Hacia la Validación de Arquitecturas de Software usando Alloy

María Marta Novaira y Sonia Permigiani
Departamento de Computación
Facultad de Ciencias Exactas, Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto
Ruta 36 Km. 601, Río Cuarto (5800)
Córdoba, Argentina
{mnovaira,spermigiani}@dc.exa.unrc.edu.ar

Resumen

En este trabajo, presentamos algunas líneas de trabajo que seguimos actualmente, con el objetivo principal de utilizar el lenguaje relacional Alloy para la validación de propiedades estáticas y dinámicas de arquitecturas de software. Alloy es un lenguaje de especificaciones con una semántica formal clara, basada principalmente en la noción de relación, y que ha ganado importancia en el último tiempo. Presentamos aquí de qué forma pueden especificarse usando Alloy algunos conceptos asociados a las arquitecturas de software, como componentes o conectores, y cómo podría, en principio, utilizarse el Alloy Analyzer para validar propiedades estáticas de arquitecturas.

Además, discutimos algunas de las limitaciones de Alloy para la especificación de arquitecturas de software, particularmente el modelado de propiedades de trazas de ejecución de sistemas basados en componentes.

1. Introducción

Alloy es un lenguaje de especificaciones formales con una semántica relacional clara, basada principalmente en la noción de relación [Jackson 2001][Jackson 2002]. Alloy ha ganado importancia, principalmente en la comunidad de métodos formales, gracias a su simplicidad y a que cuenta con soporte para la validación automática de especificaciones, mediante técnicas basadas en *SAT solving.* Esencialmente, el Alloy Analyzer (la herramienta asociada a Alloy) permite, dada una especificación de un sistema de software y una propiedad a validar, extraer contraejemplos con universos acotados, es decir modelos de la especificación del sistema que no cumplen la propiedad dada. Dado que la lógica de primer orden no es decidible, y la lógica en la cual está basada Alloy es una extensión de ésta, esto no puede considerarse un mecanismo de decisión, sino que simplemente es un mecanismo de exploración (exhaustiva) de interpretaciones acotadas en el tamaño de su universo por cierta constante k. Este mecanismo es, en general, muy útil en la práctica, ya que usualmente si una propiedad no es consecuencia de una especificación, ésta tiene contraejemplos de tamaños pequeños [Jackson 2002].

En este trabajo, discutimos sobre el uso de Alloy, y el Alloy Analyzer, para validar propiedades de arquitecturas de software. Mostramos cómo pueden modelarse algunos conceptos intrínsecos de arquitecturas de software, tales como componentes y conectores, y cómo puede, en principio, utilizarse el Alloy Analyzer para validar propiedades de arquitecturas. Discutimos además algunas limitaciones del lenguaje para analizar propiedades de arquitecturas, fundamentalmente propiedades de las llamadas *dinámicas*, es decir, aquellas propiedades que hablan de ejecuciones.

2. Modelando Arquitecturas

Presentamos aquí, brevemente y a través de un ejemplo, de qué manera se pueden modelar

arquitecturas de software en Alloy. El área de la Ingeniería de Software denominada arquitecturas de software [Garlan & Shaw 1993], se propone modelar sistemas de software a un nivel de abstracción alto, principalmente mediante el uso de las nociones de componente y conector. Las componentes representan las unidades de cómputo, y pueden asociarse a clases, módulos, subsistemas, etc, mientras que los conectores representan la interacción entre las componentes. La característica fundamental de la forma en que la interacción se representa en arquitecturas de software es que es *externa* a la definición de las componentes.

2.1. Modelando Componentes

Alloy posee una construcción que puede aprovecharse directamente para el modelado de componentes: las signaturas. Las signaturas pueden usarse para definir dominios de datos, y componentes con estructuras internas. Pensemos, por ejemplo, que deseamos modelar una arquitectura ampliamente conocida, *Productor-Consumidor*. Podemos comenzar diciendo que necesitaremos modelar los elementos que los productores producen y los consumidores consumen, es decir, los datos. En caso de requerir múltiples instancias de productores y consumidores para escenarios particulares, podríamos también pensar en dotar los productores y los consumidores con un nombre.

Los datos y los nombres de componentes pueden modelarse usando signaturas básicas (es decir, sin estructura interna):

```
sig Data { }
```

```
sig Name { }
```

Los productores y consumidores pueden también modelarse mediante signaturas, pero a diferencia de los nombres y los datos, éstas contarán con una estructura interna:

```
sig Producer {
        n: Name,
        queue: set Data,
        outgoing: Data
}
```

```
sig Consumer {
        n: Name,
        consumed: set Data,
        incoming: Data
}
```

Nótese que las signaturas denotan la estructura de los productores y consumidores, y no un productor y un consumidor (de alguna manera, describen el *tipo* de los consumidores y productores, de los cuales podremos tener varias instancias diferentes). Un productor tiene un nombre (el campo o atributo n), una cola modelada con un conjunto de datos (el campo queue) y un elemento listo para ser enviado (el campo outgoing). Un consumidor consta de un nombre (el campo n), un conjunto de elementos ya consumidos (el historial de consumo de un consumidor, modelado por el campo consumed) y un elemento de entrada, listo para ser consumido (el campo incoming). Nótese que, respetando la filosofía de las arquitecturas de software, tanto Producer como Consumer están libres de referencias a otras componentes. La interacción entre componentes se modela de manera externa a las componentes, mediante conectores.

2.1.1. Algunas Operaciones de Componentes

Las componentes de la sección anterior encapsulan información, modelada por los campos de las signaturas. Sin embargo, éstas no incluyen comportamiento. Podemos modelar comportamiento, es decir, operaciones de las componentes, mediante *funciones* en Alloy. Por ejemplo, podemos

considerar operaciones para producir y enviar datos en un productor, o para consumir datos en un consumidor:

```
fun produce ( p, p' : Producer, d: Data ){ p'.queue = p.queue + d }

fun send ( p, p' : Producer, d: Data ) {
    !(no p.queue) &&  d in p.queue && p'.outgoing= d && p'.queue = p.queue - d
}

fun receive ( c, c' : Consumer, d: Data ){
    c'.consumed = c.consumed + c.incoming
    c'.incoming=d
}
```

Las variables primadas entre los parámetros de las funciones denotan el estado de las componentes correspondientes después de la ejecución de las operaciones, aunque esto es sólo una convención [Frias et al. 2005]. El significado de estas operaciones es, a nuestro entender, bastante claro, gracias a la simplicidad de la sintaxis de Alloy.

2.2. Modelando Sistemas y Conectores

Gracias a que Alloy está basado en relaciones, y que las signaturas pueden contener campos de tipo relación, podemos modelar conectores mediante relaciones (binarias, en principio). En nuestro caso, podemos requerir conectar productores con consumidores. Luego, un sistema, de productores y consumidores, constaría, en principio, de un conjunto de productores, un conjunto de consumidores, y una relación binaria entre éstos:

```
sig System {
    ccs: set Consumer,
    pps: set Producer,
    conn: pps -> ccs
}
```

2.2.1. Modelando Operaciones del Sistema

Ahora que tenemos el sistema definido, podemos utilizar funciones en Alloy para modelar operaciones del sistema. A diferencia de otras operaciones, como por ejemplo las de productores y consumidores, podemos definir las operaciones del sistema de manera *incremental*, a partir de las definiciones que ya tenemos (esto puede notarse en la definición de la función syssend, descripta más abajo). Podemos definir operaciones de reconfiguración fácilmente, es decir, operaciones que modifican la arquitectura del sistema, agregando productores, por ejemplo:

```
fun sysnewprod (s, s' : System, p: Producer ) { s'.pps = s.pps + p }
```

o conectando productores y consumidores desconectados:

```
fun connect(s,s': System, p: Producer, c: Consumer) {
    p in s.pps && c in s.ccs && s'.pps = s.pps && s'.ccs = s.ccs
    && s'.conn = s.conn ++ (p -> c)
}
```

2.3. Funciones Asociadas a Conectores

En la definición de la signatura System, definimos conectores mediante una relación entre productores y consumidores. Pero esto no modela la interacción entre las instancias. Podemos definir la interacción, ligada a la existencia de un conector, mediante funciones de System:

```
fun syssend(s, s' : System, p,p':Producer, c,c':Consumer, d: Data ) {
    p in s.pps && c in s.ccs &&  s'.conn = s.conn && (p -> c) in s.conn
    && send(p,p',d) && receive(c,c',d) && p.n = p'.n && c.n = c'.n
    && p' in s'.pps && c' in s'.ccs
}
```

Nótese que, como parte de la precondición de esta función, tenemos que (p -> c) tiene que pertenecer a s.conn.

2.4. Restricciones de Integridad

Se pueden imponer restricciones de integridad al modelo de una arquitectura de software mediante el uso de *hechos* en Alloy. Estas restricciones, debido a limitaciones de Alloy, pueden referirse, en principio, sólo a restricciones estructurales (estáticas), pero no pueden referirse a ejecuciones [Frias et al. 2005].

A modo de ejemplo, consideremos la restricción de que cada productor puede estar conectado con a lo sumo un consumidor. Podemos decir esto de la siguiente manera:

```
fact FunctionalityConn { all s: System | all p1,p2 : s.pps | p1 = p2 => p1.conn = p2.conn }
```

Podemos decir también que, no existen en un sistema dos productores distintos con el mismo nombre:

```
fact NameUnicity {
    all s: System |
    !(some x,y : Producer | x in s.pps && y in s.pps && !(x=y) && (x.n = y.n))
}
```

2.5. Propiedades a Verificar

Las propiedades a verificar pueden modelarse en Alloy usando *aserciones*. También podemos usar el Alloy Analyzer para validar la especificación, es decir, para comprobar, al menos para modelos con dominios acotados, que las aserciones son consecuencia de los hechos y de las definiciones de los dominios. A modo de ejemplo, podríamos intentar validar el hecho (falso) de que no pueden existir productores y consumidores con el mismo nombre:

```
assert { all s: System | all p : s.pps | all c : s.ccs |  !(s.n = c.n) }
```

3. Limitaciones de Alloy

Como se observa en [Frias et al. 2003][Frias et al. 2005], Alloy no es apropiado para la validación de propiedades acerca de trazas de ejecución de sistemas. Estas propiedades son de fundamental importancia, especialmente en arquitecturas dinámicas. Si bien Alloy es limitado en este sentido, en

[Jackson et al. 2001] se presenta una técnica para validar, mediante el Alloy Analyzer, propiedades de trazas. Estamos intentando, actualmente, modelar propiedades de ejecuciones en arquitecturas reconfigurables usando esta técnica.

4. Conclusiones y Trabajos Futuros

Hemos desarrollado un caso de estudio simple para analizar el modelado de arquitecturas de software en Alloy. Presentamos el caso de estudio, la caracterización de las construcciones esenciales y algunas ideas sobre cómo modelar dinamismo usando Alloy. Actualmente, estudiamos formas de representar en Alloy propiedades más complejas, como aquellas relacionadas a dinamismo, en la reconfiguración de arquitecturas. Para tal fin, estamos empleando la técnica presentada en [Jackson et al. 2001], y estamos estudiando en paralelo la alternativa presentada en [Frias et al. 2005]. Por otra parte, dado que el Alloy Analyzer sólo puede usarse para validar propiedades (en lugar de verificarlas), planeamos estudiar lo presentado en [Frias et al. 2004] para realizar verificaciones de propiedades mediante deducción.

Agradecimientos

Las líneas de trabajo presentadas aquí se realizan bajo la dirección de Nazareno Aguirre, en el marco del proyecto *"Tratamiento Formal y Automatizable de Arquitecturas de Software, su Reconfiguración y Evolución"* (UNRC, dirigido por Gabriel Baum, y actualmente en evaluación). Nazareno Aguirre ha colaborado en el desarrollo del caso de estudio presentado, y ha contribuido con numerosas sugerencias y críticas en la redacción de este artículo.

Referencias

[Jackson 2001] D. Jackson, *Lightweight Formal Methods*, en Proceedings de International Symposium of Formal Methods Europe FME 2001, Berlín, Alemania, Lecture Notes in Computer Science, Springer-Verlag, 2001.

[Jackson et al. 2001] D. Jackson, I. Shlyakhter y M. Sridharan, *A Micromodularity Mechanism*, en Proceedings de 8th European Software Engineering Conference en conjunto con 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering ESEC/FSE 2001, Viena, Austria, 2001.

[Jackson 2002] D. Jackson, *Alloy: a lightweight object modelling notation*, en ACM Transactions on Software Engineering and Methodology (ACM TOSEM), Vol. 11, Nro. 2, 2002.

[Garlan & Shaw 1993] D. Garlan y M. Shaw, An Introduction to Software Architecture, Advances in Software Engineering and Knowledge Engineering, Vol. 1. World Scientific Publishing Co., 1993.

[Frias et al. 2003] M. Frias, C. López Pombo, G. Baum, N. Aguirre y T. Maibaum, *Taking Alloy to the Movies*, en Proceedings de International Symposium on Formal Methods FM 2003, Pisa, Italia, Lecture Notes in Computer Science, Springer-Verlag, 2003.

[Frias et al. 2004] M. Frias, C. López Pombo y N. Aguirre, *An Equational Calculus for Alloy*, en Proceedings de International Conference on Formal Engineering Methods ICFEM 2004, Seattle, Estados Unidos, Lecture Notes in Computer Science, Springer-Verlag, 2004.

[Frias et al. 2005] M. Frias, J. Galeotti, C. López Pombo y N. Aguirre, *DynAlloy: Upgrading Alloy with Actions*, a publicarse en Proceedings de International Conference on Software Engineering ICSE 2005, St. Louis, Estados Unidos, ACM Press, 2005.