Analizando el uso de (Dyn)Alloy como Herramienta Educativa

César Cornejo[1], Mariano Politano[1], Fernando Raverta[1], Sonia Permigiani[1], Pablo Ponzio[1,2], Germán Regis[1], and Nazareno Aguirre[1,2]

[1] Dpto. de Computación, FCEFQyN, Universidad Nac. de Río Cuarto, Argentina
[2] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina

Resumen Los significativos avances en técnicas automáticas de análisis, como model checking, constraint solving y computación evolutiva, adquieren constantemente mayor relevancia en actividades complejas de construcción de software, y son exitosamente utilizadas para generar tests automáticamente, refinar requisitos de software, verificar diseños, y descubrir errores de programas. Sin embargo, el uso de tales técnicas para asistir en actividades de enseñanza-aprendizaje es muy escaso. En este trabajo, analizamos el uso del lenguaje formal Alloy y su extensión DynAlloy, que incorpora acciones y programas, como herramienta educativa, para la asistencia y el soporte de tareas de elaboración y comprensión de especificaciones, y otras actividades en las cuales las especificaciones, entendidas como descripciones declarativas de software, son centrales. Mostraremos cómo el análisis automático subyacente a estos lenguajes puede emplearse efectivamente para dar soporte en la depuración de especificaciones, permite introducir naturalmente conceptos que suelen escapar a cursos introductorios, como el no determinismo, y puede facilitar el uso de abstracciones de datos adecuadas en cursos introductorios.

1. Introducción

En las últimas décadas, los enormes avances en técnicas automáticas de análisis han comenzado a dar lugar a novedosas aplicaciones en diversas áreas de la Ingeniería de Software. En efecto, algunas tecnologías de análisis automático como el *model checking* [5], constraint solving (SAT, SMT) [4] y la computación evolutiva [10], se aplican con éxito a tareas como la operacionalización de requisitos, la verificación de diseños de software, la generación automática de tests, el descubrimiento de defectos en software, entre otras. Sin embargo, la adopción de estas poderosas tecnologías como herramientas de apoyo a actividades educativas en Ciencias de la Computación es escasa. A pesar del limitado uso de estas técnicas para tareas vinculadas a la educación en Informática, es significativo el potencial que las mismas tienen, en particular como formas de automatizar, o semi automatizar, actividades de soporte a los estudiantes. Un ejemplo singular es el PEX [13], una herramienta de generación automática de tests basada en ejecución simbólica dinámica. PEX ha sido utilizada con éxito para realizar debugging automático y así guiar estudiantes en la construcción de programas,

mediante la contrastación de las soluciones de los estudiantes con implementaciones correctas (secretas) para detectar discrepancias de comportamiento y proveer las mismas como feedback a los usuarios [14]. Otro ejemplo similar, basado en tests como feedback para estudiantes en la resolución de ejercicios, lo constituye la herramienta `Codeboard`[1], que permite cargar ejercicios de programación (en varios lenguajes), proveer tests ocultos para los mismos, y ejecutar los tests sobre las resoluciones de los estudiantes, para proveer feedback previo a la entrega de los trabajos.

Mientras PEX y CodeBoard se concentran en proveer feedback automáticamente, pero vinculado a la construcción de programas, en este artículo analizamos un problema igualmente relevante, el de proveer feedback automático vinculado a la construcción de *especificaciones*. Es ampliamente aceptado en el ámbito de la Ingeniería de Software y las Ciencias de la Computación que una correcta, precisa y completa comprensión del problema a resolver, y una adecuada captura del mismo, son fundamentales para el desarrollo efectivo y eficiente de software. Más aún, en contextos de programación *in the small*, suele contarse con lenguajes formales para capturar "el problema", idealmente previo al desarrollo del software en sí mismo. De hecho, no son pocas las asignaturas de programación vistas en carreras de Computación del país y del exterior que utilizan tales lenguajes como parte de la instrucción fundamental de la disciplina. Sin embargo, la asistencia y soporte a los estudiantes en relación a la ayuda en construcción de especificaciones es casi enteramente manual, dependiendo de contar, en persona, con un instructor o asistente que funcione como "oráculo", es decir, que evalúe las soluciones de estudiantes y guíe a los mismos en la resolución de problemas con sus especificaciones. Esto deriva en frecuentes problemas relacionados a las dificultades tanto a la hora de escribir especificaciones como de comprenderlas, observados por docentes y estudiantes. Las técnicas avanzadas de constraint solving, y en particular las basadas en satisfactibilidad booleana (SAT solving) y satisfactibilidad módulo teorías (SMT solving) tienen el potencial de contribuir a paliar este problema, como ilustraremos y discutiremos en este artículo. Más precisamente, nos concentraremos en el uso del lenguaje relacional Alloy, y su extensión DynAlloy, que incorpora acciones y programas, y mostraremos cómo estos lenguajes y las herramientas de análisis (basadas en SAT solving) asociadas a los mismos pueden ser usadas para asistir en tareas como las descriptas previamente.

2. DynAlloy y su Herramienta de Análisis

En esta sección describimos Alloy y DynAlloy, los dos lenguajes que analizamos y proponemos para el análisis de especificaciones como parte de actividades educativas. Alloy [7] es un lenguaje de especificación, de los denominados *basados en modelos*. La sintaxis de Alloy es simple e intuitiva para los desarrolladores acostumbrados a lenguajes orientados a objetos, con una semántica relacional

[1] `http://codeboard.io`

simple. Sus modelos se componen de dominios de datos, propiedades y operaciones entre estos dominios. Una característica que destaca a Alloy sobre otros lenguajes formales de especificación como Z [9] y B [1] radica en su principal objetivo centrado en brindar análisis automático. Alloy Analyzer es la herramienta soporte del lenguaje que permite simular modelos y buscar contraejemplos de propiedades, en modelos acotados. El análisis de modelos Alloy se basa en la reducción de los mismos a fórmulas proposicionales cuyas satisfactibilidad se verifica mediante la utilización de SAT-solvers [6, 12]. DynAlloy [8] extiende Alloy con estado, acciones que permiten alterar el mismo, programas compuestos de acciones, y aserciones de corrección parcial que acompañan tales programas. Si uno especifica dominios de datos en Alloy, y programas sobre los mismos usando DynAlloy, es posible simular la ejecución de tales programas, y buscar contraejemplos en modelos acotados, para aserciones de corrección parcial (pre y post condiciones, etc.).

Los dominios de datos en (DynAlloy) se construyen a través de *signaturas* (el único tipo predefinido es `Int`). Suponiendo que queremos modelar secuencias de elementos es cierto tipo, podemos definir el dominio de los elementos (sobre el cual definiremos secuencias) utilizando una signatura simple, de la siguiente manera:

```
sig Elem { }
```

Las signaturas pueden también tener estructura, definida en términos de *campos*. Por ejemplo, podemos modelar secuencias de elementos de tipo `Elem` usando la siguiente signatura:

```
sig Seq {
   idx: set Int,
   length: Int,
   at: idx -> one Elem }
```

Una secuencia tiene un conjunto de índices, una longitud, y una función que para cada índice, indica cuál es el elemento almacenado en el mismo. Nótese cómo podemos definir campos de tipos relacionales (es una de las potencias del lenguaje). Ahora, si solicitamos a la herramienta que nos brinde un ejemplo de una secuencia de caracteres, seguramente nos desilusionaremos un poco, porque aún no hemos capturado en el modelo algunos elementos importantes sobre las secuencias. Debemos imponer, por ejemplo, que el tamaño de una secuencia es siempre mayor o igual a 0, el conjunto de indices es el rango de 0 a la longitud de la secuencia menos 1, etc. Estas restricciones son especificadas en Alloy mediante *hechos*, que restringen modelos con fórmulas que se imponen como válidas, como axiomas. Por ejemplo, tenemos para nuestro modelo:

```
fact SpecSeq { (all s: Seq | s.length >= 0 ) &&
   (all s: Seq, i: Int | i >= 0 && i < s.length => some s.at[i]) && ... }
```

Además de hechos, Alloy soporta *predicados* y *aserciones*. Ambas modelan propiedades; los predicados son fórmulas que capturan modelos con características particulares. Por ejemplo, podemos caracterizar *palíndromos* con un predicado como el siguiente:

```
pred esPalindromo(s: Seq) {
   all i: Int | i>=0 and i<s.length => s.at[i] = s.at[s.length - (i+1)] }
```

Las aserciones capturan propiedades a verificar, es decir fórmulas que uno espera que sean verdaderas en todos los modelos que satisfagan los hechos. Por ejemplo, la siguiente aserción:

```
assert idxDefinido {
   all s: Seq, i: Int | i>=0 and i<s.length => some s.at[i] }
```

Esta aserción indica que en toda secuencia, si un índice es menor que la longitud de la secuencia y mayor o igual a cero, entonces la secuencia está "definida" en ese índice. La diferencia principal entre predicados y aserciones está en análisis. Mientras Alloy Analyzer buscará *instancias* de predicados, la herramienta buscará *contraejemplos* de aserciones.

El modelo anterior es *estático*, en el sentido que no captura estado, ni permite cambiar estados. Si bien estos elementos se pueden codificar en Alloy (como predicados que vinculan estados previos y posteriores a la ejecución de operaciones, y modelos ad hoc de trazas de ejecución), DynAlloy provee una forma más conveniente, eficiente y clara de hacerlo, a través de *acciones* y *programas* [8]. La siguiente acción, por ejemplo, agrega un elemento al final de una secuencia:

```
act append[s: Seq, e: Elem] {
   pre { }
   post { s'.length = s.length+1 and s'.at[s'.length-1] = e and
           all i: Int | i>=0 && i<s'.length-1 => s'.at[i] = s.at[i] } }
```

Las acciones atómicas se *definen* por su pre y postcondición (es decir, su definición es un hecho). En cambio las pre y postcondiciones para programas son *aserciones*, es decir se verificarán (en escenarios acotados). DynAlloy provee tres operadores que permiten componer acciones atómicas: composición secuencial (;), elección no determinista (+) e iteración (*). Además de los operadores de composición, Dynalloy ofrece la posibilidad de incorporar *tests* (asunciones) en la especificación de los programas y azúcar sintáctico que convierten el proceso de especificación en una tarea de programación en un lenguaje imperativo. La verificación de estos programas se realiza mediante la definición de pre y postcondiciones de los mismos. A modo de ejemplo de programa más complejo anotado por su especificación, consideremos la búsqueda secuencial:

```
assertCorrectness busquedaSecuencialCorrecta {
   pre { s.length >= 0 }
   program busquedaSecuencial[s: Seq, elem: Elem, out: Boolean] {
      var i: Int;
      out := False;
      i := 0;
      while (i<s.length and not out) {
         if (s.at[i]=elem) { out := True; }
         i := i+1; } }
   post { out <=> (some x: Int | x>=0 and x<s.length and s.at[x]=elem) } }
```

DynAlloy Analyzer permite realizar diversos análisis sobre especificaciones (Dyn)Alloy. Además de buscar instancias de predicados y contraejemplos de

aserciones, permite simular ejecuciones de programas DynAlloy, y verificar aserciones de corrección parcial. La verificación se realiza, al igual que en el caso de Alloy, mediante una reducción a SAT solving, y para escenarios acotados (es decir que, para todo escenario acotado que satisfaga la precondición, al ejecutar el programa deriva necesariamente en un estado que satisface la postcondición). El comando de análisis recibe una especificación de las cotas. Por ejemplo, el comando:

```
check busquedaSecuencialCorrecta for 1 Seq, 5 Elem, 5 lurs
```

verificará que efectivamente el programa cumple con su especificación en todas las ejecuciones de hasta 5 iteraciones (5 lurs), partiendo de estados compuestos por 1 secuencia (1 Seq) y hasta 5 elementos (5 Elem). Esta comprobación podría evidenciar problemas en la especificación y/o el programa. Suponiendo que esta comprobación se satisface, no constituye una *prueba*, pues sólo garantiza la corrección del programa en un número de escenarios acotados. La Figura 1 muestra una captura de pantalla de DynAlloy Analyzer, con el programa correspondiente a la búsqueda secuencial. Los contraejemplos se pueden visualizar de diferentes maneras, como ilustraremos más adelante.

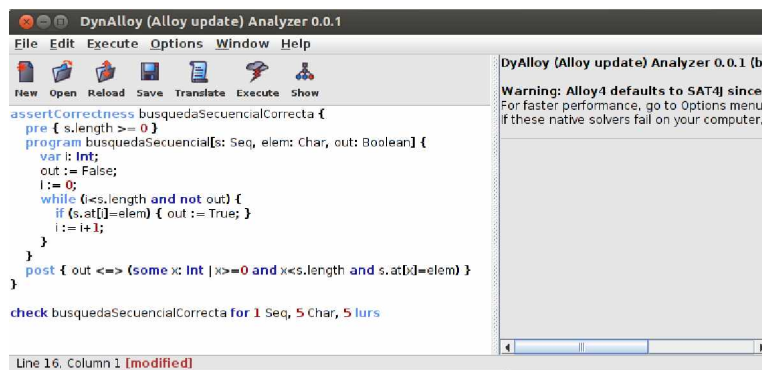

Figura 1. DynAlloy Analyzer: Interfaz gráfica con un ejemplo simple

3. Análisis del uso Educativo de (Dyn)Alloy

En esta sección discutiremos una variedad de tareas en las cuales herramientas como las asociadas a Alloy y DynAlloy pueden contribuir en el desarrollo de especificaciones.

3.1. Asistencia en la Elaboración de Especificaciones

Especificar formalmente lo que se intenta resolver mediante un programa es una tarea compleja. Demanda no sólo comprender el lenguaje de especificación y la semántica de sus construcciones, sino el uso correcto de sus construcciones, y la captura correcta mediante las mismas del problema en cuestión. Es frecuente encontrarse con dificultades vinculadas a la construcción de especificaciones inconsistentes (a veces sutilmente), casos no considerados, y demás problemas. En

estas circunstancias, y por tratarse las especificaciones de objetos formales no ejecutables, los estudiantes tienen pocas herramientas para depurar sus especificaciones más que ponerlas a consideración de docentes. Constraint solving puede usarse para ayudar en estas circunstancias, como herramienta de comprobación automática.

Consideremos el siguiente ejemplo. Supongamos que se desea especificar una operación que, dada una secuencia s, determina si la misma es o no un *palíndromo*. Esta operación puede ser especificada como mostramos en la sección anterior. Este ejemplo, de hecho, fue tomado de un examen. Un error común cometido con frecuencia en la resolución de este ejercicio en el mencionado examen se dio en el manejo de índices. Más precisamente, muchos estudiantes escribieron la siguiente especificación:

```
pred esPalindromoEst(s: Seq) {
   all i: Int | i>=0 and i<s.length => s.at[i] = s.at[s.length - i] }
```

Suponiendo que los docentes proveen la especificación correcta dada precedentemente, Alloy podría utilizarse como asistente automático, mediante el chequeo de la siguiente aserción:

```
assert esPalindromoEstEsCorrecto {
    all s: Seq | esPalindromo[s] iff esPalindromoEst[s] }
```

Alloy Analyzer producirá en este caso contraejemplos, como el que se muestra en la Fig. 2, que el estudiante puede aprovechar para depurar su especificación. En el mismo se muestra que la especificación no es equivalente para la secuencia de un elemento. Efectivamente, la especificación provista por el estudiante intenta evaluar entre la posición 0 y la posición 1 de la secuencia, con esta última inexistente. Con este razonamiento el estudiante podría corregir su especificación.

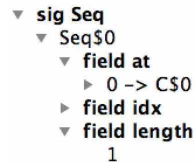

Figura 2. Contra-ejemplo generado por Alloy Analyzer

3.2. Asistencia en la Derivación/Verificación de Programas

Verificar que un programa satisface su especificación es indecidible, y por lo tanto, escapa a lo que puede resolverse automáticamente. Sin embargo, podemos automatizar parte de las tareas en este proceso, y como mínimo brindar ayuda a los estudiantes mediante herramientas, en pasos de estos procesos que corresponden a los denominados "eureka" (i.e., que requieren de la intervención por parte del usuario, para la provisión de definiciones adecuadas). Un ejemplo de esto es la verificación de programas iterativos, que demanda la provisión de *invariantes* adecuados. Consideremos, por ejemplo, el siguiente caso. Supongamos que contamos con el programa anotado para la búsqueda secuencial, provisto en la sección anterior. La verificación del mismo requiere la provisión de un *invariante* adecuado para el ciclo. Podemos sin embargo realizar algunas tareas automáticamente

usando DynAlloy Analyzer, incluso previo a la construcción del invariante. La primera es comprobar, para ejecuciones acotadas en su número de iteraciones y para conjuntos de estados acotados, si efectivamente el programa cumple con su especificación. Por ejemplo, el comando:

```
check busquedaSecuencialCorrecta for 1 Seq, 5 Elem, 5 lurs
```

verifica, como describimos anteriormente, que el programa satisfaga su especificación en los escenarios y ejecuciones acotados según se indica en el mismo. Sin embargo, esto no constituye una prueba, dado que podrían existir ejecuciones más extensas (con más iteraciones) o sobre estados más complejos que violen la propiedad. Una verificación completa del mismo demanda una prueba, que requiere un invariante para el ciclo del programa. Un posible invariante, que debe ser provisto por el usuario, es el siguiente:

```
inv: out <=> (some x: Int | x>=0 and x<i and s.at[x]=elem)
```

Ahora que tenemos una propuesta de invariante, podemos realizar nuevos análisis. Una primera comprobación consiste en chequear que, para un número acotado de escenarios, esta fórmula se comporta como un invariante, es decir, es válida antes de comenzar a iterar, y se preserva luego de cada iteración. Esto se consigue en DynAlloy simplemente agregando:

```
assert inv;
```

como sentencia inmediatamente previa al ciclo, y como última sentencia del cuerpo del mismo. Nuevamente, los contraejemplos que pudieran surgir de esta etapa ayudarán al estudiante a depurar y mejorar su invariante. Finalmente, si esta nueva etapa es superada, debemos comprobar que el invariante es *inductivo*. Si bien esta tarea se realiza siguiendo reglas de inferencia y el cálculo de *weakest precondition*, DynAlloy Analyzer puede emplearse para encontrar contraejemplos automáticamente, y depurar el invariante previo a encarar la prueba. Esto se consigue verificando las siguientes aserciones de corrección parcial:

```
assertCorrectness proofOblig1 {
   pre { s.length >= 0 }
   program init[s: Seq, elem: Elem, out: Boolean] {
      var i: Int;
      out := False;
      i := 0; }
   post { inv }
}
assertCorrectness proofOblig2 {
   pre { inv and i<s.length and not out }
   program loop[s: Seq, elem: Elem, out: Boolean, i: Int] {
      if (s.at[i]=elem) { out := True; }
      i := i+1; }
   post { inv }
}
assert proofOblig3 {
   all s: Seq, elem: Elem, out: Boolean, i: Int |
   inv and not (i<s.length and not out) =>
   (out <=> (some x: Int | x>=0 and x<i and s.at[x]=elem)) }
```

Para nuestro ejemplo, `proofOblig1` se cumple, mientras que `proofOblig2` y `proofOblig3` no se cumplen; el invariante no es inductivo, dado que debe fortalecerse con el rango de la variable `i`.

3.3. No Determinismo y Abstracciones de Datos

En asignaturas sobre metodologías de la programación (programación estructurada, verificación de programas *in the small*, etc.), suele emplearse pseudocódigo de algún tipo, para facilitar la comunicación de diseños algorítmicos, y no estar atado a un lenguaje de programación particular. Se usa desde pseudocódigo informal, hasta lenguajes del estilo del lenguaje de comandos con guardas (GCL) de Dijkstra. Si bien estos lenguajes ofrecen facilidades como no determinismo (en IF o DO, en el caso de GCL, por ejemplo) o tipos de datos más abstractos que los disponibles en lenguajes de programación, el no contar con herramientas de software que soporten los lenguajes limitan su utilización. De hecho, si miramos bibliografía que utilice lenguajes como GCL (e.g., [2,3]), rara vez se utiliza no determinismo. En cuanto al uso de abstracciones de datos, es común observar un uso intensivo de cadenas de caracteres y de tipos numéricos en cursos introductorios. La utilización de lenguajes y herramientas de análisis como DynAlloy pueden ayudar a resolver parcialmente este problema. Consideremos el caso de no determinismo. Para esto pensemos en el acertijo conocido como el *Problema de las Jarras de Agua*: dadas dos jarras de agua de 3 y 4 litros respectivamente, sin marcas de nivel de contenido, podemos conseguir tener exactamente 2 litros en una de ellas si las acciones que podemos hacer son pasar el contenido de una a otra, vaciar o llenar cualquiera de las jarras? En DynAlloy podemos capturar este problema naturalmente con elección no determinista, e incluso ilustrar la definición ad hoc de tipos de datos de manera sumamente conveniente, como mostramos a continuación:

```
sig Estado {
   jarraA: Int,
   jarraB: Int }

fact{all e: Estado | gte[e.jarraA,0] and gte[e.jarraB,0]
   and lte[e.jarraA,4] and lte[e.jarraB,3] }

action vaciarA[e:Estado]{
   pre{gt[e.jarraA,0]}
   post{eq[e'.jarraA,0] and eq[e.jarraB,e'.jarraB]} }
action llenarA[e:Estado]{
   pre{lt[e.jarraA,4]}
   post{eq[e'.jarraA,4] and eq[e.jarraB,e'.jarraB]} }
...
pred jarrasVacias[e: Estado] {
   eq[e.jarraA, 0] and eq[e.jarraB, 0] }
pred final[e: Estado] {
   eq[e.jarraA, 2] or eq[e.jarraB, 2] }

program acertijoJarras[e:Estado]{
   [jarrasVacias[e]]?;
```

```
   repeat{
      vaciarA[e] + vaciarB[e] + llenarA[e] +
      llenarB[e] + transferirDeA[e] + transferirDeB[e] } ;
   [final[e]]?
}
run acertijoJarras for 7 but 5 int lurs exactly
```

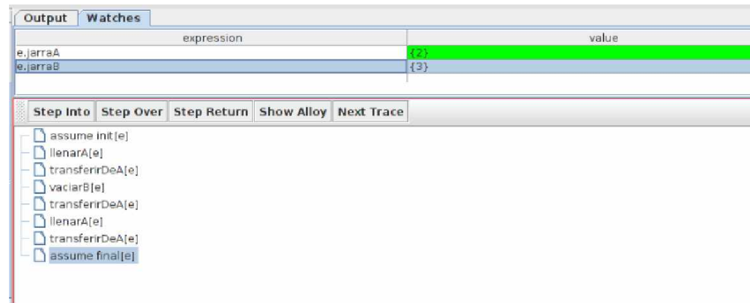

Figura 3. Traza de Ejecucion del problema de las Jarras de Agua

La Figura 3 ilustra la herramienta en ejecución, mostrando una solución al acertijo.

4. Discusión, Conclusiones y Trabajos Futuros

La Ingeniería de Software Educacional [14] propone el uso de tecnología de análisis con fines educativos. Las ventajas de este uso son variadas, especialmente en momentos en los cuales se empieza a observar un uso regional más intensivo de plataformas de educación a distancia. En este tipo de contextos, contar con herramientas de *tutoring* y asistencia automática a los estudiantes (que podría derivar incluso en corrección automática de ejercicios o similares) son sumamente útiles. Mientras existen casos exitosos de tales tecnologías, como PEX [14] y CodeBoard, los mismos se concentran en código. En este trabajo, nos centramos en tareas vinculadas a la construcción de *especificaciones*, centrales en el desarrollo de software, y mostramos cómo el constraint solving, específicamente SAT solving, puede ayudar en tareas como la asistencia a la construcción de especificaciones, y la evaluación preliminar de invariantes. Más aún, lenguajes como Alloy y DynAlloy, estudiados en este artículo, brindan flexibilidad en abstracciones de datos (conjuntos, relaciones built-in) y construcciones de programación (elección no determinista, iteración acotada), que pueden usarse para incorporar, por ejemplo, resolución de acertijos como parte de la enseñanza de la algoítmica, con soporte computacional (el uso de acertijos en la enseñanza de la programación es propuesto en varios trabajos, notablemente [11]). Sin embargo, nuestra propuesta no viene sin limitaciones. Incorporar lenguajes como Alloy y DynAlloy tienen sus dificultades, incluyendo el uso de un lenguaje adicional en el proceso de enseñanza aprendizaje. En nuestra opinión, su correcta adopción demanda la

construcción de interfaces que permitan a los estudiantes incorporar la tecnología sin necesidad de aprender nuevos lenguajes y abstracciones; la discrepancia entre las construcciones de (Dyn)Alloy y las construcciones de metodologías de construcción de programas como las utilizadas en [2, 3] (estas últimas centradas en el uso de expresiones cuantificadas con diferentes tipos de “cuantificación”, numéricas por ejemplo) hacen al desarrollo de interfaces adecuadas una tarea a la vez indispensable y compleja.

Este artículo presenta un análisis de cuánto tienen para aportar herramientas de análisis basado en constraint solving como Alloy y DynAlloy. En cuanto a trabajo futuro, planeamos realizar estudios de campo que nos ayuden a construir interfaces adecuadas para la tecnología estudiada, incorporar las herramientas a asignaturas de enseñanza de la programación y evaluar el impacto de las mismas en los procesos de enseñanza-aprendizaje.

Referencias

1. J. Abrial, *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, 2005.
2. R. Backhouse, *Program construction: calculating implementations from specifications*, Wiley, 2003.
3. D. Barsotti, J. O. Blanco, S. Smith, *Clculo de Programas*, Universidad Nacional de Crdoba, 2008.
4. A. Biere, M. Heule, H. van Maaren y T. Walsh, *Handbook of Satisfiability*: Volumen 185, Frontiers in Artificial Intelligence and Applications. IOS Press, Amsterdam, Holanda. 2009.
5. E. Clarke, O. Grumberg y D. Peled, *Model Checking*, MIT Press, 2000.
6. N. Eén, N. Sörensson, *An Extensible SAT-solver*, en Proc. de 6th International Conference SAT 2003, LNCS 2919, Springer, 2004.
7. D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, The MIT Press, 2006.
8. M. Frias, J.P. Galeotti, C. López Pombo y N. Aguirre, *DynAlloy: upgrading alloy with actions*, en Proc. de International Conference on Software Engineering ICSE 2005, St. Louis, USA. ACM Press, 2005.
9. J. Jacky, *The Way of Z: Practical Programming with Formal Methods*, Cambridge University Press, 1997.
10. J. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.
11. A. Levitin, M. Papalaskari, *Using puzzles in teaching algorithms*, en Proc. 33rd Technical Symposium on Computer Science Education SIGCSE 2002, USA, 2002
12. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik,*Chaff: Engineering an Efficient SAT Solver*, en Proc. de 38th Design Automation Conference DAC 2001, ACM, 2001.
13. N. Tillmann y J. de Halleux, *PEX: White Box Test Generation for .NET*, en Proc. de 2nd. International Conference on Tests and Proofs TAP 2008, LNCS 4966, Springer, 2008.
14. T. Xie, N. Tillmann y J. de Halleux, *Educational software engineering: where software engineering, education, and gaming meet*, en Proc. de 3rd International Workshop on Games and Software Engineering: Engineering Computer Games to Enable Positive, Progressive Change, IEEE Press, 2013.