Generación Dirigida por Modelos de Pruebas de Rendimiento desde UML y MARTE

Antonio García Domínguez

UCASE
Universidad de Cádiz, España

19 noviembre 2012

Contenidos

Motivación

Rendimiento como requisito

- En algunos contextos, puede ser clave para tener éxito
- Se firman Acuerdos de Nivel de Servicio en partes críticas
- Es difícil garantizar el cumplimiento de un acuerdo en una composición de Servicios Web

Principal reto: dependencia en otros servicios

- ¿Cuánto rendimiento debemos pedir?
- Insuficiente: no cumpliremos nuestro acuerdo
- Excesivo: pagaremos más de la cuenta

Enfoques existentes para obtener el rendimiento deseado

¿Desde acuerdos o desde código? ¿Ascendente o descendente?

Ingeniería de rendimiento tradicional: acuerdos, ascendente

- Tenemos acuerdos para todos los componentes
- Estimamos el rendimiento global y comparamos
- ¿Y si no tenemos esa información para algún servicio?

Perfilado o monitorización: código, ascendente

- Hemos implementado todos los servicios
- Medimos tiempos reales y corregimos cuellos de botella
- ¿Y si tenemos que firmar un acuerdo antes de implementar?

Nuestro enfoque: inferencia de rendimiento

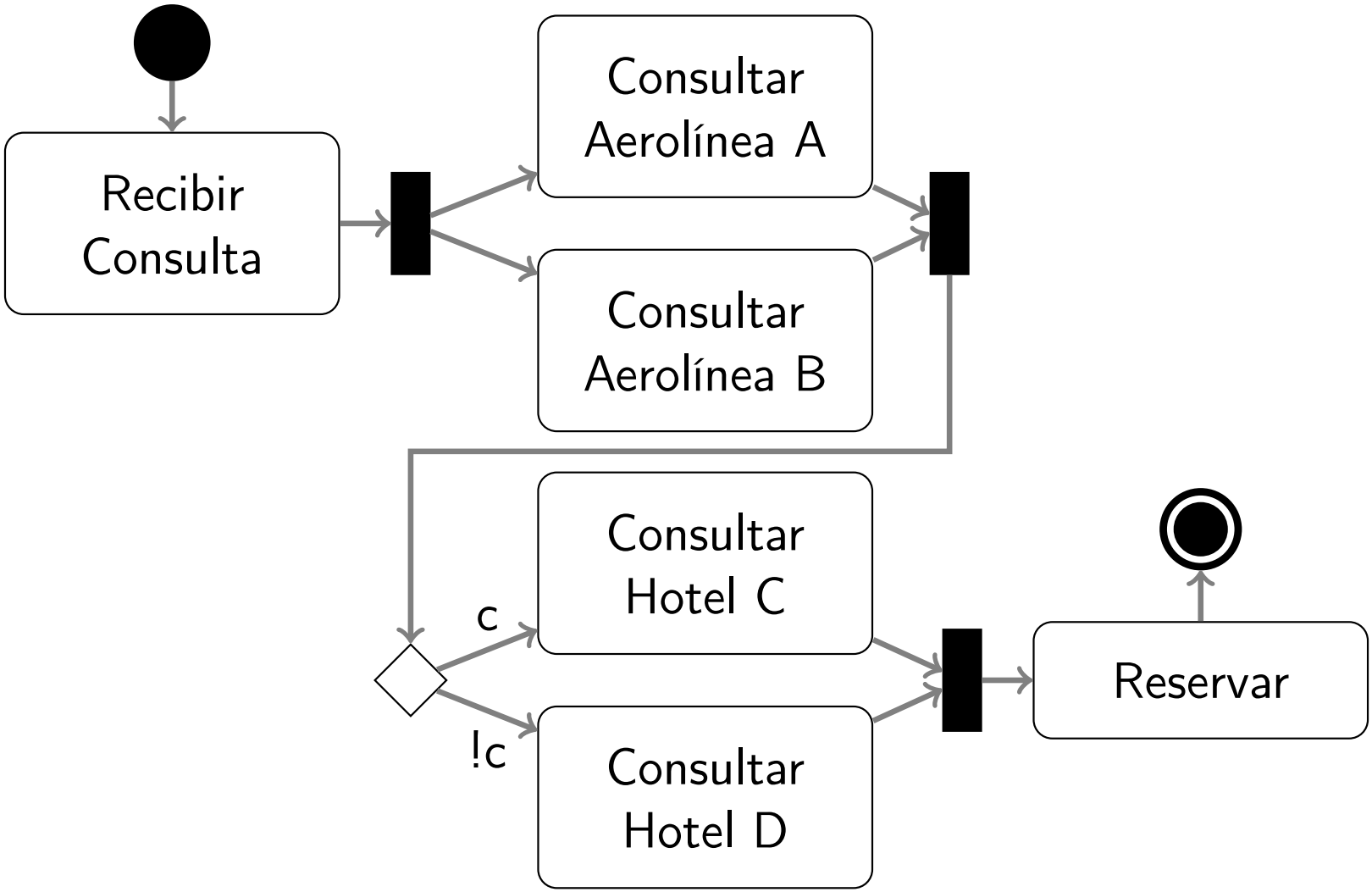
Características

- Desde acuerdos, descendente
- Tenemos requisitos para la composición y anotaciones locales con nuestros conocimientos parciales sobre los servicios usados
- Inferimos el rendimiento mínimo a exigir a los servicios

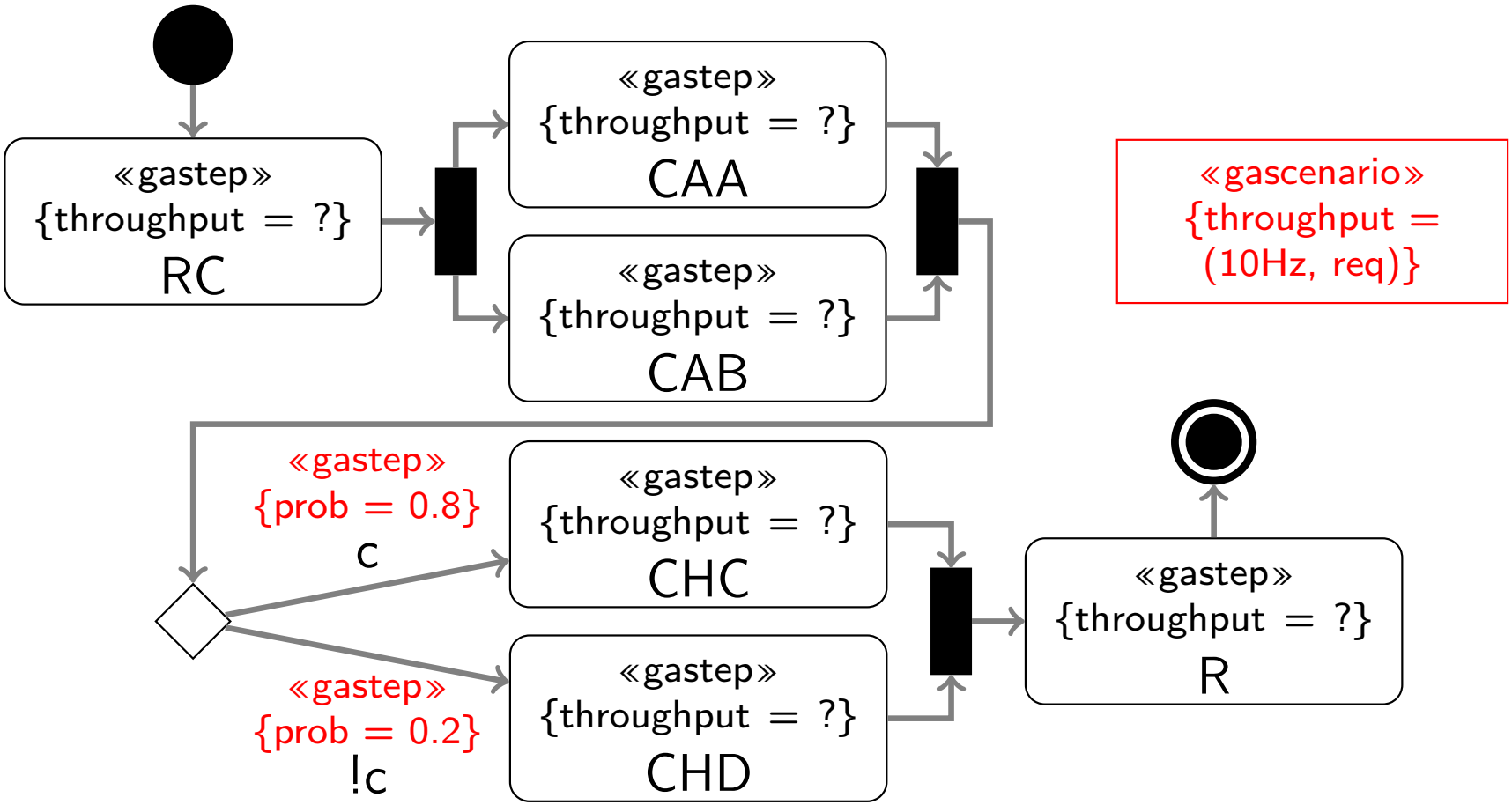
Ventajas

- Puede usar la información incompleta que haya disponible sobre los servicios
- Puede ayudar a definir el acuerdo antes de implementar

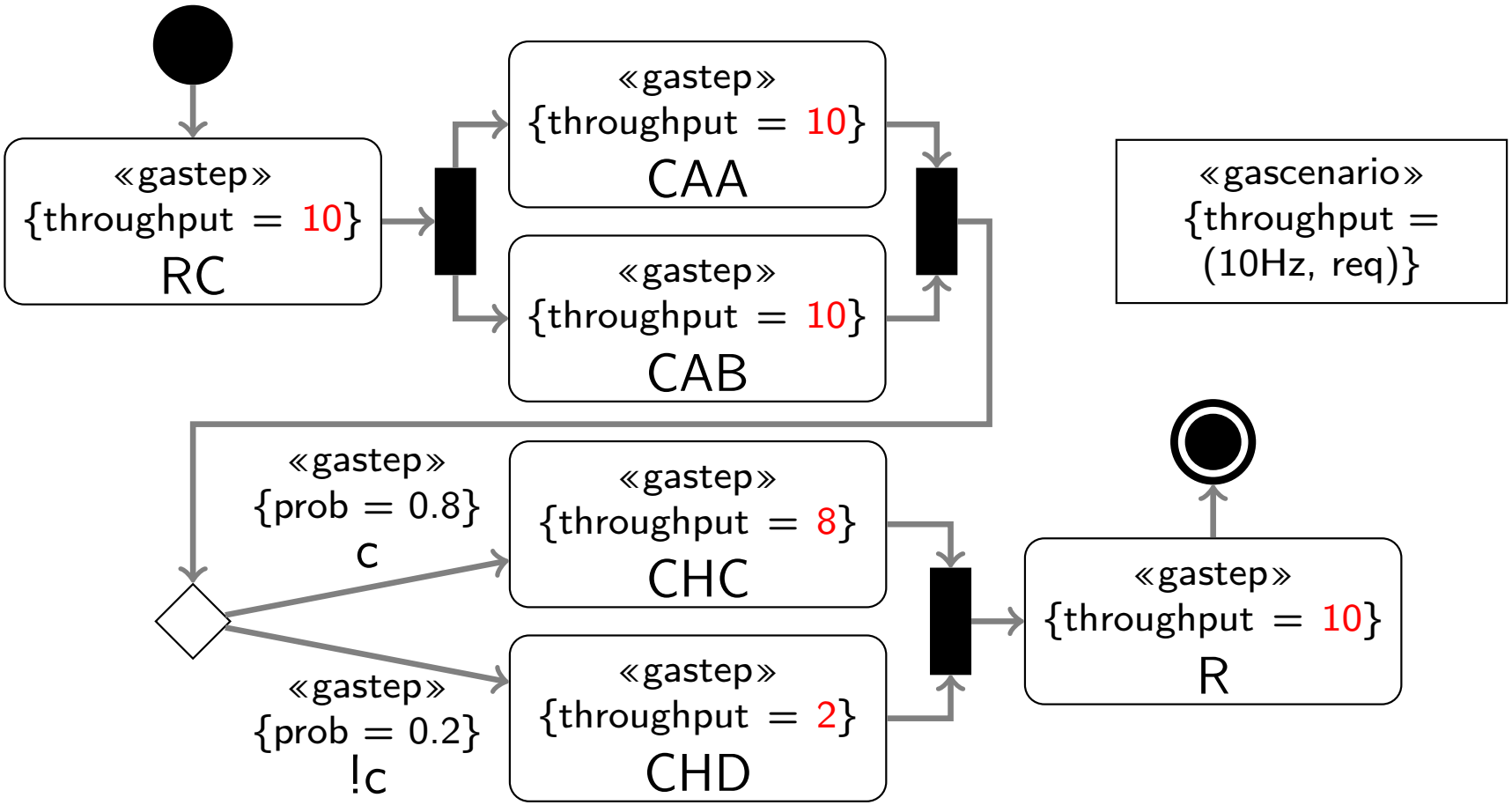
Ejemplo de modelo: motor de búsqueda de viajes



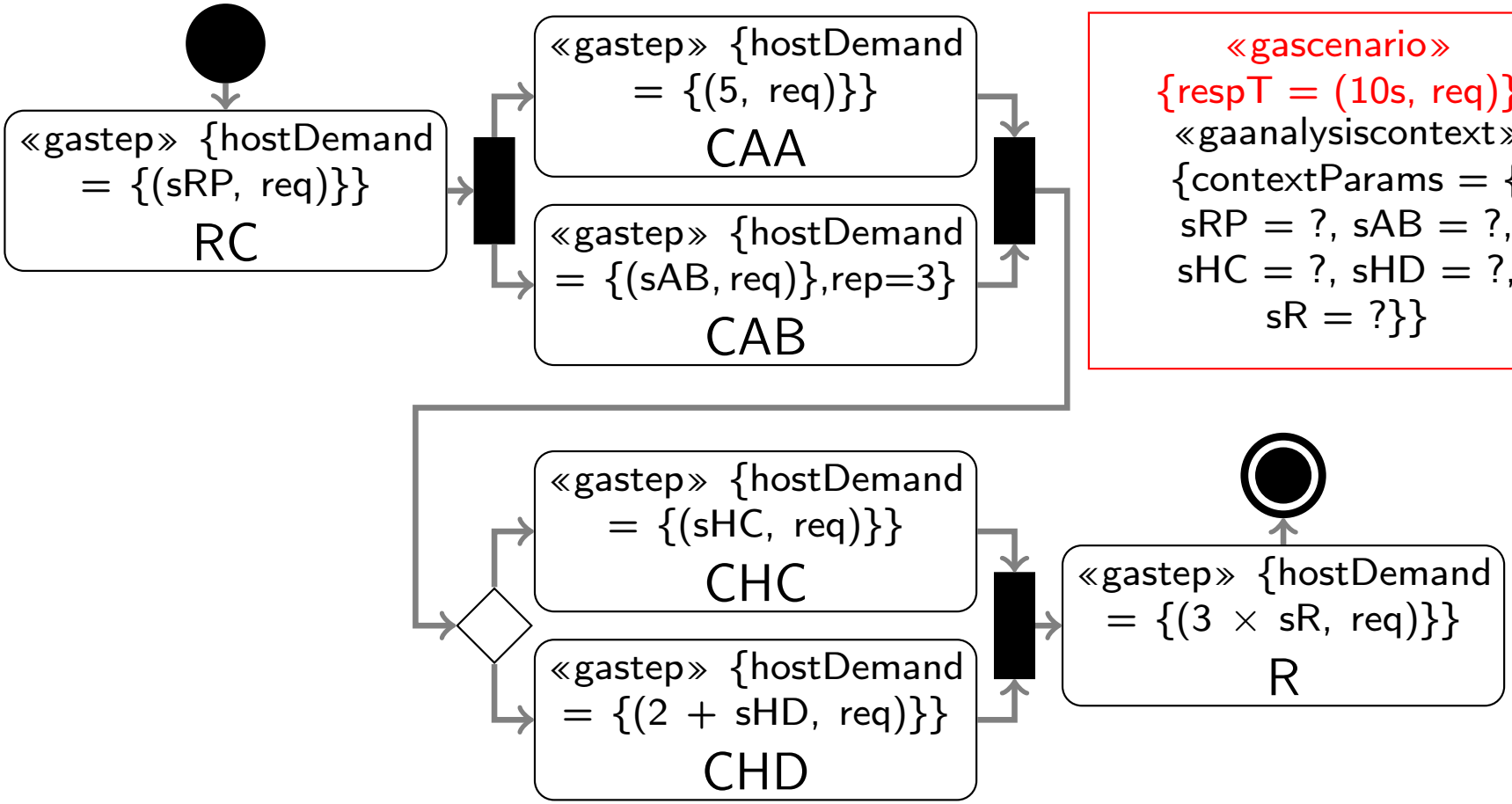
Inferencia de peticiones por segundo



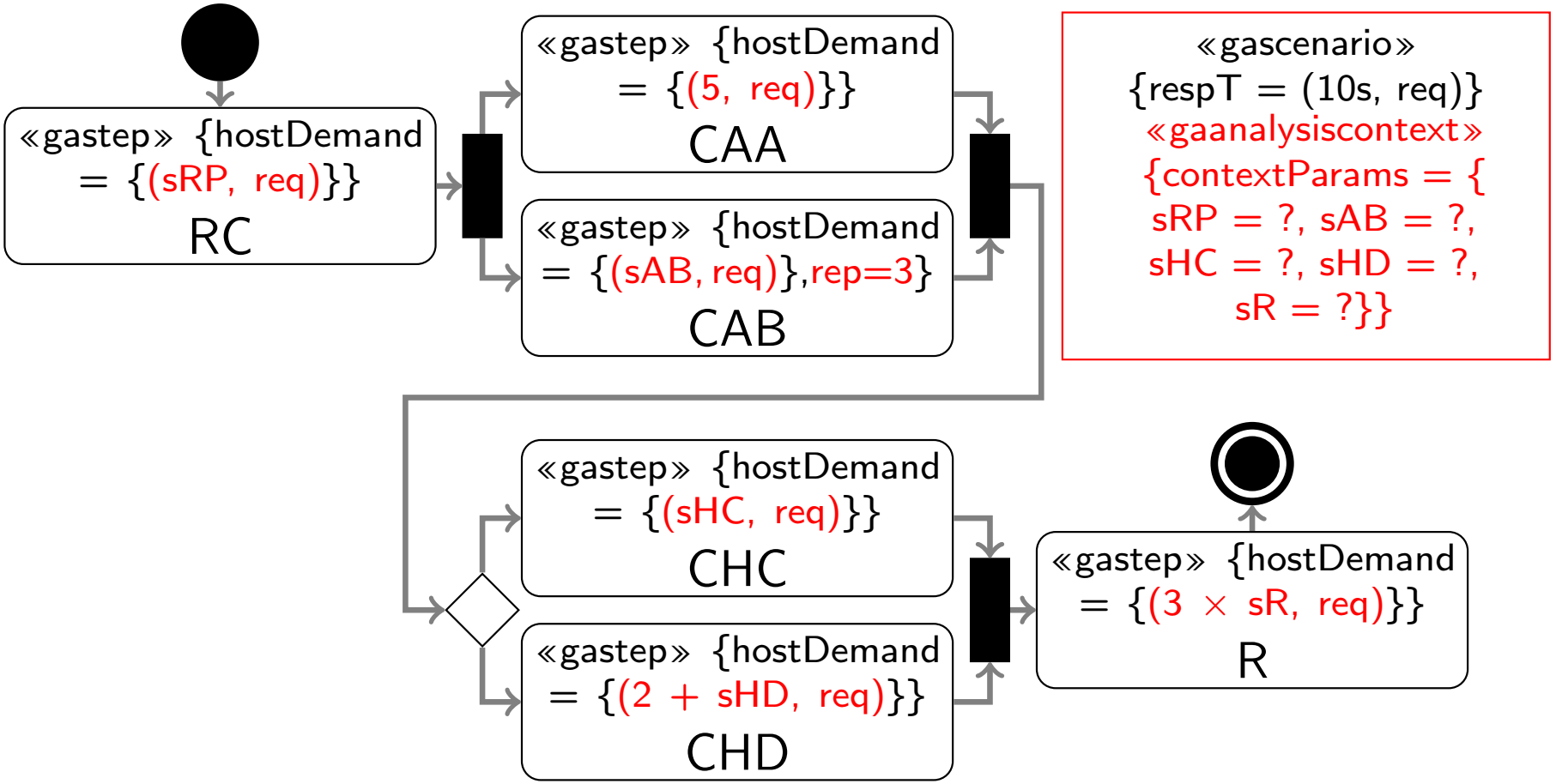
Inferencia de peticiones por segundo



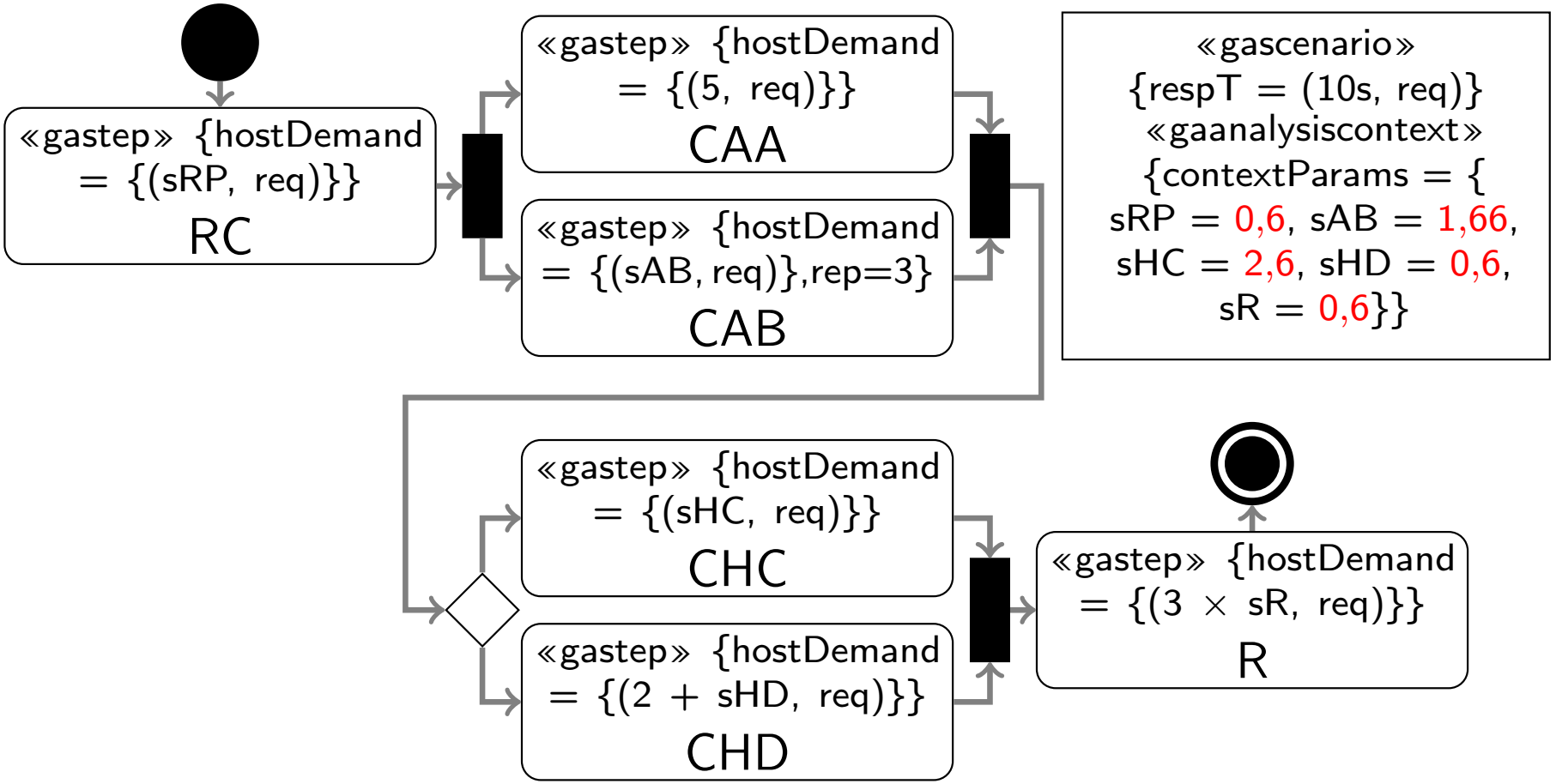
Inferencia de tiempos límite



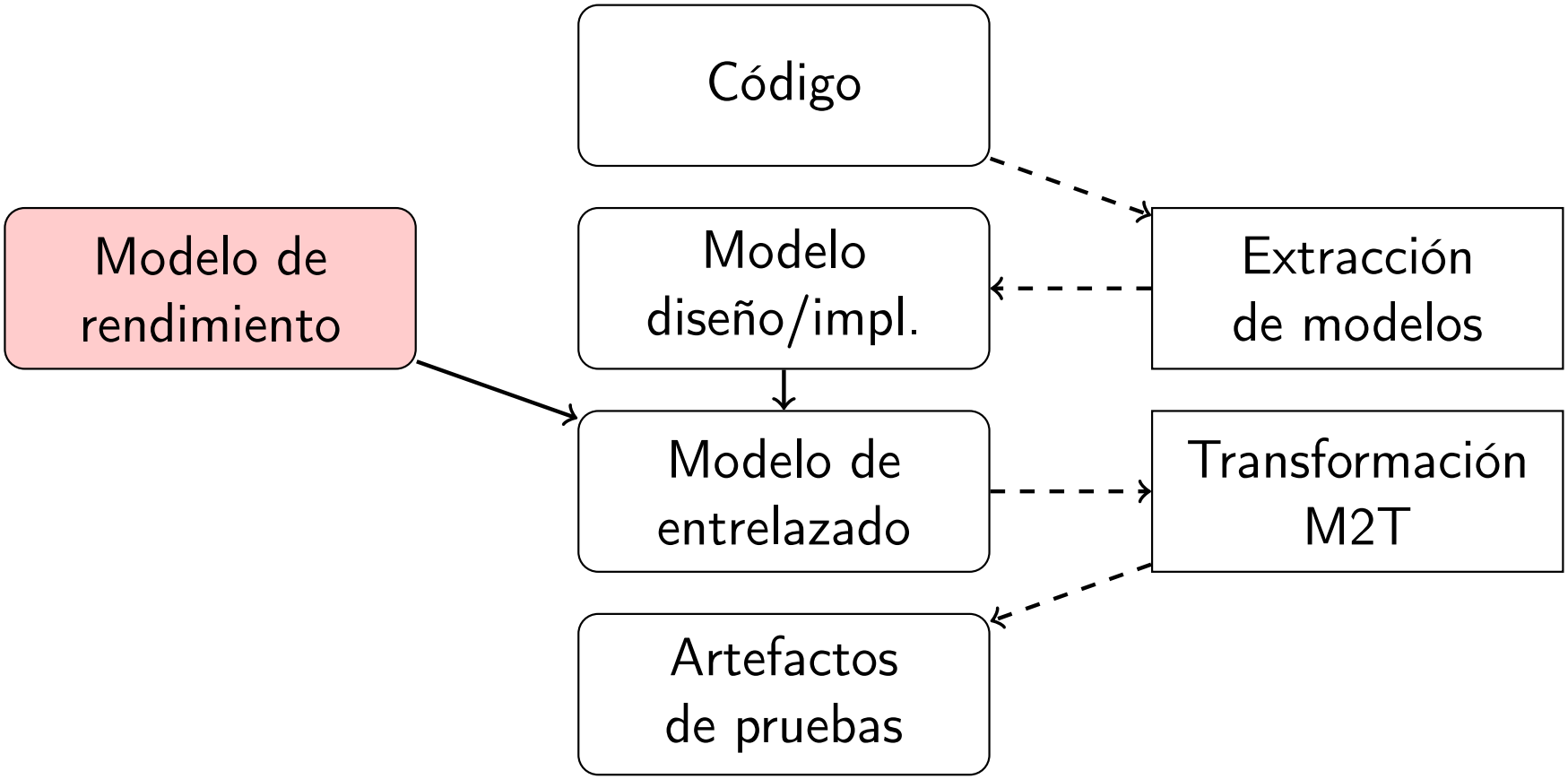
Inferencia de tiempos límite



Inferencia de tiempos límite



Enfoque general para generación de pruebas



Implementaciones hechas

Java

- Código: pruebas unitarias JUnit.
- Extractor de modelos existente: MoDisco.
- Se generan nuevas pruebas JUnit que envuelven a las anteriores usando la biblioteca ContiPerf.

WSDL

- Código: interfaz WSDL de un servicio Web.
- Extractor de modelos propio: ServiceAnalyzer.
- Se generan plantillas de mensajes y un conjunto aleatorio de entradas mediante SpecGenerator y TestGenerator (propios).
- Se genera código Jython para realizar pruebas de rendimiento mediante la herramienta The Grinder.

Estado actual y trabajo futuro

¿Qué está hecho?

- Modelos de rendimiento basados en OMG MARTE
- Algoritmos para tiempos límite y peticiones por segundo
- Generación de pruebas de rendimiento para código Java y servicios Web descritos mediante WSDL

¿Qué está por hacer?

- Añadir más opciones para generar entradas a las pruebas de rendimiento desde documentos WSDL
- Validar el enfoque con casos de estudio mayores

Fin de la presentación

¡Gracias por su atención!

Código y descargas:
`https://neptuno.uca.es/redmine/projects/sodmt`

E-mail:
antonio.garciadominguez@uca.es

Twitter:
`@antoniogado`

Generación desde pruebas unitarias en Java

Muchos SW se hacen en Java (Apache Axis, Apache CXF) y se prueban con JUnit

Código | Desc. modelos | Modelo entrel. | Artefactos de prueba

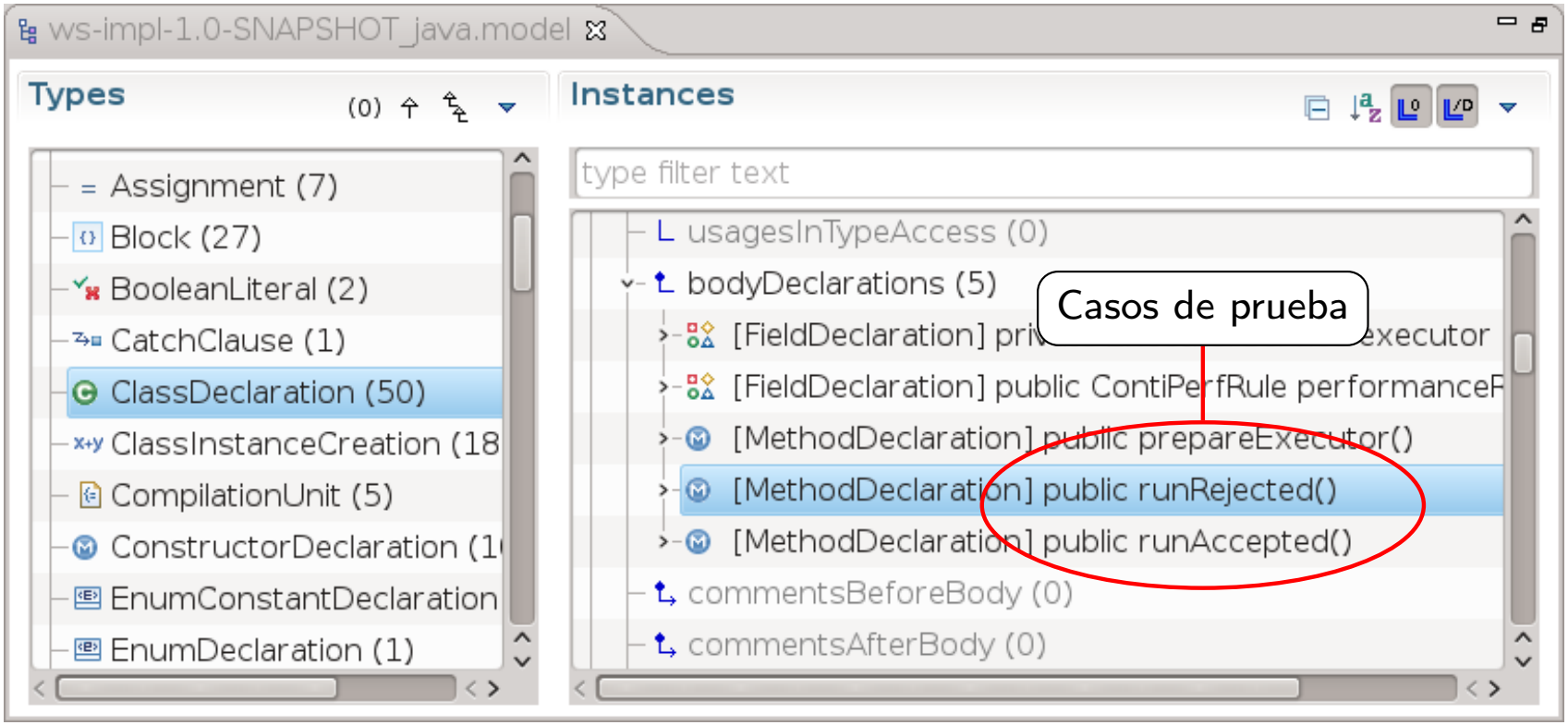
```
public class MisPruebasUnitarias {
  @Test
  public void rechazado() {
    // ... test code ...
  }

  @Test
  public void aceptado() {
    // ... test code ...
  }
}
```

Generación desde pruebas unitarias en Java

Muchos SW se hacen en Java (Apache Axis, Apache CXF) y se prueban con JUnit

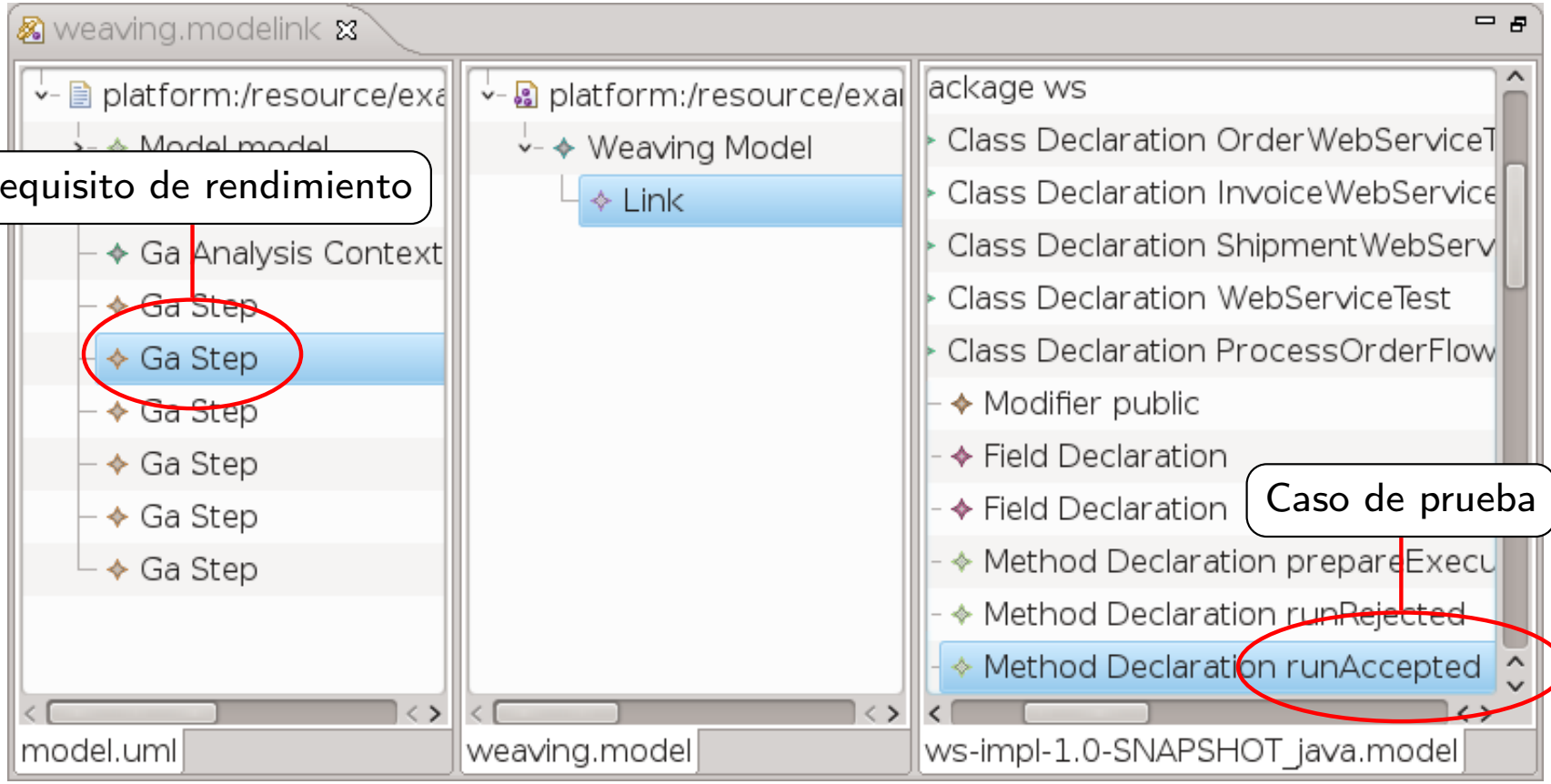
Código | **Desc. modelos** | Modelo entrel. | Artefactos de prueba



Generación desde pruebas unitarias en Java

Muchos SW se hacen en Java (Apache Axis, Apache CXF) y se prueban con JUnit

Código | Desc. modelos | Modelo entrel. | Artefactos de prueba



Generación desde pruebas unitarias en Java

Muchos SW se hacen en Java (Apache Axis, Apache CXF) y se prueban con JUnit

Código | Desc. modelos | Modelo entrel. | Artefactos de prueba

```
@RunWith(ContiPerfSuiteRunner.class)
@SuiteClasses(MyUnitTests.class)
@PerfTest( invocations = 100, threads = 10)
@Required(max=1000)
public class PruebaCargaInferida {}
```

- Se usa la biblioteca ContiPerf para generar menos código
- Se pueden convertir conjuntos completos o pruebas sueltas
- Se pueden expresar tiempos límite como máximos, promedios, percentiles (90 %, 95 % , 99 %) o medianas

Desde WSDL: extracción y entrelazado (1/2)

Enfoque independiente del lenguaje, pero WSDL es complejo

Diferencias

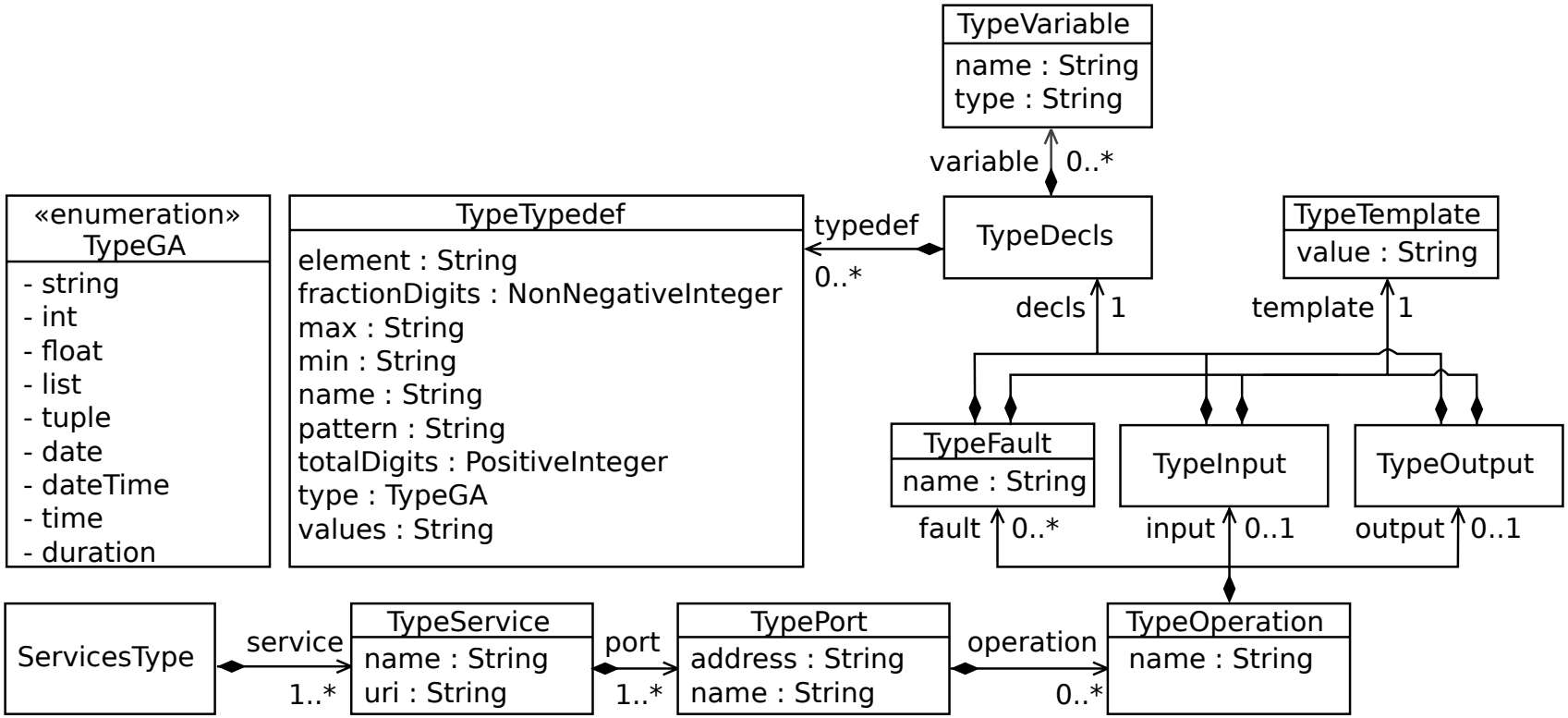
- Descripción WSDL no depende del lenguaje
- Herramienta especializada: The Grinder
- Se necesita generar entradas

Implementación actual

- Se extrae un "catálogo" de los documentos WSDL
- Se usa un metamodelo de entrelazado especializado:
  - Enlaza nodos ejecutables UML con operaciones del catálogo
  - Opciones de generación (Maven, Eclipse, The Grinder)

Desde WSDL: extracción y entrelazado (2/2)

Simplificamos los WSDL y añadimos plantillas de mensajes



Desde WSDL: generación (1/3)

Actualmente: aleatoria uniforme, servicios sin estado

- Las variables definen tipos, rangos y restricciones
- Se generan valores aleatorios y se pasan a las plantillas
- Plantillas, declaraciones y valores son personalizables

Ejemplo de declaraciones

typedef int (min=0, max=100) TArtID;
typedef float (min=0.01, max=2000) TPrice;
typedef list (element=TPrice, min=1, max=1) TL_float;
typedef tuple (element={TArtID, TL_float}) TArticleQtys;
typedef list (element=TArticleQtys, min=0) TOrder;
typedef list (element=TOrder, min=1, max=1) TEvaluate;
TEvaluate evaluate;

Desde WSDL: generación (2/3)

Ejemplo de plantilla Apache Velocity

```
<w:evaluate xmlns:w="http://ws.sodmt.uca.es/">
  #foreach($V1 in $evaluate)
  <newOrder>
    #foreach($V2 in $V1)
    <articleQtys>
      <id>$V2.get(0)</id>
      #foreach($V3 in $V2.get(1))<qty>$V3</qty>#end
    </articleQtys>
    #end
  </newOrder>
  #end
</w:evaluate>
```

Desde WSDL: generación (3/3)

Código de pruebas

- Se generan guiones Jython para The Grinder
- Regeneran las pruebas si el usuario ha retocado las declaraciones

Infraestructura

- Se genera un módulo Maven: desde una terminal, `mvn post-integration-test` lanza las pruebas y genera los informes
- Se genera un proyecto Eclipse (aún básico)