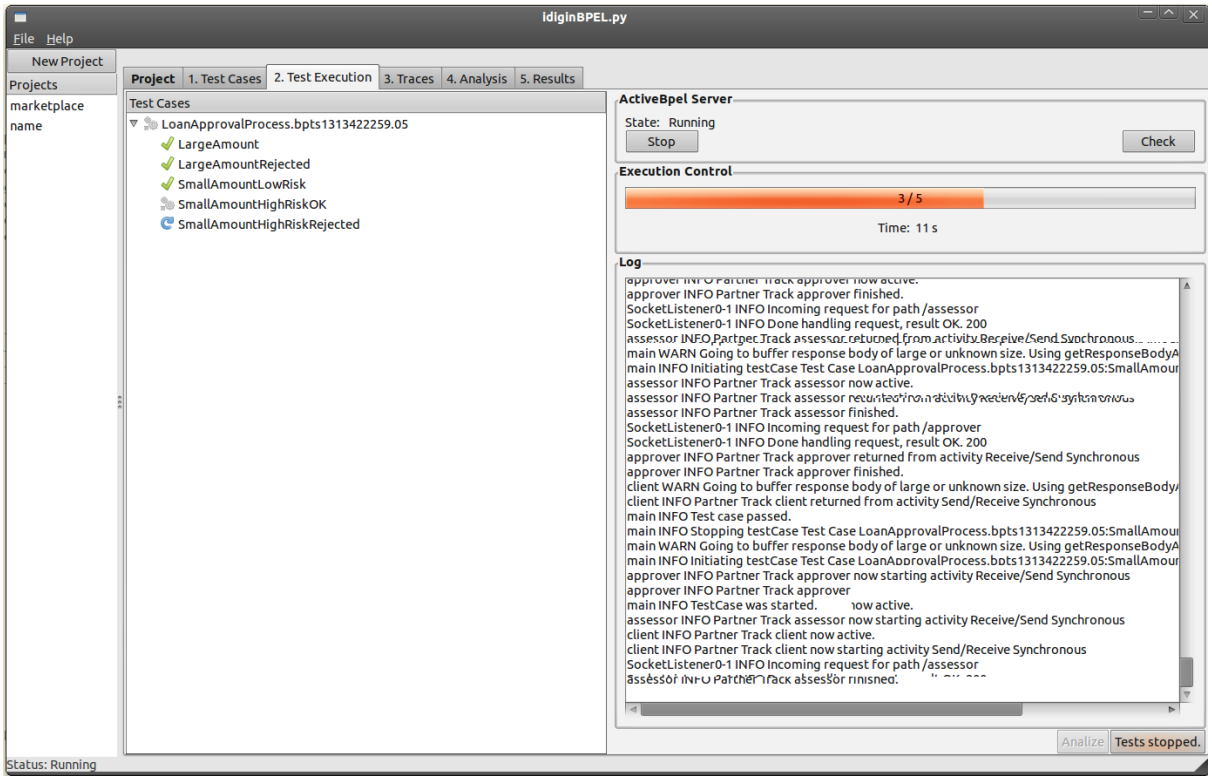


Figura B.7: Pantalla de ejecución de casos de prueba unitarios *BPELUnit* en *IdiginBPEL*

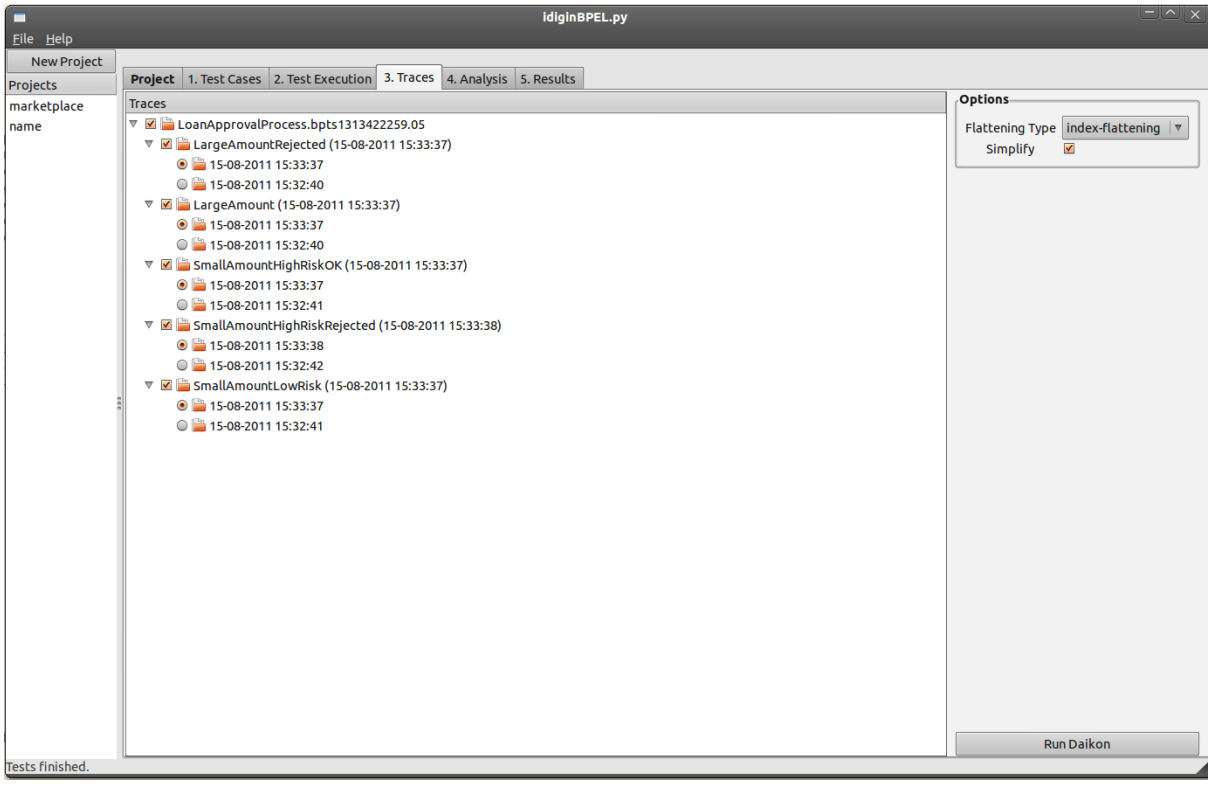


Figura B.8: Pantalla de selección de trazas generadas por las ejecuciones de los casos de prueba en *IdiginBPEL*

Análisis

Durante la ejecución de estos casos de prueba, se generan trazas con información sobre el comportamiento del programa. Estas trazas son almacenadas y pueden verse en la pantalla de selección de trazas en la Figura B.8.

Puede seleccionar las trazas que desea que sean analizadas a la hora de inferir los invariantes del programa. Una vez haya realizado su selección puede pulsar el botón de *Análisis* para que el motor estadístico *Daikon* trabaje con ellas y extraiga los invariantes potenciales.

Daikon realiza un análisis estadístico sobre las trazas generadas y obtiene los invariantes buscados. Es un proceso costoso que puede tomar mucho tiempo de proceso. Puede vigilar el «log» generado por *Daikon* en busca de información de progreso y posibles errores como puede verse en la figura B.9.

Una vez terminado el proceso de análisis, los invariantes resultantes son mostrados como puede verse en la Figura B.10.

Tras este último paso, los invariantes han sido generados satisfactoriamente y el proceso ha terminado. El sistema permite que pueda volver a ejecutar exactamente el mismo proceso, en caso de que la composición *WS-BPEL* dependa de elementos externos o aleatorios, añadir casos

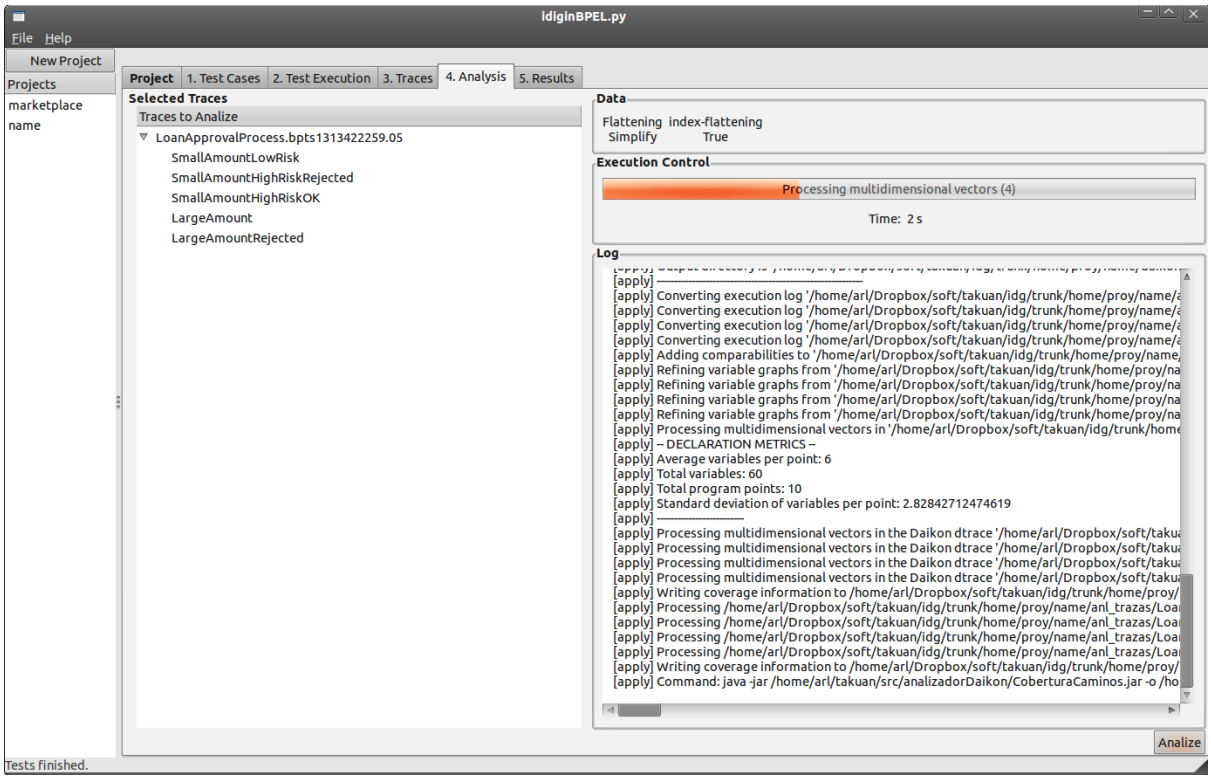


Figura B.9: Pantalla de configuración de la fase de Análisis de las trazas generadas por la ejecución en el motor estadístico *Daikon*

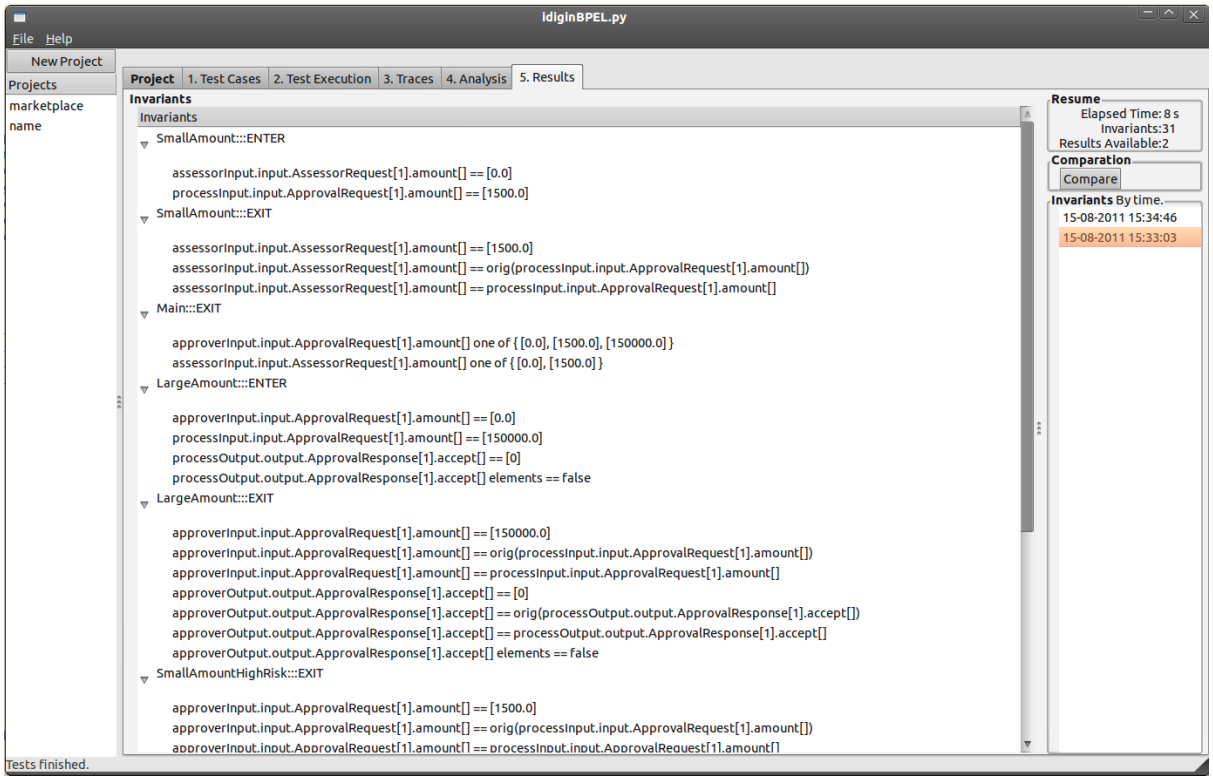


Figura B.10: Invariantes generados por la aplicación

de prueba extra, volver a ejecutar casos de prueba distintos o analizar juntas trazas de distintas ejecuciones.

Este es el final de la guía rápida introductoria a *IdiginBPEL*. Para mayor detalle e información extra sobre el programa y el proceso de generación de invariantes, consulte el *Manual de Uso*.

B.3. Manual de uso

B.3.1. Instalación

Dependencias

Los requisitos de *IdiginBPEL* son los siguientes:

- Python 2.5+ (python2.5)
- GTK 2.0 (libgtk2.0-0)

- Python GTK bindings (python-gtk2)
- Webkit (python-webkit)

Takuan es imprescindible para el funcionamiento de la aplicación. Puede encontrarse en la forja del grupo *SPI&FM* [SPI]. La instalación de *Takuan* puede realizarse por separado, sin embargo la misma instalación de *IdiginBPEL* la incluye y recomendamos emplearla.

Guión de instalación

IdiginBPEL incluye un guión de instalación que puede encontrarse en la forja del proyecto [Jav10]. El guión permite instalar tanto *IdiginBPEL* como *Takuan* y acepta las siguientes opciones:

- **idg**: Instala o actualiza sólo *IdiginBPEL* .
- **takuan**: Instala o actualiza sólo *Takuan* .
- **both**: Instala o actualiza primero *Takuan* y posteriormente *IdiginBPEL* .

```
Installs (or update) IdiginBPEL and Takuan
use: install.py [idg|takuan|both]
```

El guión toma los siguientes pasos para la instalación de *IdiginBPEL* que es la que nos ocupa:

1. Descargar e instalar *Takuan* (el proceso es largo).
2. Descargar el repositorio de *IdiginBPEL*.
3. Instalar traducciones disponibles y ficheros comunes.
4. Crear estructura de carpetas locales en `~/.idiginbpel`.
5. Detectar la instalación de *Takuan* y realizar configuración.

home	Directorio *IdiginBPEL* del usuario.	`~/.idiginbpel`
share	Directorio *IdiginBPEL* de datos estáticos.	`/usr/share/idiginbpel`
takuan	Ruta a la instalación de *Takuan*.	`~/takuan`
bpelunit	Ruta a la instalación de *BPELUnit*.	`~/AeBpelEngine`
srv	Url de acceso al servidor *Tomcat* para la ejecución de casos de prueba.	`localhost`
port	Puerto de acceso al servidor *Tomcat* para la ejecución de casos de prueba.	7777
activebpel	Ruta al script de control del demonio *ActiveBPEL*.	`~/bin/ActiveBPEL.sh`

Tabla B.1: Campos de configuración disponibles y valores por defecto

B.3.2. Configuración

La configuración de *IdiginBPEL* es almacenada en un fichero de tipo *xml* que por defecto se encontrará en la carpeta por defecto del usuario, esto es `~/.idiginbpel/config.xml`. Pueden existir varios `config.xml`, algunos de ellos conteniendo valores por defecto que permitirán a la aplicación arrancar y funcionar en el caso de que no se encuentre el fichero de configuración en el directorio casa del usuario.
Estos ficheros de configuración serán evaluados por el siguiente orden:

1. Local de Usuario: `~/.idiginbpel/config.xml`
2. Local de Desarrollo: `.home/config.xml`
3. General del Sistema: `/usr/share/idiginbpel/config.xml`

Estos ficheros son buscados en orden, empleando el siguiente de la lista en caso de la no existencia del anterior.

El fichero de configuración

El fichero de configuración incluye valores necesarios para operar con *Takuan*, con el servidor *Tomcat* contra el que ejecutar las pruebas y las rutas a los ficheros propios de la aplicación.

Un ejemplo de fichero de configuración típico sería el siguiente:

Listado B.1: Ejemplo de fichero de configuración por defecto.

```
<!-- Default configuration file

The application looks for this file in the following order:
1. ~/.idiginbpel/config.xml
2. ./home/config.xml
3. /usr/share/idiginbpel/config.xml -->
<config>
    <!-- User directory -->
    <home src="~/.idiginbpel">"idg.help.home"</home>
    <!-- Runtime directory -->
    <share src="~/IdiginBPEL/share">"idg.help.share"</share>
    <!-- takuan instalation -->
    <takuan src="~/takuan">"idg.help.takuan"</takuan>
    <!-- BPELUnit instalation -->
    <bpelunit src="~/AeBpelEngine">"idg.help.bpelunit"</bpelunit>
    <!-- ActiveBpel Daemon -->
    <activebpel src="~/bin/ActiveBPEL.sh">
        "idg.help.activebpeldaemon"
    </bpelunit>
    <!-- Server url -->
    <svr value="localhost">"idg.help.svr"</svr>
    <!-- Server port -->
    <port value="7777">"idg.help.port"</port>
</config>
```

El fichero de configuración también especifica una cadena de traducción para la ayuda a mostrar en el editor gráfico de configuración de la aplicación. Como puede verse en el listado de código de la Figura B.1, cada valor puede ser o bien *src* o *value*, diferenciando entre valores sencillos (*value*), ya sean cadenas o numéricos, y rutas (*src*), que serán comprobadas.

Listado B.2: Ejemplo de valor individual en el fichero `config.xml`.

```
<port value="7777">"idg.help.port"</port>
```

Como puede verse en la línea del fichero de configuración situada en la Figura B.2, cada línea del fichero de configuración contiene su nombre, su valor y la referencia a la cadena de ayuda.

Editor de configuración gráfico

IdiginBPEL incluye un editor de configuración que permite ajustar las propiedades del fichero de configuración del usuario mediante una interfaz gráfica que evita la tarea de editar el fichero *xml* y que permite modificar ”al vuelo” durante una misma sesión las distintas variables usadas.

A este editor puede accederse mediante el menú `Fichero ->Configuración`. Cada valor incluye su nombre, valor, descripción y un indicador que muestra el estado de la variable. Si el texto introducido no es válido, bien porque sea una ruta que no existe o no valide, el indicador mostrará un símbolo rojo indicando error y la configuración no guardará ese valor para próximas ejecuciones del programa.

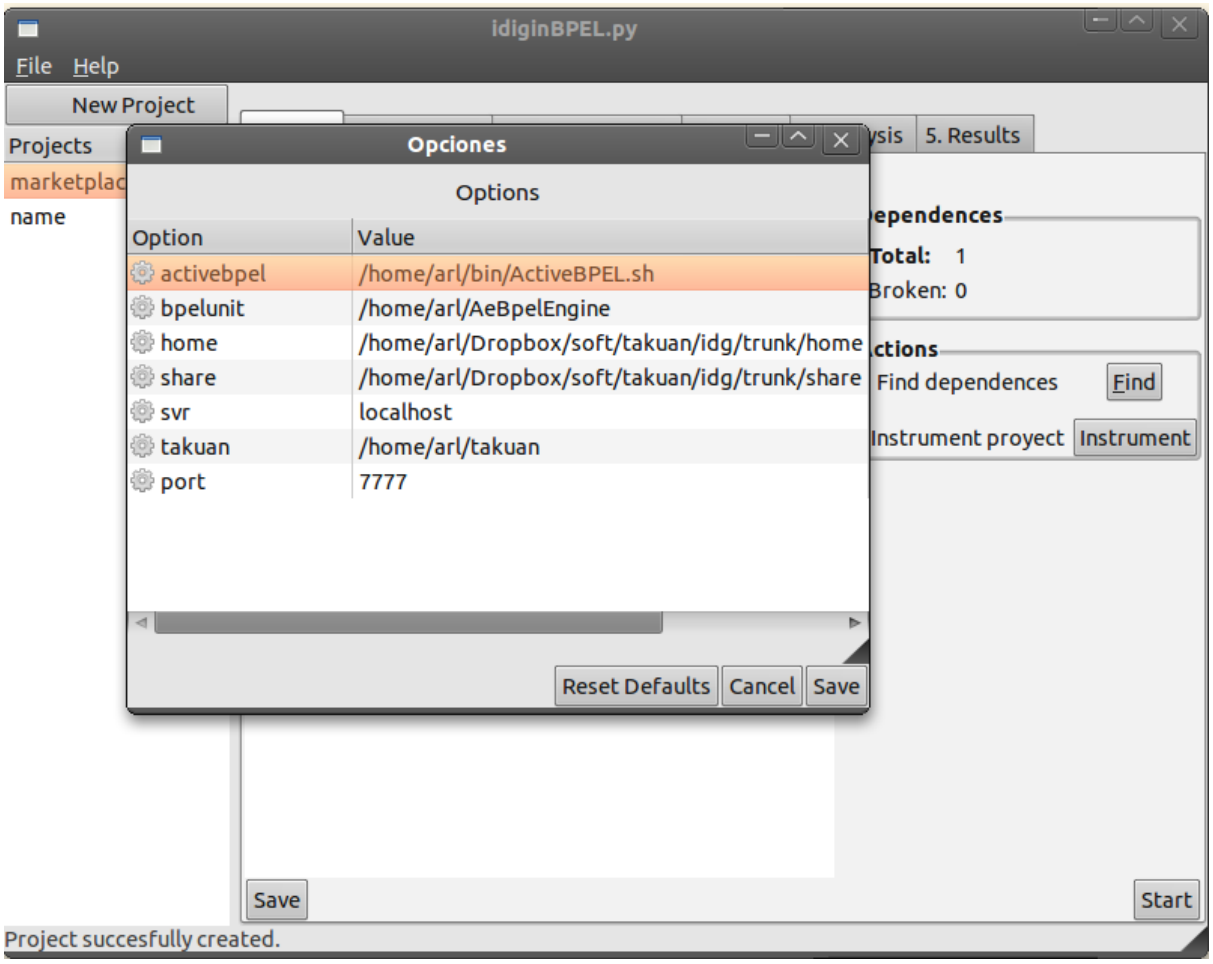


Figura B.11: Editor gráfico de configuración de *IdiginBPEL*

Configuración	Fichero con la configuración del proyecto.
Fichero *bpel* original y dependencias	El fichero de código *WS-BPEL 2.0* originalmente empleado para crear el proyecto así como todas sus dependencias.
Casos de prueba	Los ficheros de casos de prueba *BPELUnit* añadidos al proyecto.
Trazas de ejecución	Los ficheros de trazas resultantes de la ejecución de los casos de prueba *BPELUnit* en el servidor *ActiveBPEL.*
Invariantes resultantes	Los invariantes resultantes de los distintos análisis de trazas realizados.

Tabla B.2: Información contenida en un proyecto de *IdiginBPEL*

B.3.3. Multilenguaje

IdiginBPEL se encuentra disponible en idioma Español e Inglés. El idioma aplicado por defecto se toma desde las opciones del sistema operativo, y es en su defecto, el Inglés. Sin embargo, es posible tener el programa en otros idiomas realizando traducciones a las plantillas estándar adjuntas al sistema, el cual se encuentra preparado para admitir traducciones.

B.3.4. Formato de proyecto

La unidad mínima de trabajo en *IdiginBPEL* es el proyecto. Para *IdiginBPEL* un proyecto es un conjunto de ficheros y directorios que contienen el estado de trabajo sobre un fichero *WS-BPEL 2.0* junto a unos casos de prueba dados. Comprende la configuración, los ficheros originales junto a sus dependencias así como todos los productos intermedios que se generan durante el uso de la aplicación.

La información que un proyecto incluye se resume en la Tabla B.2.

Siendo un proyecto un conjunto de de ficheros y directorios no relacionados con la instancia de instalación concreta de *IdiginBPEL* es pues posible trasladar esos proyectos entre distintas instalaciones sin problema.

Exportación de proyectos

A la hora de trasladar un proyecto, guardarlo para mantener un estado anterior o compartirlo, es posible exportar este en un formato propio *.idg* de manera que pueda ser instalado y usado en cualquier máquina con la misma versión de *IdiginBPEL* instalada.

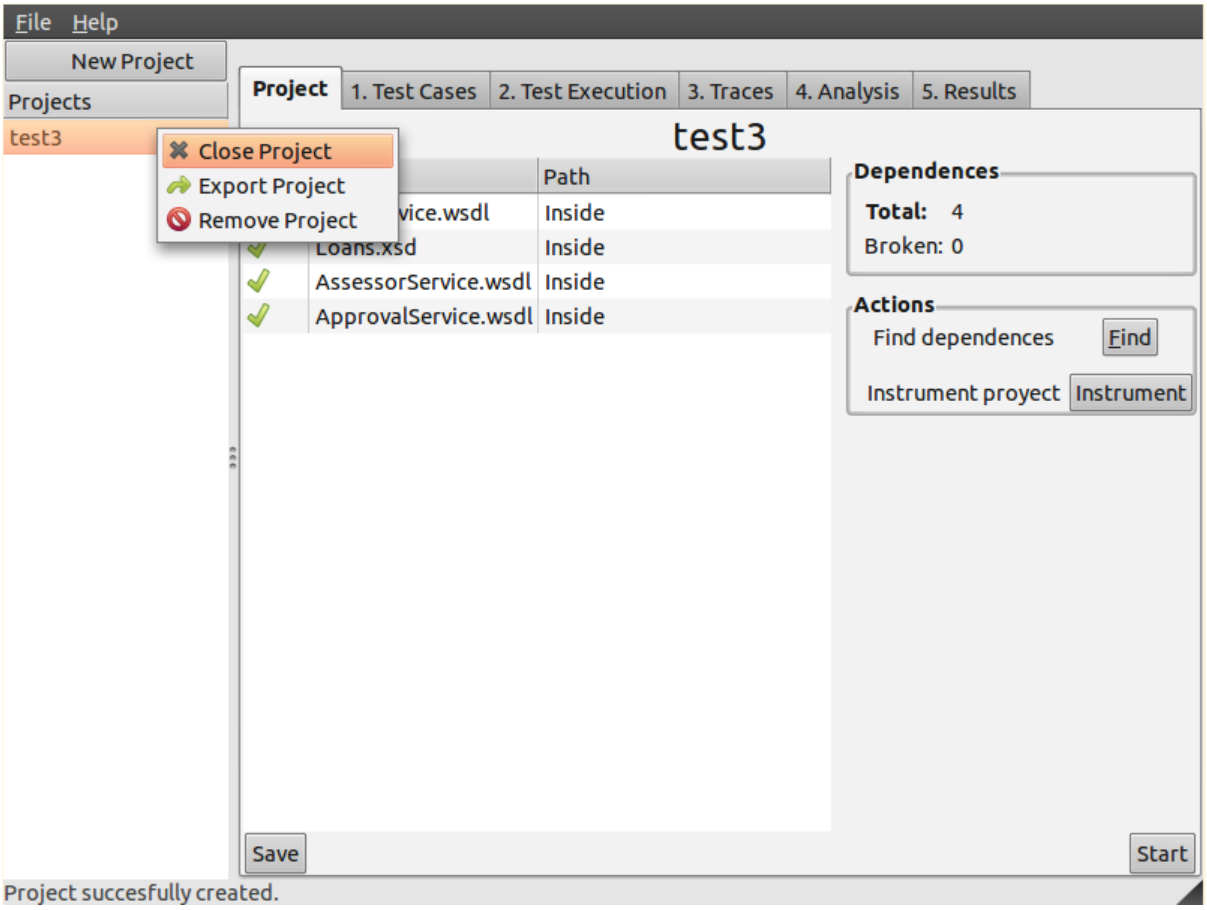


Figura B.12: Menú de exportación de un proyecto *IdiginBPEL*

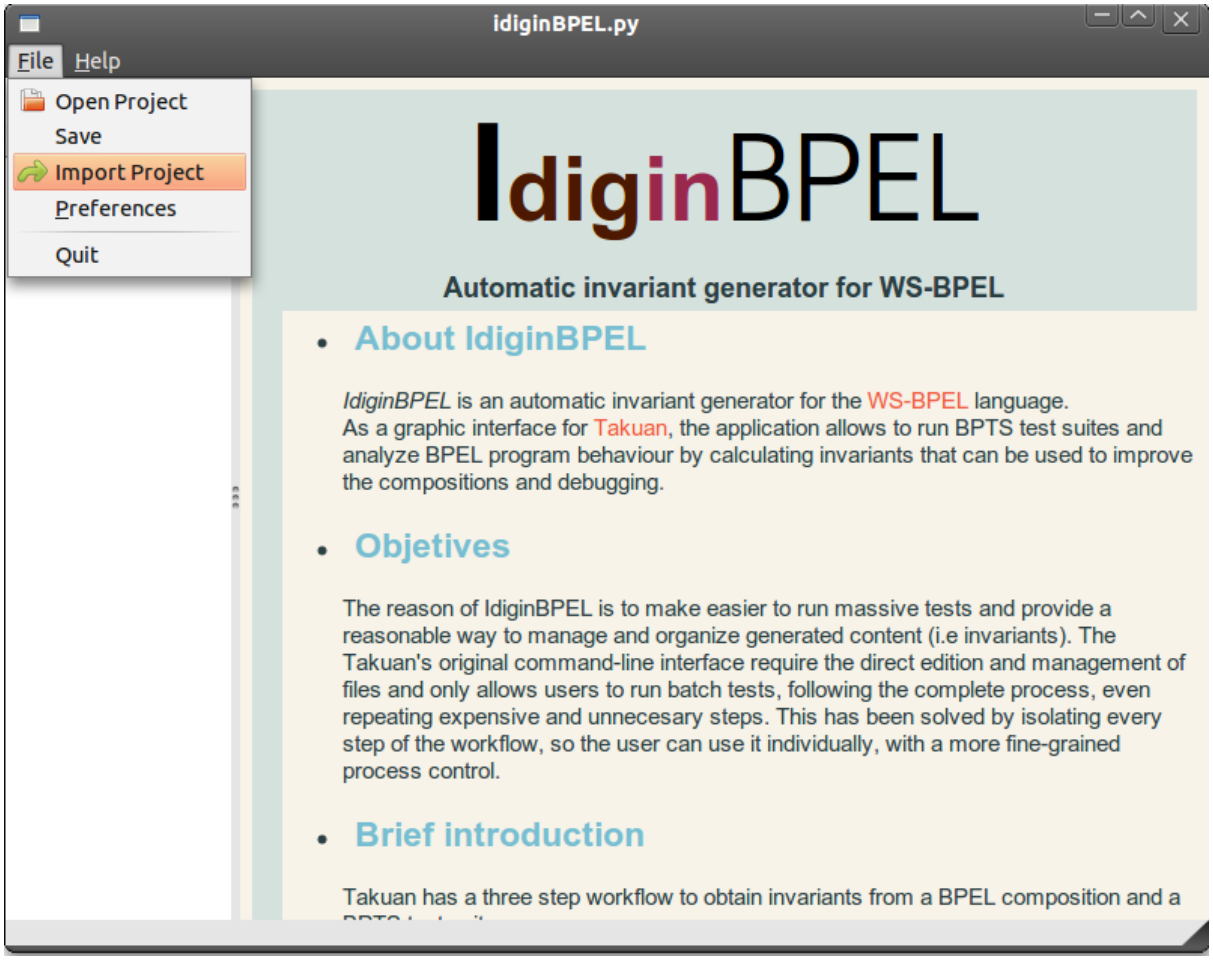


Figura B.13: Pantalla de importación de un proyecto *IdiginBPEL*

Para realizar la exportación de un proyecto de *IdiginBPEL* solo es necesario realizar ”click” derecho sobre el nombre del proyecto en la barra de la derecha, y seleccionar Exportar en el menú desplegable como puede verse en la Figura B.12.

También es posible emplear la opción disponible en `Fichero ->Exportar` mientras el proyecto se encuentra abierto. De cualquier manera, aparecerá una nueva ventana preguntando donde debe guardarse el proyecto exportado y podrá guardarlo.

Importación de proyectos

Una vez disponemos de un fichero de tipo *.idg* que contiene un proyecto exportado, es posible importarlo en nuestra instalación de *IdiginBPEL* mediante la opción disponible en `Fichero ->Importar`. Aparecerá una ventana emergente preguntando donde se encuentra el proyecto a importar y el sistema procederá a incluirlo con los proyectos de *IdiginBPEL* ya existentes como puede verse en la Figura B.13.

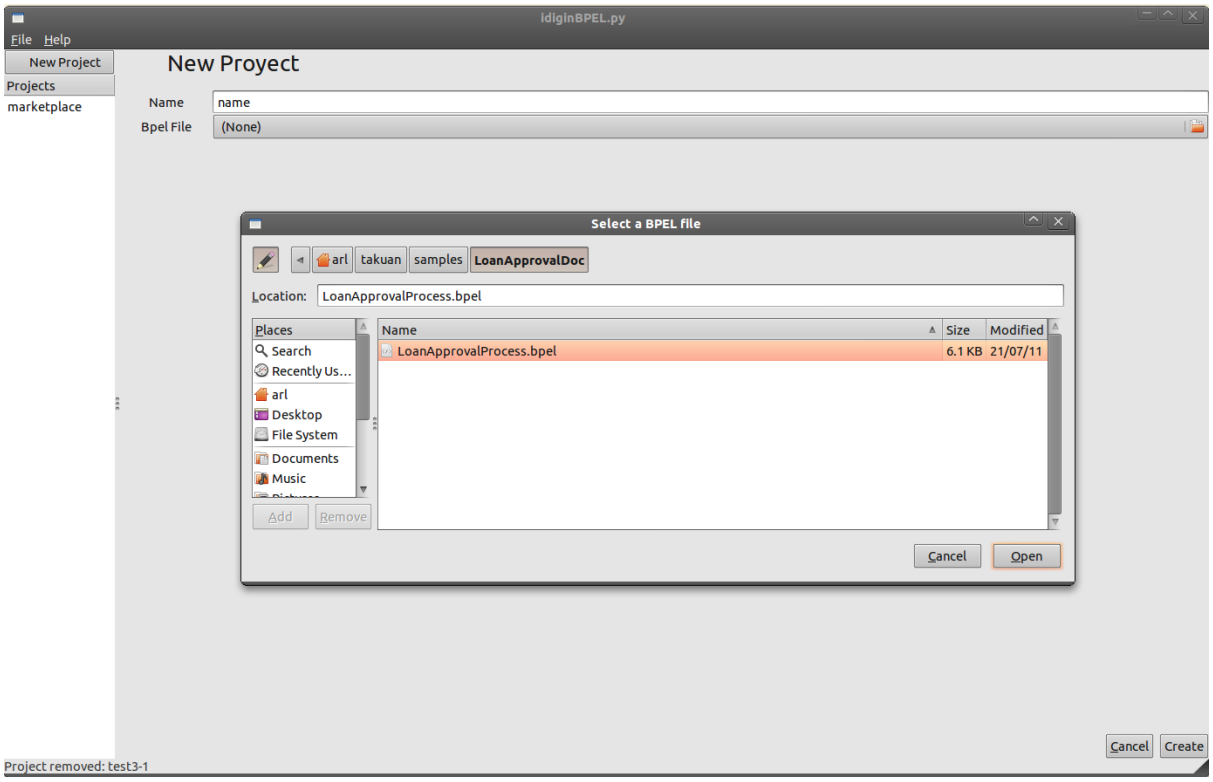


Figura B.14: Creación de un proyecto *IdiginBPEL* a partir de un fichero `.bpel` y un nombre dado

En el caso de colisión entre el nombre del proyecto a importar y el de uno ya existente en la instalación actual de *IdiginBPEL* se modificará el nombre del proyecto a importar empleando un sufijo.

Una vez importado un proyecto dentro de la instalación de *IdiginBPEL* es posible continuar su uso de forma idéntica a en la instalación donde se creó.

B.3.5. Creación de proyecto

A la hora de crear un proyecto, debemos proveerle simplemente de un nombre y un fichero de código fuente *WS-BPEL* `.bpel` como puede verse en la Figura B.14.

El nombre del proyecto tiene ciertas restricciones sobre el conjunto de caracteres que admite. Debido a que el nombre se utilizará como identificador único del proyecto en varios aspectos, este debe ser único, dentro de la instalación, y evitar caracteres no alfanuméricos.

El fichero de código fuente *WS-BPEL 2.0* `.bpel` estar preparado para su instrumentación inmediata. Esto es, debe mantener disponibles las dependencias de la composición *WS-BPEL 2.0* o de lo contrario no podrá ser importado completamente.

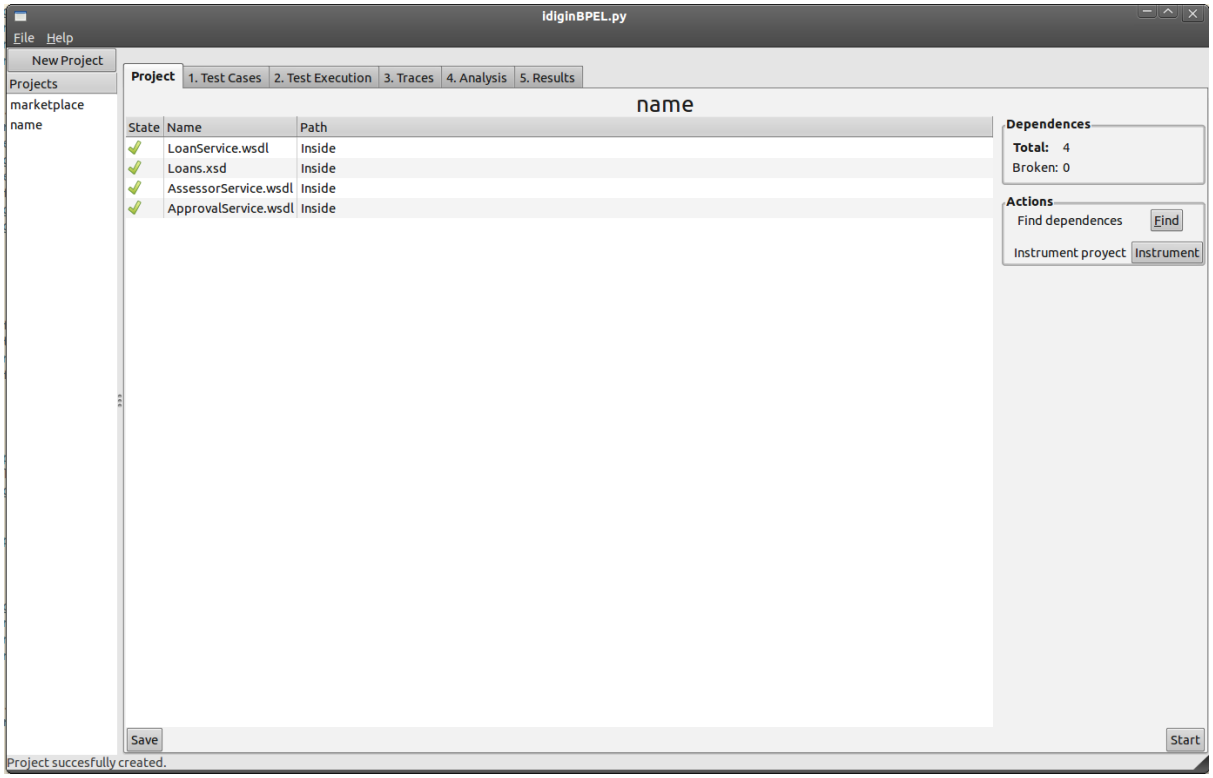


Figura B.15: Pantalla de informe tras la creación de un proyecto *IdiginBPEL*.

Una vez pulsado el botón de crear proyecto, el programa buscará el fichero de código fuente *WS-BPEL 2.0* `.bpel` y lo analizará, importando también sus dependencias. Tras importar todos los ficheros necesarios, se iniciará el proceso de *instrumentación*, que transforma el código fuente *WS-BPEL 2.0* originalmente seleccionado en otro distinto, preparado para el análisis al que será sometido posteriormente.

Como puede verse en la Figura B.15, que muestra un proyecto correctamente importado, en el cuadro central se listan las dependencias del importadas, junto a un icono indicando si la importación fue exitosa o no. De no encontrarse las dependencias, debe solucionar el problema y volver a realizar la búsqueda de dependencias, o de lo contrario no será posible instrumental el fichero de código fuente *WS-BPEL 2.0* `.bpel` ni por tanto realizar ninguno de los pasos posteriores de ejecución y análisis.

Puede comprobar que la instrumentación fue realizada correctamente en el mensaje de estado de la interfaz, en el margen inferior izquierda.

B.3.6. Casos de prueba

El código a ejecutar contra el servidor a bien de obtener las trazas está dispuesto en casos de prueba *BPELUnit*. Estos casos se encuentran organizados en «suites», de casos individuales, que son ejecutados uno tras otro, por orden.

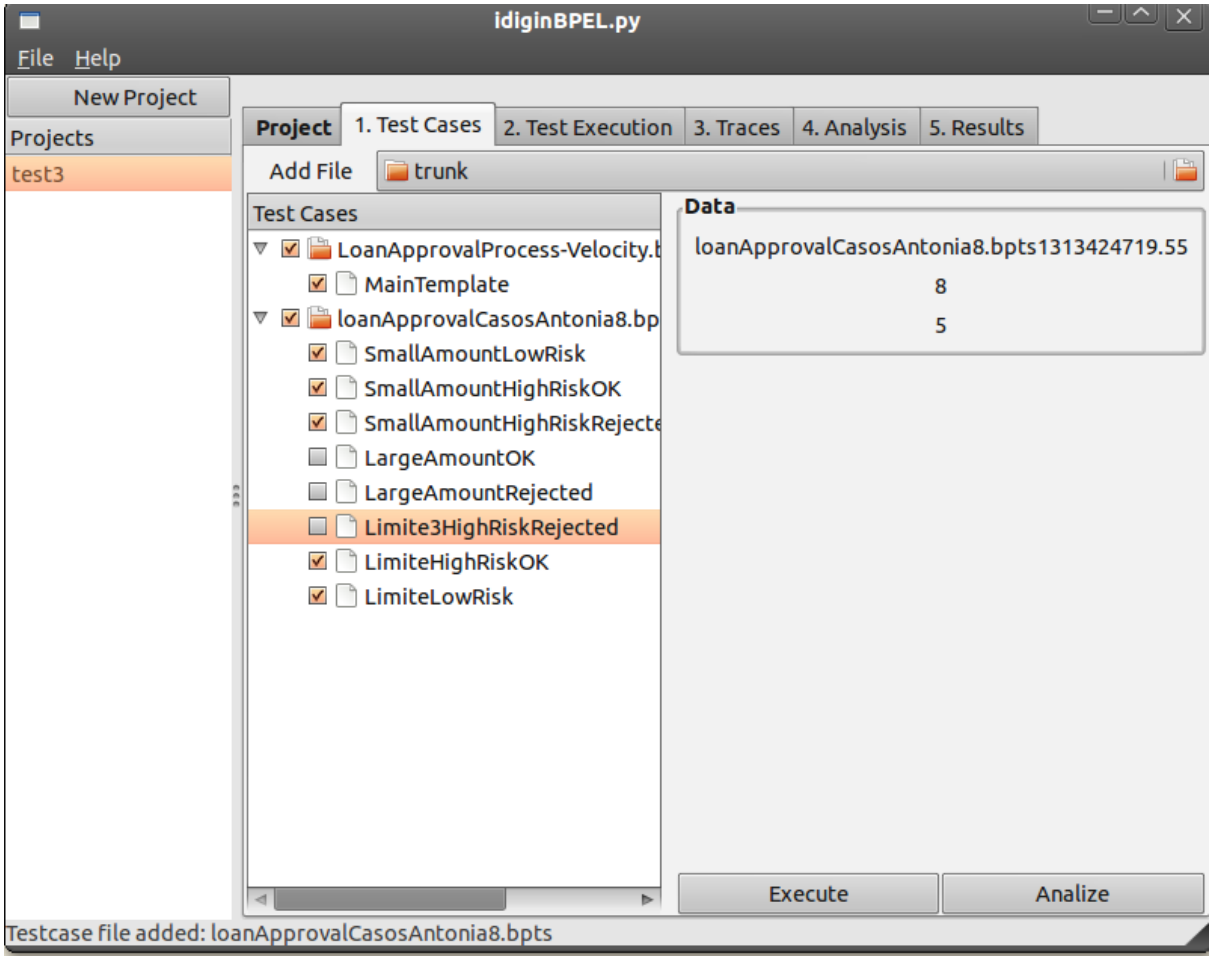


Figura B.16: Pantalla de selección de casos de prueba en *IdiginBPEL*

IdiginBPEL acepta la inclusión en el proyecto múltiples casos de prueba, los cuales pueden ser seleccionados para ejecutarse a disposición del usuario como puede verse en la Figura B.16

Los casos pueden añadirse mediante la adición de suites en ficheros de código fuente `.bpts`. Desde la misma pantalla de casos tan solo es necesario pulsar sobre el botón de *Añadir Casos* para incluir una nueva suite en los casos de prueba.

Los casos de prueba son listados lógicamente dentro de los ficheros de suite en la que se importaron a la aplicación, sin embargo es posible ejecutar cualquier combinación de casos disponible en el proyecto sin problema, más que seleccionándolos explícitamente en el selector de casos.

Para evitar en lo posible la colisión de nombres entre casos, el nombre de cada uno de ellos es precedido por el fichero de suite en el cual se encontraba definido al importarlo. En el caso de importar dos veces el mismo fichero, o dos ficheros con mismo nombre de fichero y de casos coincidente con otros ya existentes en el proyecto, se renombrarán empleando un sufijo.

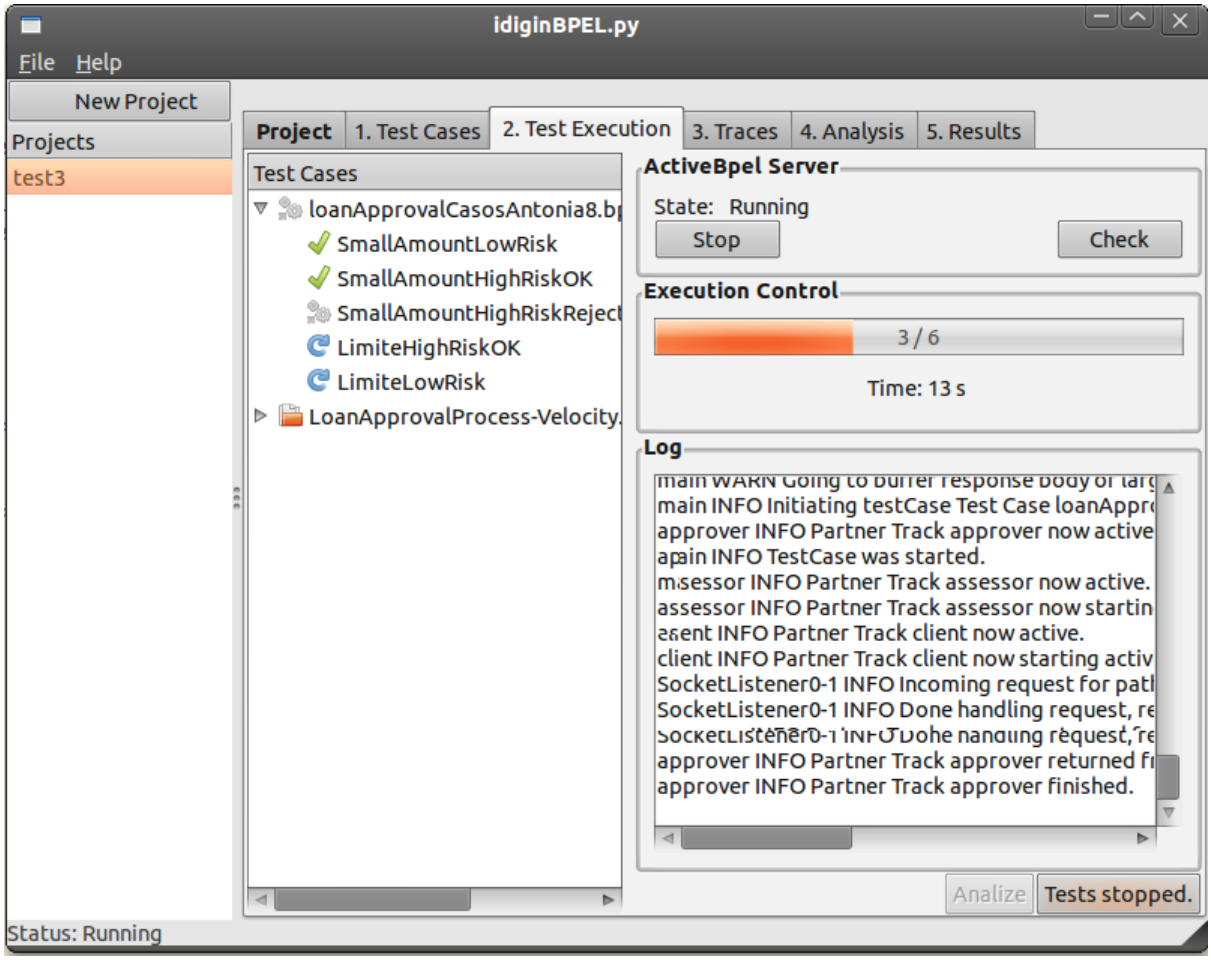


Figura B.17: Pantalla de control de ejecución de casos de prueba unitarios *BPELUnit* en *IdiginBPEL*

B.3.7. Ejecución de pruebas

Para ejecutar los casos de prueba seleccionados, es necesario que el servidor *Tomcat*con *ActiveBPEL*se encuentre activo y escuchando peticiones en el puerto especificado en la configuración de *IdiginBPEL*. Puede iniciarse, parar y comprobar el estado del servidor *Tomcat*desde la propia interfaz a través de los botones situados arriba a la derecha como puede verse en la Figura B.17.

Una vez el servidor está activo y dispuesto, es posible ejecutar la suite personalizada de casos de prueba. Los casos, que se encuentran listados en el árbol de la izquierda, se ejecutan por orden, uno tras otro. Inicialmente todos se encuentran en estado de espera, representado este por el icono de un pequeño reloj. Tras su ejecución, este icono de estado cambia a una marca verde o un aspa roja, según el resultado del test sea positivo o negativo. Puede verse una imagen de este código y de la ejecución en la Figura B.17.

En la parte inferior derecha de la pantalla de *IdiginBPEL* se muestra el «log» en plano que genera la ejecución de los casos de prueba, de manera que es posible visualizar que sucede en

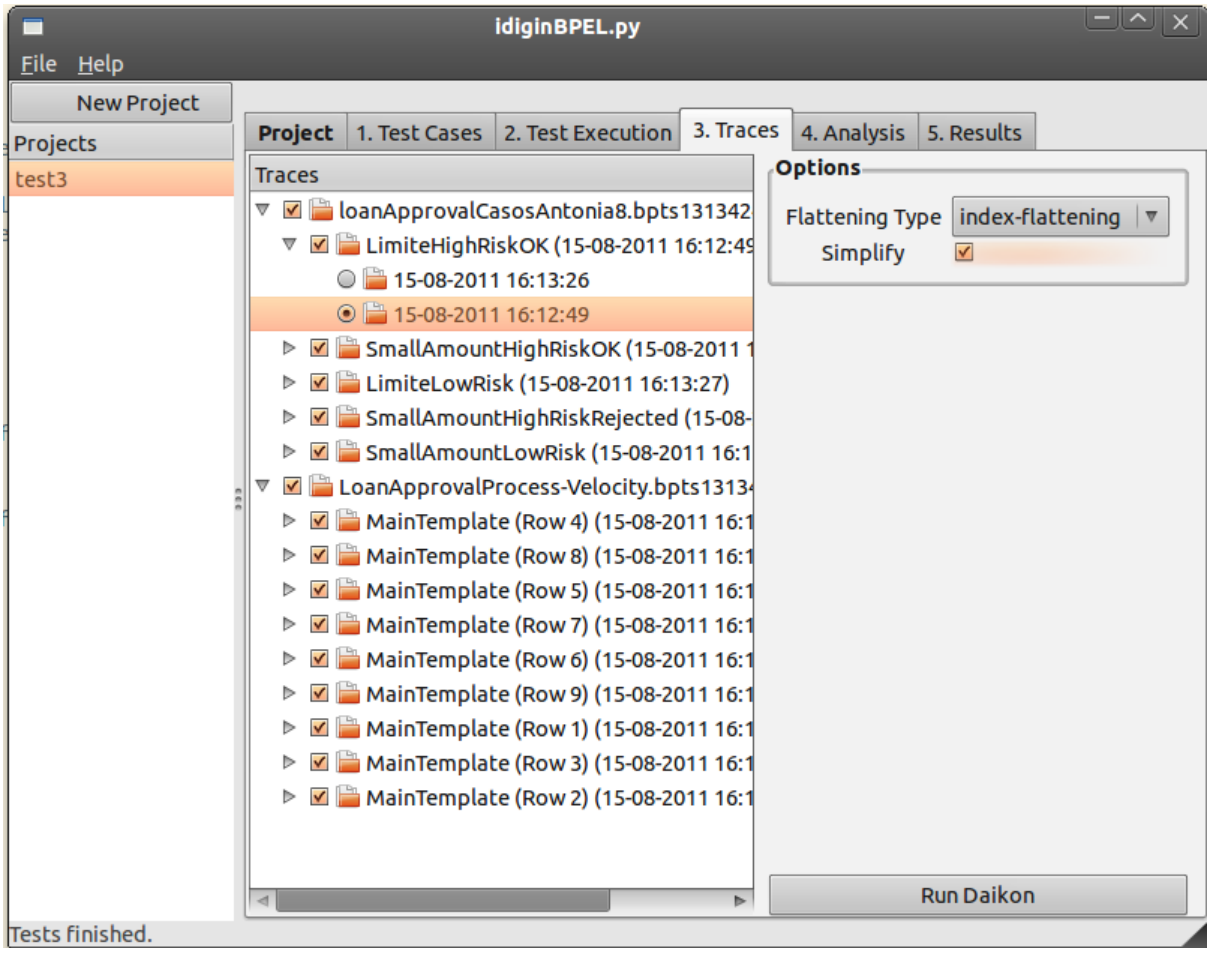


Figura B.18: Pantalla de trazas de *IdiginBPEL* tras una ejecución de casos de prueba

cada paso, en el supuesto de que se desee más detalle que el proporcionado por los estados y mensajes propios del programa, que a fin de cuentas no deja de ser una capa más sobre la herramienta *BPELUnit*.

B.3.8. Trazas generadas

Tras la ejecución de los casos de prueba, el servidor genera trazas de ejecución que son importadas automáticamente al finalizar por *IdiginBPEL* y se ponen a disposición del usuario en la siguiente pantalla de *Trazas* como puede verse en la Figura B.18.

Cada ejecución de un caso de prueba sin errores genera una traza, que es nombrada con respecto a su suite, caso y el momento de ejecución de la misma para diferenciarla en distintas ejecuciones.

En esta misma pantalla es posible elegir las trazas a tener en cuenta por el analizador estadístico *Daikon* pudiendo seleccionar una traza por caso.

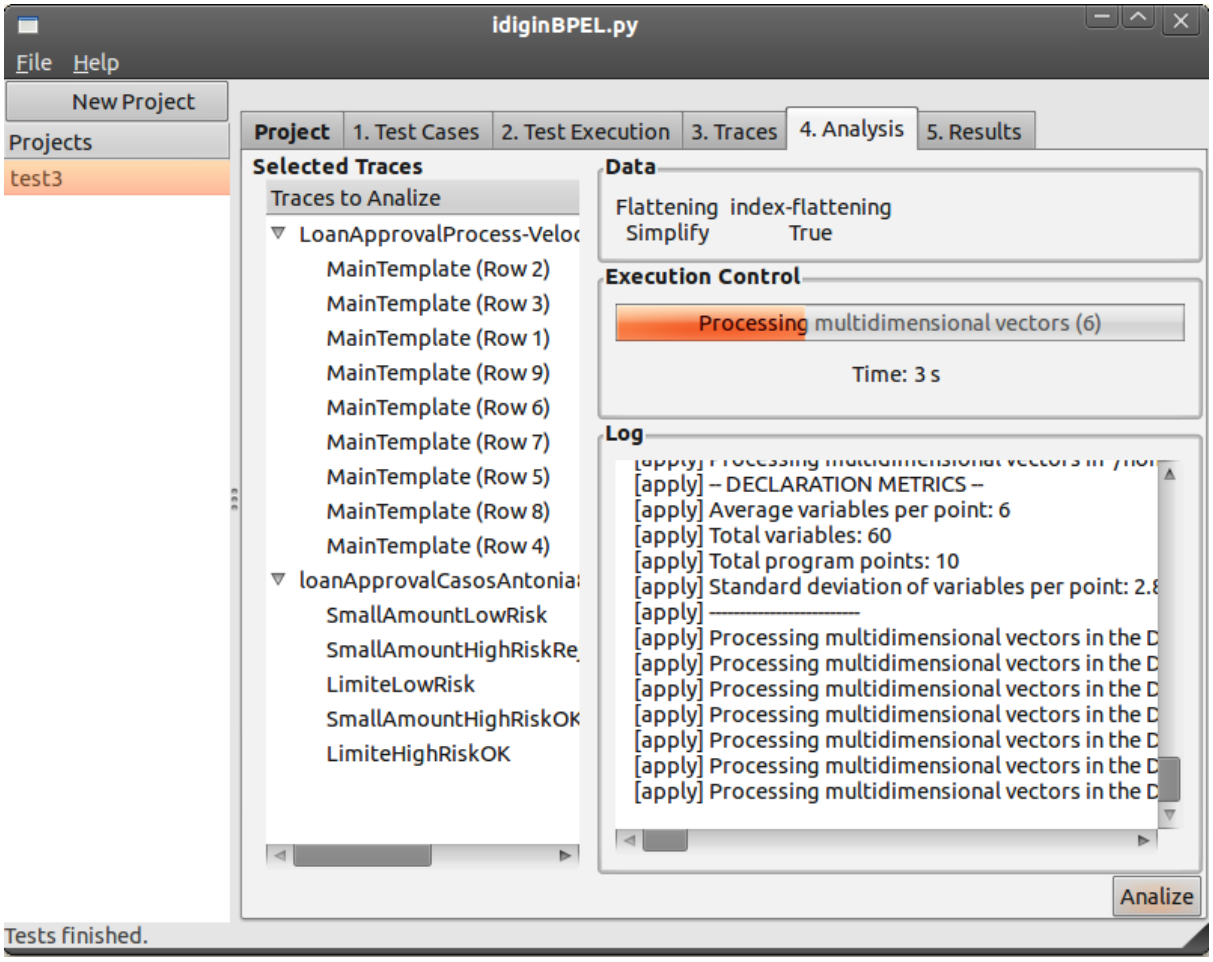


Figura B.19: Análisis estadístico de las trazas generadas por la ejecución mediante la herramienta *Daikon*

Es necesario asegurarse de que la aplicación se ejecuta con permisos suficientes como para acceder al directorio de «logs» de *ActiveBPEL* o de lo contrario no será capaz de importar las trazas y por tanto de continuar con el proceso hacia el análisis.

B.3.9. Análisis Estadístico

Una vez seleccionadas las trazas candidatas al análisis estadístico por *Daikon* estas son pasadas al analizador y procesadas. El progreso de ejecución y el «log» generado por *Daikon* durante el mismo se muestran en la pantalla de *Análisis* como puede verse en la Figura B.19.

El análisis de las trazas por *Daikon* puede demorar bastante tiempo, dependiendo directamente de la cantidad de propiedades a procesar. El proceso toma de tres pasos:

1. Importación de trazas.

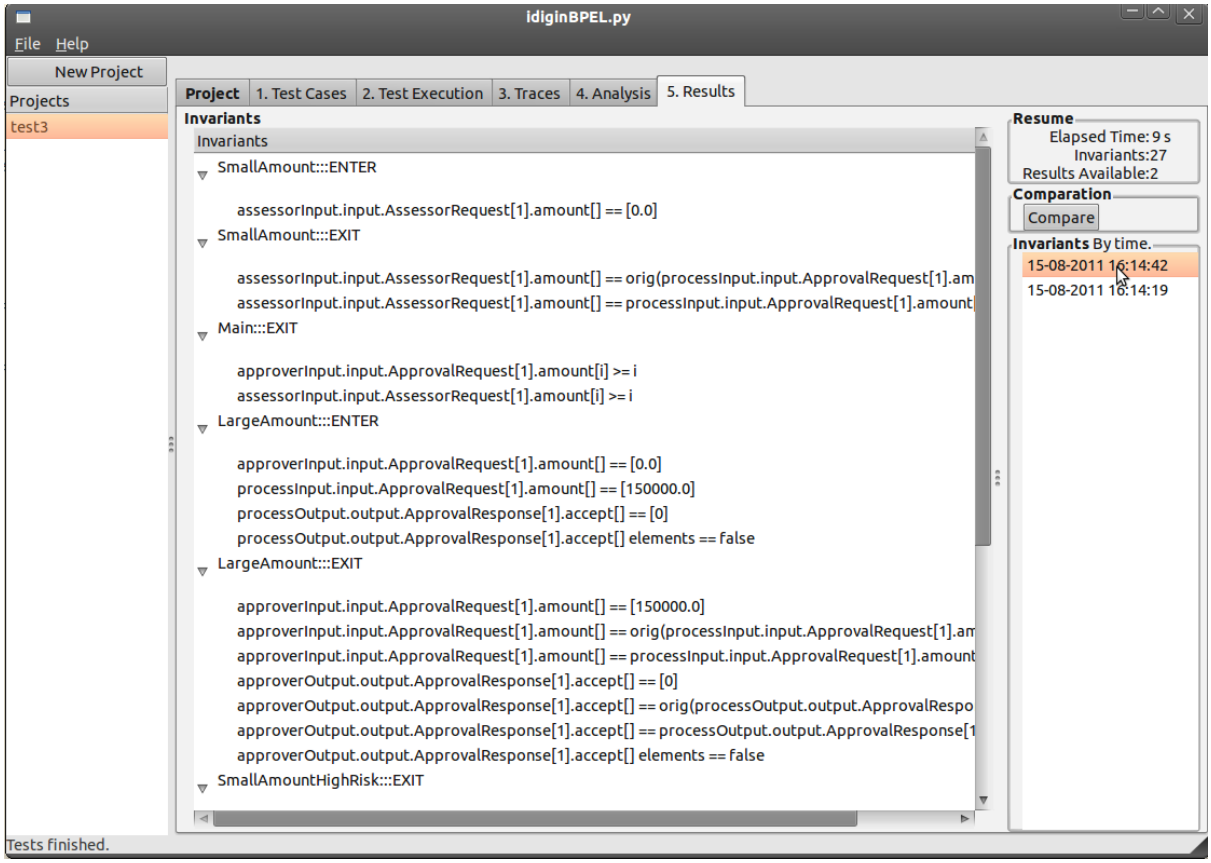


Figura B.20: Invariantes generados por *Daikon* al final del proceso

2. Transformación de trazas.
3. Análisis por *Daikon*.

Una vez terminado el análisis, el producto final resultante son los invariantes potenciales asociados a la composición *WS-BPEL* que pueden verse en la pantalla de *Invariantes*.

B.3.10. Invariantes generados

Tras un análisis estadístico por *Daikon* finalizado con éxito, una serie de invariantes son generados y mostrados en la pantalla. En el árbol de la derecha es posible ver todos los invariantes generados a lo largo de distintas ejecuciones como puede verse en la Figura B.20.

Estos invariantes pueden ser comparados entre si empleando la herramienta de comparación adjunta. Para ello deben seleccionarse dos invariantes generados de la lista y pulsar el botón *Comparar*.

Tras pulsar el botón *Comparar*, el resultado de la comparación será mostrado en pantalla como puede verse en la Figura B.21.

Figura B.21: Comparación de dos juegos de invariantes resultantes del análisis estadístico *Daikon*

En la comparación podemos observar como los invariantes que han cambiado se encuentran marcados en amarillo, los nuevos invariantes marcados en verde, y los invariantes que ya no aparecen en la siguiente versión, se encuentran marcados en rojo.

Es posible navegar entre los invariantes comparándolos entre si, de forma que es posible ver las diferencias entre el añadido de diferentes trazas.

APÉNDICE C

GNU Free Documentation License

Version 1.3, 3 November 2008

<http://fsf.org/>

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or

XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Bibliografía

[Á10] Alejandro Álvarez Ayllón. Reingeniería y ampliación del generador dinámico de invariantes potenciales para composiciones de servicios web en WS-BPEL Takuan. http://dx.doi.org/10498/8940, 2010. Proyecto fin de carrera de Ingeniería en Informática.

[Act08] Active Endpoints. Página principal del proyecto ActiveBPEL. http://www.activebpel.org, 2008.

[Apa11] Apache Team. Sitio Web de Apache Ant. http://ant.apache.org, 2011.

[Arl05] Jim Arlow y Ila Neustadt. *UML2 and the Unified Process*. Addison-Wesley, 2005.

[Aur11] Aurelien Pohu. Página oficial de BOUML. http://bouml.free.fr, 2011.

[Bjø97] Nikolaj Bjørner, Anca Browne, y Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, tomo 173(1):páginas 49–87, 1997.

[BPE06] BPELUnit team. Página principal del proyecto BPELUnit. http://www.bpelunit.net, 2006.

[BPE08] BPELUnit team. Forja en SourceForge del proyecto BPELUnit. http://www.bpelunit.sourceforge.net, 2008.

[Bus10] Business Process Management Initiative. BPMN FAQ. http://www.bpmn.org/Documents/FAQ.htm, 2010.

[Cha09] Scott Chacon. *Pro Git*. Apress, 2009.

[Con02] Joseph Conrad. *Heart of Darkness*. Penguin Books, 1902.

[Con11] Concurso Universitario de Software Libre. Página oficial del Concurso Universitario de Software Libre . http://www.concursosoftwarelibre.org/, 2011.

[Dai00] Daikon Team. Página principal del proyecto Daikon. http://groups.csail.mit.edu/pag/daikon/, 2000.

[Dij75] Ernst W. Dijkstra. Guarded commands, nondeterminancy and the formal derivation of programs. páginas 453–457, agosto 1975.

[Dij76] Ernst W. Dijkstra. *A Discipline for Programming*. Prentice Hall, 1976.

[Ern01] Michael D. Ernst, Jake Cockrell, William G. Griswold, y David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, tomo 27(2):páginas 99–123, 2001.

[Flo67] Robert Floyd. Assigning meanings to programs. 1967.

[Gam94] Erich Gamma, Richard Helm, Ralph Johnson, y John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[Gar08a] Antonio García-Domínguez, Manuel Palomo Duarte, y Inmaculada Medina-Bulo. Framework para la generación dinámica de invariantes en composiciones de servicios web con WS-BPEL. páginas 1–6, octubre 2008.

[Gar08b] Antonio García Domínguez, Manuel Palomo Duarte, y Inmaculada Medina Bulo. Implementación de un framework para la generación dinámica de invariantes en composiciones de servicios web con WS-BPEL. páginas 91–96, octubre 2008.

[Ger] Gerardo Aburruzaga García. Make. Un programa para controlar la recompilación. http://www.uca.es/softwarelibre/publicaciones/make.pdf.

[Gno11a] Gnome Foundation. Documentación oficial de las bibliotecas GTK+ 2.24. http://developer.gnome.org/gtk/2.24/, 2011.

[Gno11b] Gnome Foundation. Documentación oficial de los bindings PyGTK. http://www.pygtk.org/reference.html, 2011.

[Gno11c] Gnome Foundation. Listado de aplicaciones existentes desarrolladas con PyGTK. http://www.pygtk.org/, 2011.

[Gno11d] Gnome Foundation. Página oficial del proyecto GTK+. http://www.gtk.org/, 2011.

[Gno11e] Gnome Project. Documentación oficial de gtkBuilder. http://developer.gnome.org/gtk/stable/GtkBuilder.html, 2011.

[Gno11f] Gnome Project. Documentación oficial de libglade. http://developer.gnome.org/libglade/stable/, 2011.

[Gno11g] Gnome Project. Página principal del proyecto Glade. http://glade.gnome.org/, 2011.

[Gre11] Greg Smith. Filtro de adaptación a Python para Doxygen. http://code.foosel.org/doxypy#download, 2011.

[Gui01] Guido van Rossum. Style Guide for Python Code. http://www.python.org/dev/peps/pep-0008/, 2001.

[H.04] Swaroop C H. *A Byte of Vim*. 2004.

[Hoa69] C.A.R Hoare. An axiomatic basis for computer programming. páginas 576–580, octubre 1969.

[IBM03] IBM. Business Process Execution Language for Web Services version 1.1. http://www.ibm.com/developerworks/library/specification/ws-bpel/, 2003.

[Jav09a] Javier Santacruz. Blog del proyecto IdiginBPEL. http://idiginbpel.wordpress.com/, 2009.

[Jav09b] Javier Santacruz. Forja de IdiginBPEL alojada en RedIris (svn). https://forja.rediris.es/projects/cusl4-idigin/, 2009.

[Jav10] Javier Santacruz. Forja de IdiginBPEL. https://gitorious.org/idiginbpel, 2010.

[Jim09] Juan José Domínguez Jiménez, Antonia Estero Botaro, Antonio García Domínguez, y Inmaculada Medina Bulo. GAmera: an automatic mutant generation system for WS-BPEL compositions. páginas 97–106, noviembre 2009.

[JUn00] JUnit Team. Unit test framework for Java. http://www.junit.org, 2000.

[Lut09] Mark Lutz. *Programming Python*. O'Reilly, 2009.

[Man08] Manuel Palomo Duarte. Página oficial del proyecto Takuan. https://neptuno.uca.es/redmine/projects/takuan-website, 2008.

[Mee08] Heather J. Meeker. *The open source alternative*. John Wiley and Sons, 2008.

[Mor08] Shoichi Morimoto. A survey of formal verification for business process modeling. *ICCS '08: Proceedings of the 8th international conference on Computational Science, Part II*, páginas 514–522. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-69386-4. doi:http://dx.doi.org/10.1007/978-3-540-69387-1_58.

[Mye04] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, 2 edición, 2004. ISBN 0471469122.

[Nie99] Jakob Nielsen. *Designing Web Usability*. Peachpit Press, 1999.

[Nok11a] Nokia. Documentación oficial de las bibliotecas Qt. http://doc.qt.nokia.com/, 2011.

[Nok11b] Nokia. Página oficial de las bibliotecas Qt. http://qt.nokia.com/products/, 2011.

[OAS07] OASIS. Web Services Business Process Execution Language 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, 2007.

[Ora11] Oracle. Sitio web oficial de Oracle Netbeans. http://www.oracle.com/technetwork/java/javase/downloads/jdk-netbeans-jsp-142931.html, 2011.

[Pal08a] Manuel Palomo Duarte, Antonio García Domínguez, y Inmaculada Medina-Bulo. An architecture for dynamic invariant generation in WS-BPEL web service compositions. páginas 37–44, julio 2008.

[Pal08b] Manuel Palomo Duarte, Antonio García Domínguez, y Inmaculada Medina Bulo. Takuan: A dynamic invariant generation system for WS-BPEL compositions. páginas 63–72, 2008. doi:http://dx.doi.org/10.1109/ECOWS.2008.17.

[Pal09] Manuel Palomo Duarte, Antonio García Domínguez, y Inmaculada Medina-Bulo. Enhancing ws-bpel dynamic invariant generation using xml schema and xpath information. páginas 469–472, 2009.

[Pal10] Manuel Palomo Duarte, Antonio García Domínguez, Inmaculada Medina Bulo, Alejandro Álvarez Ayllón, y Javier Santacruz. Takuan: A tool for ws-bpel composition testing using dynamic invariant generation. páginas 531–534, 2010.

[Pal11] Manuel Palomo Duarte. *Generación dinámica de invariantes para composiciones de servicios web en WS-BPEL.* Tesis Doctoral, Universidad de Cádiz, 2011.

[Pyl06] Pylint project. Página oficial de Pylint. http://www.logilab.org/857, 2006.

[Pyt11a] Python Software Foundation. Listado de aplicaciones existentes desarrolladas con pyQt. http://www.diotavelli.net/PyQtWiki/SomeExistingApplications, 2011.

[Pyt11b] Python Software Foundation. Pylint en la página oficial de Python. http://pypi.python.org/pypi/pylint, 2011.

[Pyt11c] Python Software Foundation. Python 2.7 Documentation. http://docs.python.org/license.html, 2011.

[Pyt11d] Python Software Foundation. Página wiki de la biblioteca pyQt. http://wiki.python.org/moin/PyQt, 2011.

[Pyt11e] Python Software Foundation. Página wiki de la biblioteca Tkinter. http://wiki.python.org/moin/TkInter, 2011.

[RC07] Enric Rodríguez-Carbonell y Deepak Kapur. Generating all polynomial invariants in simple loops. *J. Symb. Comput.*, tomo 42(4):páginas 443–476, 2007.

[Riv11] Riverbank Computing Limited. Documentación oficial de pyQt. http://www.riverbankcomputing.com/static/Docs/PyQt4/html/, 2011.

[Som02] I Sommerville. *Ingeniería del Software*. Addison Weasley, 2002.

[SPI] SPI&FM Group. Gestor de proyectos del SPI&FM Group. http://www.neptuno.uca.es.

[SPI11] SPI&FM Group. Repositorio de código compartido. https://neptuno.uca.es/redmine/projects/sources-fm/repository, 2011.

[Van] Van Heesch, Dimitri. Pagina oficial de Doxygen. http://www.doxygen.org.

[vR98] Guido van Rossum. Glue it all together with python. enero 1998.

[Wika] Wikipedia. BPEL Wikipedia. http://en.wikipedia.org/wiki/Business_Process_Execution_Language.

[Wikb] Wikipedia. Definición de Invariante en Wikipedia. http://en.wikipedia.org/wiki/Invariant_(computer_science).

[Wikc] Wikipedia. Dynamic Testing Wikipedia. http://en.wikipedia.org/wiki/Dynamic_testing.

[Wikd] Wikipedia. OASIS Wikipedia. http://en.wikipedia.org/wiki/OASIS_(organization).

[Wik11a] Wikipedia. Control de Versiones. http://es.wikipedia.org/wiki/Control_de_versiones, 2011.

[Wik11b] Wikipedia. Definición de Modelo de Ciclo de Vida iterativo en Wikipedia. http://es.wikipedia.org/wiki/Desarrollo_iterativo_y_creciente, 2011.

[Wik11c] Wikipedia. Monty Python. http://en.wikipedia.org/wiki/Monty_Python, 2011.

[wxW11] wxWidgets. Página principal del proyecto wxPython. http://www.wxpython.org/, 2011.