Inferencia Automática de Requisitos Locales de Rendimiento en Flujos de Trabajo Anotados con MARTE

Antonio García Domínguez e Inmaculada Medina Bulo

Escuela Superior de Ingeniería, Universidad de Cádiz,
C/Chile, 1, CP 11002
{antonio.garciadominguez,inmaculada.medina}@uca.es
http://neptuno.uca.es/{~agarcia,~imedina}

Resumen Obtener el rendimiento esperado de un flujo de trabajo sería más fácil si cada tarea incluyera sus expectativas individuales. Sin embargo, normalmente sólo se tienen requisitos globales de rendimiento, y los requisitos locales son derivados a mano. Esto lleva a errores y requiere revisar las estimaciones cada vez que se cambia el flujo de trabajo. En este trabajo presentamos dos algoritmos que infieren automáticamente restricciones locales de tiempos máximos y peticiones por segundo a partir de restricciones globales. Los flujos de trabajo son diagramas de actividad UML con anotaciones del perfil MARTE. El algoritmo de inferencia de tiempos límite reparte el tiempo disponible según los pesos de cada tarea y sus repeticiones, y comprueba que las restricciones locales sean consistentes con las globales. Opera de forma incremental, evitando caminos redundantes. Ambos algoritmos se han integrado en el editor UML Papyrus del proyecto Eclipse.

Keywords: ingeniería de rendimiento, pruebas del software, flujos de trabajo, UML, MARTE, acuerdos de nivel de servicio.

1. Introducción

El software debe cumplir requisitos tanto funcionales como no funcionales. Dentro de los no funcionales, los requisitos de rendimiento son muy comunes, y en ciertos contextos tienen tanta importancia como los requisitos funcionales. Además de los sistemas de tiempo real (duros o blandos), hay que considerar el caso de las arquitecturas orientadas a servicios (*Service Oriented Architectures* o SOA) [3]. En las SOA, es común asignar a cada servicio un acuerdo de nivel de servicio (*Service Level Agreement* o SLA): en caso de no cumplirse, el proveedor del servicio puede tener que compensar al consumidor de alguna forma.

Dada la importancia de los requisitos de rendimiento, existe una variedad de propuestas para su estimación y medición [11]. Realizar estimaciones normalmente requiere crear modelos abstractos de alto nivel de la ejecución y arquitectura del sistema y utilizar un método formal para analizarlo, como se hace en [9]. Por otro lado, medir el rendimiento conlleva instrumentarlo para obtener

la información necesaria. Estos enfoques se complementan entre sí: antes de implementar el sistema se pueden usar estimaciones, y una vez se ha implementado, las mediciones pueden dar más información.

Medir el rendimiento de un sistema puede servir para diversos objetivos, como detectar pérdidas de rendimiento a largo plazo, identificar patrones de carga o comprobar si se están cumpliendo sus requisitos de rendimiento. Para comprobar los requisitos de rendimiento, se necesita que hayan sido definidos antes: sin embargo, normalmente no se suelen dar requisitos detallados de rendimiento. Esto exige cumplir requisitos de alto nivel sin saber realmente qué rendimiento se necesita en cada parte del sistema.

En este trabajo se propone un enfoque dirigido por modelos para derivar los requisitos de rendimiento de bajo nivel a partir de requisitos de alto nivel. Se describen dos algoritmos de inferencia sobre diagramas de actividad UML con anotaciones del perfil MARTE [7]. Estos algoritmos usan las anotaciones para completar el modelo con el rendimiento a exigir en cada actividad.

El resto de este trabajo se divide en varias secciones. Tras introducir el perfil MARTE y dar ejemplos en la sección 2, se describen los algoritmos en la sección 3. Se evalúa la funcionalidad y rendimiento de los algoritmos en la sección 4. En la sección 5 se comentan diversos trabajos relacionados. Finalmente, en la sección 6 se resumen las contribuciones de este artículo y se listan líneas de trabajo futuro.

2. Perfil UML MARTE

UML es el estándar *de facto* para modelar software hoy en día. Sin embargo, UML por sí sólo no permite modelar aspectos no funcionales en profundidad, como el rendimiento esperado. Por este motivo, el *Object Management Group* (OMG) ha ido publicando y actualizando diversos perfiles para describir aspectos de rendimiento en UML. El más reciente en esta línea es el perfil *Modelling and Analysis of Real-Time and Embedded Systems* (MARTE) [7]. MARTE incluye tanto un vocabulario predefinido de restricciones no funcionales, como un marco para definir nuevas restricciones.

En esta sección se describe la parte de MARTE que utilizan los algoritmos de inferencia, con ejemplos de las entradas y salidas de los algoritmos descritos en la sección 3. Para una introducción general a MARTE, se recomienda consultar los tutoriales oficiales disponibles en [8].

2.1. Subconjunto a Usar

MARTE incluye varios subperfiles con vocabularios predefinidos para diversos conceptos de calidad de servicio. Entre ellos, nos interesa el subperfil *Generic Quantitative Analysis Modelling* (GQAM). La figura 1 muestra un diagrama de clases UML con la parte de MARTE que usan los algoritmos de inferencia. Los atributos no empleados han sido omitidos.

Todos los requisitos no funcionales de los subperfiles de MARTE comparten una serie de características, ya que heredan de la clase NFP_CommonType.

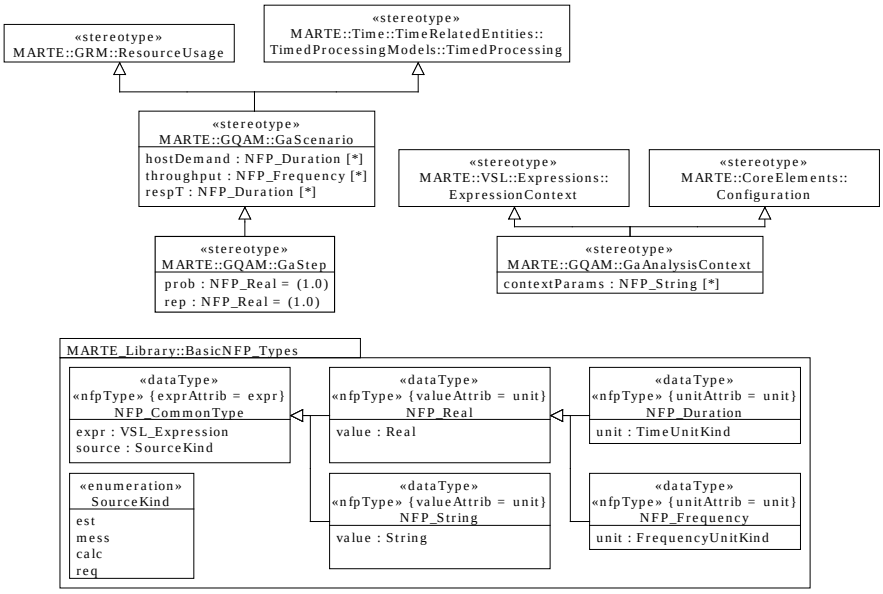


Figura 1. Diagrama de clases de la parte usada de MARTE

Se les pueden asignar valores constantes mediante el atributo *value*, o en base a una expresión en el *Value Specification Language* (VSL) definido en MARTE, usando el atributo *expr*. El origen de un requisito (estimado, medido, calculado o exigido) se indica en el atributo *source*.

Las instancias de NFP_CommonType son tuplas VSL, y su notación textual sigue el patrón `(clave1=valor1,...,claveN=valorN)`. Por ejemplo, una duración (NFP_Duration) de 5 milisegundos exigida por el cliente se expresaría usando `(value=5, unit=ms, source=req)`.

2.2. Ejemplo

La figura 2 muestra el diagrama de actividad UML que usaremos en los algoritmos de la sección 3. La actividad «Gestionar Pedido» describe cómo procesar un pedido. Empezando por el nodo inicial, evalúa el pedido y decide si rechazarlo o aceptarlo. Si se rechaza, el pedido se cierra y se termina la actividad. Si se acepta, se ejecutan dos ramas concurrentemente: mandar la orden de envío por un lado, y enviar la factura y realizar el pago por el otro. Una vez ambas ramas han terminado, se cierra el pedido y se termina la actividad.

La actividad UML y cada una de las acciones incorporan anotaciones MARTE con los requisitos de rendimiento. Algunas de estas anotaciones son entradas del algoritmo, y otras (en negrita) son salidas inferidas por los algoritmos:

- Cada acción tiene una anotación «GaStep». Inicialmente, el usuario establece en el atributo *hostDemand* el tiempo mínimo m que debe concederse,

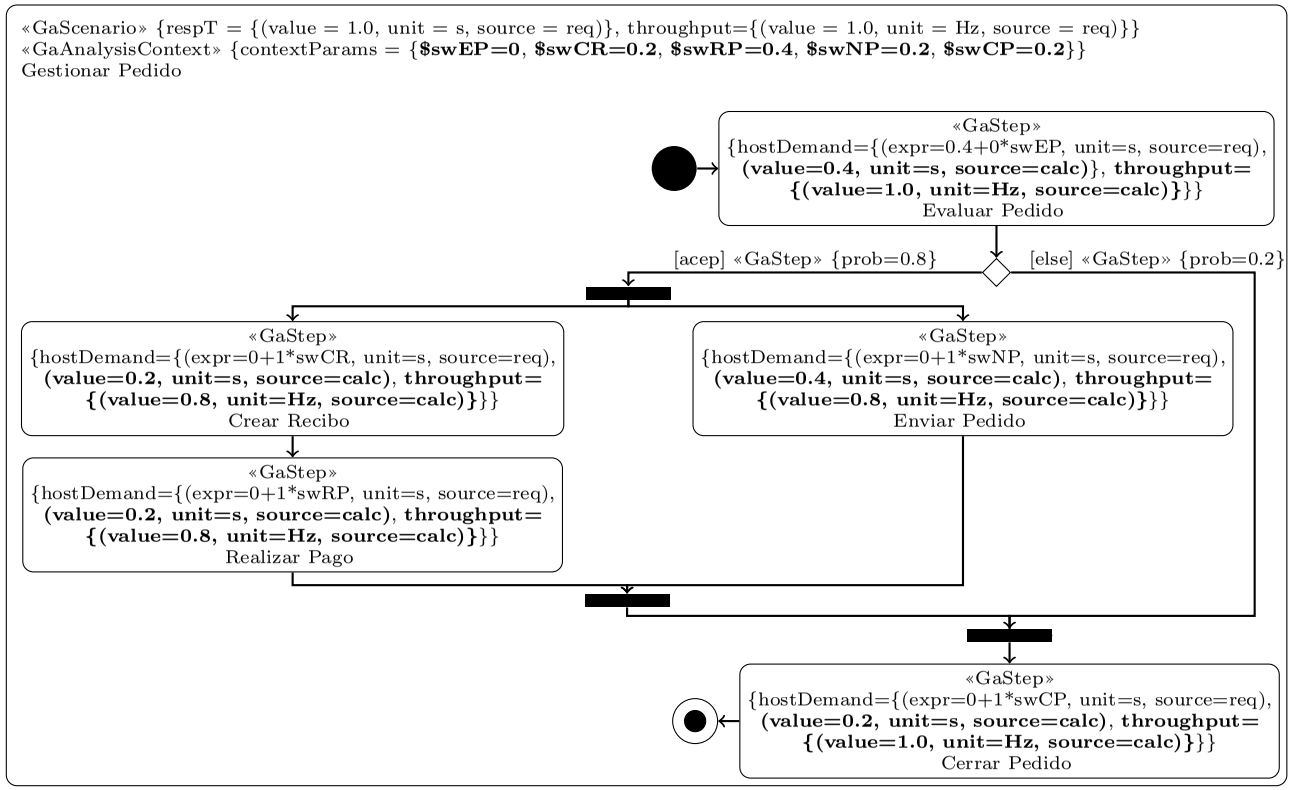


Figura 2. Ejemplo de diagrama de actividad UML (salidas de los algoritmos en negrita)

el peso w que tiene para repartir el tiempo sobrante y la variable v que le corresponde de entre las declaradas en el atributo *contextParams* del estereotipo «GaAnalysisContext». Para ello, se utiliza una expresión con el formato $m + w \times v$. La variable v representa la cantidad de tiempo por unidad de peso que se le concede a la acción más allá de su tiempo mínimo. Se puede utilizar el atributo *rep* para indicar su número de ejecuciones: si no se utiliza, se asume que se ejecuta una sola vez.
El algoritmo de inferencia de tiempos límite añade una restricción más a *hostDemand* con el límite de tiempo calculado (0,4 segundos en «Evaluar Pedido», por ejemplo). El algoritmo de inferencia de peticiones por segundo añade el atributo *throughput*, indicando el número de peticiones que deberá atender la acción (1 por segundo en «Evaluar Pedido»).

- Cada una de las aristas salientes de un nodo de decisión se encuentra anotada con el estereotipo «GaStep». En este caso, sólo se utiliza el atributo *prob* como entrada, para indicar la probabilidad de que se cumpla la condición de dicha arista. Por ejemplo, en la figura 2 se indica que el 20 % de los pedidos son rechazados.
- La actividad completa tiene de entrada un estereotipo «GaScenario» con el tiempo máximo de respuesta para toda invocación en *respT* (aquí 1 segundo) y con el número de peticiones que deben atenderse por segundo en *throughput* (1 petición por segundo).
  El atributo *contextParams* del estereotipo «GaAnalysisContext» es tanto entrada como salida: inicialmente sólo incluye los nombres de las variables a emplear en el resto del modelo, y el algoritmo de inferencia de tiempos límite las inicializa con valores concretos.

3. Algoritmos de Inferencia

En la sección anterior se ha presentado el perfil MARTE y se ha mostrado un ejemplo del formato de las entradas y las salidas de los algoritmos. En esta sección definiremos los dos algoritmos de inferencia. El primero calcula el número de peticiones por segundo que debe atender cada acción, y el segundo obtiene el tiempo límite de cada acción. Son versiones mejoradas de los algoritmos de [5]: el algoritmo de tiempos límite obtiene mejor rendimiento gracias a un cálculo incremental de los resultados y da mayor control sobre los tiempos a asignar, el algoritmo para peticiones por segundo ha sido simplificado y ambos algoritmos han sido adaptados a una notación estándar (MARTE). Ambos algoritmos requieren que las actividades no tengan ciclos, que tengan sólo un nodo inicial, y que todas las acciones sean alcanzables a través de él.

3.1. Definiciones

- $s(e)$ y $g(e)$ son los nodos origen y destino de la arista e, respectivamente.
- $i(n)$ y $o(n)$ son los arcos entrantes y salientes del nodo n, respectivamente.
- $L > 0$ es el tiempo máximo de respuesta de la actividad, medido en segundos.

- $c(n) = (m(n), w(n)) \in C(L)$ es la *restricción* del nodo n, donde $m(n)$ es el tiempo mínimo a conceder a n y $w(n)$ es su peso proporcional a la hora de repartir el tiempo sobrante, en relación con el resto de nodos del camino más restrictivo en que participe.
- El conjunto de todas las restricciones válidas con L como tiempo de respuesta global es $C(L) = \{(m, w) \mid 0 \leq m \leq L, w \geq 0\}$.
- De las restricciones de cada nodo se derivan las restricciones de cada camino p, $c(p) = (m(p), w(p)) \in C(L)$, donde $m(p) = \sum_{n \in p} m(n)$ y $w(p) = \sum_{n \in p} w(n)$.
- La ejecución de un nodo n se repite $R(n) \geq 1$ veces (una por omisión).

3.2. Inferencia de Peticiones por Segundo

Definiremos el algoritmo como una función T que recibe un nodo o arista y produce el número de peticiones por segundo que deberá atender. Para un flujo de control e, $T(e) = P(e)T(s(e))$, donde $P(e)$ es la probabilidad de atravesar e.

Para un nodo n, la fórmula depende de su tipo. Para un nodo inicial, $T(n)$ es igual al atributo *throughput* de la actividad completa. Para un nodo de reunión («join»), $T(n) = \text{mín}_{e \in i(n)} T(e)$, ya que las peticiones en la rama más lenta marcan el ritmo. Para un nodo de fusión («merge»), $T(n) = \sum_{e \in i(n)} T(e)$, puesto que une ramas cuyas ejecuciones son mutuamente excluyentes. Para cualquier otro tipo de nodo, $T(n) = T(e_1)$, donde $e_1 \in i(n)$ es su única arista entrante.

Usando estas fórmulas, calcular T(Crear Recibo) para el ejemplo de la figura 2 requiere subir hasta el nodo inicial, encontrar un flujo de control con una probabilidad de $p = 0{,}8$ y un nodo inicial que recibe una petición por segundo ($L = 1$). El valor obtenido es $pL = 0{,}8$.

Para calcular estos valores de forma eficiente, las expresiones se evalúan recorriendo el grafo en orden topológico. Para cada acción a, *throughput* tendrá una tupla de la forma `(value=T(a),unit=Hz,source=calc)`.

3.3. Inferencia de Tiempos Límite

Este algoritmo es considerablemente más complejo que el anterior. A continuación daremos una serie de definiciones, describiremos el algoritmo y mencionaremos algunas de las optimizaciones clave. Por último, lo aplicaremos al ejemplo de la figura 2.

Definiciones El algoritmo añade una tupla al atributo *hostDemand* de cada acción n con el formato `(value=t(n), unit=s, source=calc)`, donde $t(n)$ es el tiempo límite estimado. El algoritmo también inicializa la variable correspondiente a n (de las declaradas en la actividad) con el tiempo concedido por unidad de peso más allá del tiempo mínimo.

Sea I el nodo inicial de la actividad y sea $P_S(n)$ el conjunto de todos los caminos del nodo n a un nodo final. $t(n)$ debe cumplir los siguientes requisitos:

- Para cada acción n, $t(n) \geq m(n)$: el tiempo concedido debe ser mayor o igual al mínimo que estableció el usuario.
- Para cada camino p en $P_S(I)$, $\sum_{n \in p} R(n)t(n) \leq L$: las sumas de los tiempos concedidos a cada camino cumplen el tiempo límite global.

El tiempo límite «fluye» del nodo inicial. Si un nodo n recibe $0 \leq r(n) \leq L$ segundos, cada camino $p \in P_S(n)$ que empiece en él recibe $r(p) = r(n)$ para distribuir entre sus nodos. A priori, sólo se sabe que $r(I) = L$. Nótese que $r(n)$ indica el tiempo que recibe el nodo n, y $R(n)$ indica el número de repeticiones esperadas de n.

Si las anotaciones «GaStep» y «GaScenario» son consistentes entre sí, entonces $r(p) \geq m(p)$ para cada camino p y siempre se concede el tiempo mínimo requerido. $s(p) = r(p) - m(p)$ se define como el *margen* del camino p. $s(p)$ se distribuye a lo largo de p de acuerdo con el peso de cada nodo: el *margen por unidad de peso* inicialmente asignado a cada nodo es $S_W(p) = s(p)/w(p)$. Cuando $w(p) = 0$, se asume que $S_W(p) = 0$: como todos los nodos en p tienen peso nulo, no se puede repartir el margen disponible.

Los algoritmos deben comprobar que $w(p) > 0 \Rightarrow s(p) > 0$, para que cada camino p con un peso no nulo tenga margen que distribuir. Si no se cumple esta restricción, las anotaciones son inconsistentes entre sí: esto debe indicarse al usuario, y todos los cambios deberán ser cancelados.

Esquema general El algoritmo es una función recursiva que recibe un nodo n y el tiempo que recibe, $r(n)$. Inicialmente, $n = I$ y $r(I) = L$, el tiempo límite global. El algoritmo sigue estos pasos:

1. Seleccionar dos caminos de $P_S(n)$:
   - $p_{ms}(n)$ tiene el mínimo $S_W(p)$ cuando se dispone de $r(n)$ segundos, o en caso de empate, el máximo $w(p)$.
   - $p_{Mm}(n)$ tiene el máximo $m(p)$.
2. Si $s(p_{Mm}(n)) < 0$, los tiempos mínimos no se pueden cumplir: abortar.
3. Si $s(p_{ms}(n)) = 0$ y $w(p_{ms}(n)) > 0$, no hay margen en un camino con un peso no nulo: abortar.
4. Establecer el tiempo límite de n a $t(n) = m(n) + S_W(p_{ms}(n))w(n)$ y marcar n como visitado. Sobrarán $T_R = r(n) - R(n)t(n)$ segundos.
5. Enviar T_R segundos a cada arista saliente e de n, y ordenarlas de forma ascendente según el valor de $S_W(p_{ms}(g(e)))$, el tiempo mínimo por unidad de peso para todos los caminos que comienzan en el destino de e.
6. Visitar cada arista saliente e:
   a) Si el destino de e ha sido visitado antes, comprobar si el tiempo que se le envió T'_R es estrictamente menor que T_R, el tiempo que se habría enviado por e. En dicho caso, se intentarán aprovechar los $T_R - T'_R$ segundos sobrantes en el origen de e y sus ancestros. Para ello, se define A como la secuencia de nodos con pesos no nulos desde el origen de e hacia atrás, hasta un nodo con más de una arista entrante o saliente (no inclusive). El tiempo límite de cada nodo en A se incrementará por $(T_R - T'_R)w(n)/w(A)$ segundos, donde $w(A) = \sum_{n \in A} R(n)w(n)$.

b) Si el destino de e no ha sido visitado antes, se invocará este algoritmo recursivamente, con $n = g(e)$ y $r(n) = T_R$.

7. Establecer la variable en *contextParams* de «GaAnalysisContext» relacionada con n a 0 si $w(n) = 0$ o a $(t(n) - m(n))/w(n)$ en otro caso.

Optimizaciones Este algoritmo utiliza varias optimizaciones para mejorar su eficiencia. En primer lugar, cada camino p no se representa como una secuencia de nodos, sino con su restricción $c(p) = (m(p), w(p))$, ahorrando memoria.

Para seleccionar $p_{Mm}(n)$ en cada nodo se necesita saber el $m(p)$ máximo de cada camino $p \in P_S(n)$, que denotaremos con $m(p_{Mm}(n))$. Podemos calcularlo por adelantado con (1). Como es una fórmula recursiva, se puede evaluar de forma incremental desde los nodos finales (con $m(p_{Mm}(n)) = 0$) hasta los iniciales, en orden topológico inverso.

$$\begin{aligned} m(p_{Mm}(n)) = R(n)m(n) \\ + \text{máx}\{m(p_{Mm}(g(e))) \mid e \in o(n)\} \end{aligned} \tag{1}$$

Para seleccionar $p_{ms}(n)$ en cada nodo necesitamos saber el camino más estricto que empieza en n. No se puede calcular por adelantado, ya que depende del tiempo recibido por el nodo, que es desconocido a priori. Sin embargo, podemos retirar caminos redundantes de $P_S(n)$, obteniendo el conjunto reducido $P'_S(n)$. Se retira un camino p_a cuando se dice que es *siempre menos o igual de estricto* cuando el límite es L (denotado por $c(p_a) \preceq_{s(L)} c(p_b)$) que otro camino p_b, independientemente del tiempo que reciba n o de las restricciones de los ancestros comunes de p_a y p_b. $\preceq_{s(L)}$ se define como:

$$\begin{aligned} (a, b) \preceq_{s(L)} (c, d) \equiv \\ \forall t \in [0, L]\ \forall x \in [0, L]\ \forall y \geq 0 \\ a + x \leq t \wedge c + x \leq t \wedge \\ b + y > 0 \wedge d + y > 0 \Rightarrow \\ \frac{t - (a + x)}{b + y} \geq \frac{t - (c + x)}{d + y} \end{aligned} \tag{2}$$

Se puede simplificar (2) a:

$$a \leq c \wedge (b \leq d \vee a < L \wedge b > d \wedge (b - d)L \leq bc - ad) \tag{3}$$

Puede demostrarse que esta expresión define un orden parcial (una relación binaria reflexiva, antisimétrica y transitiva) sobre $C(L)$. Por brevedad, se ha omitido la demostración.

Como $m(p_{Mm}(n))$, $P'_S(n)$ puede calcularse de forma incremental en un recorrido en orden topológico inverso. Sea n_i un hijo de n y sean p_a y p_b dos caminos en $P_S(n_i)$, de forma que $c(p_a) \preceq_{s(L)} c(p_b)$. Por definición, p_a es menos o igual de estricto que p_b independientemente de sus ancestros comunes, así que $\langle n \rangle + p_a$ también se descartaría de $P'_S(n)$, al compararse con $\langle n \rangle + p_b$. Esto significa que en vez de comparar cada camino en $P_S(n)$ para cada nodo n, se puede construir $P'_S(n)$ añadiendo n al comienzo de los caminos en $P'_S(n_i)$ para cada hijo n_i de n, y entonces filtrando los caminos redundantes con $\preceq_{s(L)}$.

Ejemplo En esta sección aplicaremos el algoritmo al modelo de la figura 2. Primero calculamos $m(p_{Mm}(n))$ y $P'_S(n)$):

- $m(p_{Mm}(\text{Cerrar Pedido})) = 0$, $P'_S(\text{Cerrar Pedido}) = \{(0,1)\}$.
- $m(p_{Mm}(\text{Realizar Pago})) = 0$, $P'_S(\text{Realizar Pago}) = \{(0,2)\}$.
- $m(p_{Mm}(\text{Crear Recibo})) = 0$, $P'_S(\text{Crear Recibo}) = \{(0,3)\}$.
- $m(p_{Mm}(\text{Enviar Pedido})) = 0$, $P'_S(\text{Enviar Pedido}) = \{(0,2)\}$.
- $m(p_{Mm}(\text{Evaluar Pedido})) = 0{,}4$, $P'_S(\text{Evaluar Pedido}) = \{(0{,}4,3)\}$.

A continuación, el algoritmo envía el tiempo disponible ($L = 1s$) al nodo inicial y luego a «Evaluar Pedido», que toma $0{,}4s$ y envía los $0{,}6s$ sobrantes a través del nodo de decisión. El resto del camino más estricto está formado por «Crear Recibo» (toma $0{,}2s$ y envía $0{,}4s$), «Realizar Pago» (ídem) y «Cerrar Pedido» (utiliza los $0{,}2s$ restantes). Tras acabar con el camino más estricto, se va hacia atrás y se procede con el siguiente más estricto, enviando $0{,}4s$ a «Enviar Pedido». En principio sólo toma $0{,}3s$ y envía los $0{,}3s$ restantes, pero como «Cerrar Pedido» recibió antes sólo $0{,}2s$, aprovechamos los $0{,}1s$ sobrantes en «Enviar Pedido», obteniendo un tiempo límite de $0{,}4s$. Seguimos con la rama para los pedidos rechazados, sin nada que hacer: hemos acabado.

4. Evaluación

Los algoritmos se han implementado en el *Epsilon Object Language* (EOL) [6] y se han integrado en el editor Papyrus [2]. El código está disponible en [4]. En esta sección analizaremos sus limitaciones y su rendimiento.

4.1. Limitaciones

Los algoritmos están limitados en varios aspectos. La restricción más importante es que el grafo formado por los nodos de la actividad debe ser acíclico, dificultando el modelado de estructuras iterativas. Se debe a que los algoritmos trabajan con el conjunto de todos los posibles caminos desde el nodo inicial hasta un nodo final: al introducir un ciclo, se introduce un número infinito de posibles caminos. Para resolver este problema, se necesitaría limitar el número de veces que se pasa por el ciclo. En este trabajo, hemos optado por mantener el grafo acíclico y emplear el atributo *rep* de «GaStep» para las repeticiones esperadas.

A primera vista, el algoritmo sigue exigiendo al modelador anotar cada acción con cierta información. Sin embargo, la información a anotar ahora sólo depende de la acción o arista en sí, y no de todos los caminos en que participan. Además, no es necesario anotar todas las acciones.

Los algoritmos no tienen en cuenta el caso en que una misma acción es repetida en distintas partes del modelo: se asume que cada acción es distinta del resto. Una solución sencilla a este problema es inferir los tiempos con esta suposición, y si se repite la acción en varios sitios, tomar la restricción más fuerte.

4.2. Rendimiento Teórico

Consideremos una actividad con n nodos y $e \in O(n^2)$ aristas, con $O(n)$ aristas entrantes en cada nodo. El algoritmo de inferencia de peticiones por segundo es fácil de analizar: yendo de los nodos finales a los iniciales, cada nodo y arista debe ser visitado justo una vez. Calcular las peticiones de los $O(n)$ nodos de reunión («join») y fusión («merge») requiere evaluar una expresión en tiempo constante sobre sus $O(n)$ aristas entrantes. Las peticiones para el resto de las $O(n + e)$ aristas pueden calcularse en tiempo constante. Por tanto, una estimación pesimista de la cota superior del coste del algoritmo es $O(n)O(n) + O(n + e)O(1) = O(n^2)$. El tiempo sólo depende de la forma del grafo.

El algoritmo de inferencia de tiempos límite es más difícil de analizar, ya que su coste depende tanto de la forma del grafo como de los valores de las anotaciones. Por este motivo, nos centraremos en un tipo específico de actividad para enfocar el análisis, al que llamaremos *actividad de ramificación y reunión*. Tal y como se muestra en la figura 3, tiene un nodo inicial I, seguido de f *niveles*. Cada nivel se divide en dos ramas, cada una con una sola acción, que se reúnen antes del siguiente nivel. Estas actividades tienen $n = 2 + 4f \in \Theta(f)$ nodos y $e = 1 + 5f \in \Theta(f)$ aristas en total, y hay 2^f caminos del nodo inicial al nodo final. Estas actividades son baratas de generar y pueden representar el peor caso del algoritmo, ya que el número de caminos desde el nodo inicial al final crece exponencialmente en función de f. Con estas actividades, podemos ver que la parte más cara del algoritmo es el cálculo por adelantado de $P'_S(n)$, ya que hay que considerar todos los pares de los $O(2^f)$ posibles caminos en cada nodo. En total, esto exige $O(n4^f)$ operaciones. Sin embargo, este caso es muy poco común, como veremos en la siguiente sección.

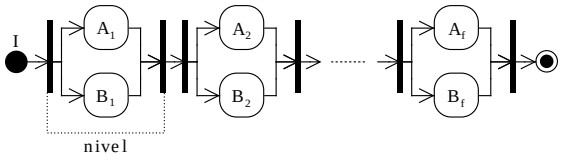


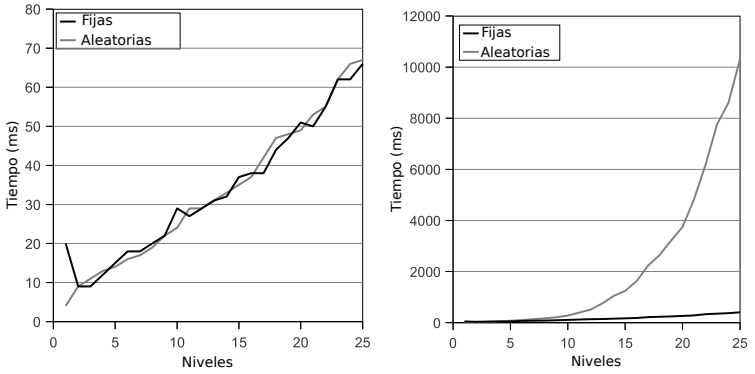
Figura 3. Actividad de ramificación y reunión con f niveles

4.3. Rendimiento Empírico

Anteriormente, concluimos que el algoritmo de inferencia de peticiones por segundo tenía coste polinómico independientemente de las anotaciones, y que el algoritmo de inferencia de tiempos límites podía llegar a tener coste exponencial, dependiendo de las anotaciones. En esta sección compararemos los casos promedios con los anteriores peores casos.

En primer lugar, medimos el rendimiento de los algoritmos usando actividades de ramificación y reunión de entre 1 y 25 niveles. Ejecutamos los algoritmos

exigiendo 1s de tiempo de respuesta cuando se hacía una petición por segundo. Las acciones se anotaron de dos formas: usando restricciones fijas (tiempo mínimo nulo y peso 1) o usando valores aleatorios distribuidos uniformemente, de forma que los tiempos límite mínimos fueran consistentes y los pesos estuvieran entre 0 y 1. Para simplificar el análisis, todas las acciones tuvieron *rep* a 1.



(a) Peticiones por segundo (b) Tiempos límite

Figura 4. Tiempos promedios (ms) sobre 10 ejecuciones de los algoritmos, usando anotaciones fijas y aleatorias, por número de niveles

Los resultados se muestran en la figura 4. Se confirma que el tiempo exigido por el algoritmo de inferencia de peticiones por segundo crece de forma lineal y es independiente de las anotaciones. Igualmente, se sugiere que el caso promedio para el algoritmo de inferencia de tiempos límite está muy lejos del caso peor teórico de $O(n4^f)$ operaciones.

Es importante destacar que cuando el tiempo límite mínimo es igual a 0 en todas las acciones, el orden parcial de (3) puede simplificarse a $a \leq c$, que es un orden total. Puede decirse que este es el mejor caso del algoritmo de inferencia de tiempos límite, en el que todos los caminos son comparables.

Se obtuvieron tiempos mucho mayores con anotaciones aleatorias, pero los tiempos no crecen tan rápido como se esperaría del caso peor de $O(n4^f)$ operaciones. Esto sugiere que retirar los caminos redundantes reduce el impacto del caso peor a niveles manejables. Sin embargo, su efectividad depende de la proporción entre los tiempos mínimos, los pesos y el tiempo límite global L. El operando izquierdo de $(b-d)L < bc - ad$, parte de (3), crece a medida que L aumenta, reduciendo los pares de caminos comparables.

Para evaluar la frecuencia del caso peor teórico y estudiar su relación con L, realizamos un estudio adicional barriendo el espacio de todas las posibles actividades de reunión y ramificación de hasta 3 niveles en las que todos los caminos de los niveles anteriores son incomparables entre sí. Probamos con $L = 0{,}5s$ y $L = 1{,}5s$. Los tiempos mínimos fueron de 0 a mín$\{L, 1\}$, en pasos de

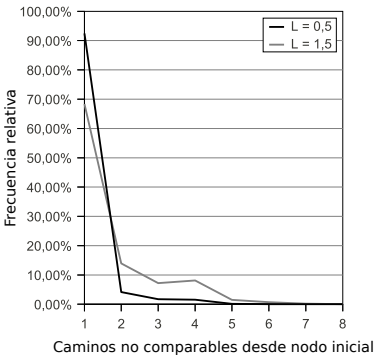


Figura 5. Porcentajes de actividades de ramificación y reunión de 3 niveles por número de caminos incomparables desde el nodo inicial, según tiempo límite global

0,1s. Los pesos fueron de 0 a 10, en pasos de 1 unidad. Se descartaron los grafos inconsistentes. Medimos el número de caminos no comparables entre sí en el nodo inicial: con 3 niveles, se pueden tener entre 1 y $2^3 = 8$ caminos.

Tras evaluar $1{,}99 \times 10^6$ actividades con $L = 0{,}5s$ and $7{,}16 \times 10^9$ con $L = 1{,}5s$, obtuvimos los resultados en la figura 5. Es interesante ver que para $L = 1{,}5s$, el peor caso constituía el 31,842 % de todas las actividades de 1 nivel, 2,492 % de las de 2, y 0,047 % de las de 3. Para $L = 0{,}5s$ se obtienen resultados parecidos, y además no hay actividades de 3 niveles en el peor caso. Esto sugiere que el peor caso teórico se vuelve más difícil de encontrar a medida que los grafos crecen, y explica por qué los tiempos no crecieron exponencialmente en la figura 4. También indica que el rendimiento empeora a medida que L crece en proporción a los valores usados en las restricciones.

5. Trabajos Relacionados

Existen básicamente dos enfoques para evaluar el rendimiento de un sistema: estudiar un modelo del sistema o medirlo una vez esté implantado. Ambos enfoques son complementarios: un modelo analítico reduce el riesgo de implementar una arquitectura de software ineficiente, y medir el rendimiento del sistema una vez implementado proporciona mayor precisión y nivel de detalle. Nuestro trabajo se acerca más al enfoque basado en modelos analíticos.

Usar modelos analíticos requiere conocimiento y notaciones muy especializadas. La adopción generalizada de UML como una notación estándar *de facto* ha hecho que los investigadores deriven sus modelos analíticos a partir de modelos UML, primero con anotaciones *ad hoc* y, posteriormente con extensiones estándar de UML como las del perfil MARTE. El estudio realizado en [12] revisa muchos de los enfoques anteriores a la aparición de MARTE en 2009. Desde entonces MARTE se ha usado para muchos propósitos. Por ejemplo, Tribastone et al. deducen especificaciones en álgebra de procesos [10] y Yang et al. obtienen redes de Petri extendidas [13]. Nuestro trabajo opera directamente sobre

el modelo, en vez de traducirlo a un formalismo intermedio. Estos formalismos también pueden calcular las peticiones por segundo de cada acción en la actividad, pero las derivan a partir de información local a cada acción, y no desde requisitos globales sobre la actividad como en nuestro caso. En cuanto a los tiempos límite, nuestra línea de trabajo es la primera en que se derivan desde requisitos globales, en vez de ser una entrada del proceso de análisis.

Alhaj y Petriu han generado modelos analíticos de rendimiento intermedios a partir de un conjunto de diagramas UML anotados con el perfil MARTE, que describen una arquitectura orientada a servicios [1]: los diagramas de actividad de UML modelan los flujos de trabajo, los diagramas de componentes de UML representan la arquitectura, y los diagramas de secuencia de UML detallan el comportamiento de cada acción en los flujos de trabajo. En nuestro trabajo previo modelamos análogamente los flujos de trabajo en una arquitectura orientada a servicios que emplea una notación diseñada a propósito inspirada en los diagramas de actividad de UML [5]. Sin embargo, nuestro enfoque no modela los recursos utilizados por el sistema: suponemos que las pruebas se realizan en un entorno que emula al de producción. Alhaj y Petriu derivan un modelo analítico a partir del que sacar conclusiones, mientras que nuestro trabajo extiende el modelo original con las restricciones derivadas.

6. Conclusiones y Trabajo Futuro

Además de sus requisitos funcionales, todo software necesita cumplir sus requisitos de rendimiento. Para conseguir esta meta, puede estimarse el rendimiento esperado con un modelo, o se puede medir el rendimiento del sistema implementado. En la actualidad, se está adoptando el perfil UML MARTE [7] como notación estándar para realizar los análisis de rendimiento iniciales. Por otro lado, las pruebas de rendimiento exigen que se definan los requisitos concretos para cada parte del sistema. Sin embargo, estas expectativas suelen estar disponibles sólo para elementos de alto nivel: los desarrolladores tienen que traducirlas manualmente a requisitos sobre sus componentes.

En este trabajo, hemos adaptado y mejorado los algoritmos de [5] para operar en diagramas de actividad UML anotados con el perfil MARTE, infiriendo restricciones de rendimiento a partir de requisitos globales y locales. Uno de los algoritmos infiere peticiones por segundo, y tiene coste polinómico sobre el número de nodos de la actividad. El otro infiere tiempos límite y su peor caso es exponencial, pero el caso promedio observado sugiere que el peor caso es muy difícil de encontrar, y se hace más difícil a medida que se complica el grafo. Esto es debido a que el algoritmo descarta caminos parciales redundantes.

Para las siguientes versiones de los algoritmos, se añadirá soporte para actividades expandibles, de forma que el usuario pueda describir el sistema de forma jerárquica e inferir restricciones de forma descendente. Laa restricción local inferida en un nivel superior servirá como restricción global para el nivel inferior.

Sería interesante manejar el caso en que una acción esté repetida en varias partes, pero podría ser muy costoso. Tras mejorar los algoritmos, nuestro prin-

cipal objetivo es generar casos de prueba para una herramienta existente. Una posibilidad es generar pruebas de rendimiento que envuelvan pruebas unitarias o de integración existentes. Otra posibilidad es generar planes parciales de pruebas para algunas de las herramientas para pruebas de rendimiento actualmente disponibles. Finalmente, se podrían derivar monitores de niveles de servicio a partir de las anotaciones inferidas.

Reconocimientos

Este trabajo fue parcialmente financiado por la beca de investigación PU-EPIF-FPI-C 2010-065 de la Universidad de Cádiz.

Referencias

1. Alhaj, M., Petriu, D.C.: Approach for generating performance models from UML models of SOA systems. In: Proc. of the 2010 Conf. of the Center for Advanced Studies on Collaborative Research. pp. 268–282. CASCON '10, ACM, New York, NY, USA (2010)
2. Eclipse Foundation: Homepage of the Eclipse MDT Papyrus project (2011), `http://www.eclipse.org/modeling/mdt/papyrus/`
3. Erl, T.: SOA: Principles of Service Design. Prentice Hall, Indiana, EEUU (2008)
4. García Domínguez, A.: Homepage of the SODM+T project. `https://neptuno.uca.es/redmine/projects/sodmt` (enero 2011)
5. García Domínguez, A., Medina Bulo, I., Marcos Bárcena, M.: Inference of performance constraints in Web Service composition models. CEUR Workshop Proc. of the 2nd Int. Workshop on Model-Driven Service Engineering 608, 55–66 (junio 2010), `http://ceur-ws.org/Vol-608/paper5.pdf`
6. Kolovos, D., Paige, R., Rose, L., Polack, F.: The Epsilon Book. `http://www.eclipse.org/gmt/epsilon` (2010)
7. OMG: UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) 1.0. `http://www.omg.org/spec/MARTE/1.0/` (noviembre 2009)
8. OMG: Official reference MARTE tutorials (mayo 2010), `http://www.omgmarte.org/node/28`
9. Smith, C.U., Williams, L.G.: Software performance engineering. In: Lavagno, L., Martin, G., Selic, B. (eds.) UML for Real: Design of Embedded Real-Time Systems, pp. 343–366. Kluwer, The Netherlands (mayo 2003)
10. Tribastone, M., Gilmore, S.: Automatic extraction of PEPA performance models from UML activity diagrams annotated with the MARTE profile. In: Proc. of the 7th Int. Workshop on Software and Performance. pp. 67–78. ACM, Princeton, NJ, USA (2008), `http://portal.acm.org/citation.cfm?id=1383569`
11. Woodside, M., Franks, G., Petriu, D.: The future of software performance engineering. In: Proc. of Future of Software Engineering 2007. pp. 171–187 (2007)
12. Woodside, M.: From annotated software designs (UML SPT/MARTE) to model formalisms. In: Proc. of the 7th Int. Conf. on Formal Methods for Performance Evaluation. pp. 429–467. Springer-Verlag, Bertinoro, Italy (2007)
13. Yang, N., Yu, H., Sun, H., Qian, Z.: Modeling UML sequence diagrams using extended Petri nets. In: Proc. of the 2010 Int. Conf. on Information Science and Applications. pp. 1–8 (2010)