ISBN 978-987-4417-90-9

Herramientas de model finding para asistir en la Construcción de Especificaciones Formales

Sonia Permigiani[1], María Marta Novaira[1], Gastón Scilingo[1], and Marcelo Arroyo[1]

Dpto. de Computación, FCEFQyN, Universidad Nac. de Río Cuarto, Argentina
{spermigiani,mnovaira,gscilingo,marcelo.arroyo}@dc.exa.unrc.edu.ar

Resumen Este trabajo plantea el desafío técnico de asistir en la construcción de especificaciones en el contexto de la enseñanza de la programación, en materias introductorias, a través de la provisión automática de información de análisis, basada en constraint solving relacional. Se presentan a través de ejemplos concretos, por un lado, la traducción de especificaciones que involucran expresiones cuantificadas al lenguaje Alloy, y por otro la definición de consultas de análisis que, mediante las herramientas Alloy y DynAlloy, brindan *feedback* automático a los estudiantes. Se discuten además algunas dificultades técnicas que se presentan al construir modelos de especificaciones con dominios numéricos, para su utilización en herramientas de model finding.

1. Introducción

Los sustanciales avances en técnicas de análisis automático de software han dado lugar a poderosas herramientas que comienzan a ser usadas en contextos industriales reales, explotando mecanismos como la generación automática de tests, el constraint solving y la ejecución simbólica. Estas herramientas permiten encontrar defectos sutiles en software, y pueden brindar *feedback* de gran utilidad a los desarrolladores. No es sorprendente entonces que también hayan encontrado aplicaciones en contextos educativos, donde pueden emplearse para brindar feedback automático a estudiantes, en sus procesos de construcción de programas. Herramientas como KeY (`https://www.key-project.org/`), Pex [16], y herramientas como Infer en su versión accesible desde CodeBoard (`http://codeboard.io`), son ejemplos de herramientas con sofisticados análisis, orientadas a contextos de enseñanza de la programación. Asimismo, otras técnicas menos elaboradas, como la provisión *manual* de tests, suelen usarse para contrastar soluciones de estudiantes contra las expectativas de los docentes (véase su uso, por ejemplo, en plataformas como HackerRank).

En todos los ejemplos antes mencionados el énfasis está en la construcción de *programas*, y su respectivo análisis. No son pocos, sin embargo, los cursos introductorios a la programación que ponen especial énfasis en las *especificaciones*. Enfoques introductorios a la programación como los presentados en [2,3] son sólo algunos ejemplos de estrategias de enseñanza que se centran en especificaciones.

En éstas y otras metodologías relacionadas, el proceso de *construcción* de una especificación es muy difícil, y los estudiantes suelen contar con prácticamente ningún tipo de asistencia automática, a través de herramientas.

Nuestro objetivo es justamente cubrir esta falencia. En [18] comenzamos a delinear una metodología basada en el uso de *model finders*, como los subyacentes a lenguajes del estilo de Alloy y DynAlloy, para asistir automáticamente a estudiantes en el proceso de construcción de especificaciones, por ejemplo, proveyendo escenarios que sus especificaciones candidatas no capturan, o capturan erróneamente. En este trabajo, mostramos pasos concretos en esta dirección. Tomando como punto de partida el lenguaje de expresiones cuantificadas introducido en [3], mostramos una caracterización indirecta del mismo en el lenguaje formal Alloy, con el objetivo de aprovechar el model finder Alloy Analyzer, asociado a este último. La traducción es indirecta porque, como mostramos en el artículo, requiere pasar por el lenguaje DynAlloy, una extensión dinámica de Alloy, con el objetivo de poder capturar fielmente algunas expresiones del lenguaje origen. Mostramos las dificultades que se dan en el análisis del lenguaje origen a través de model finders, en especial debido al fuerte énfasis del mismo en el uso de aritmética. Finalmente, mostramos algunos ejemplos de la herramienta resultante, con casos específicos de problemas de especificación, y un análisis del feedback provisto por nuestra herramienta.

2. Alloy y DynAlloy

Alloy [8] es un lenguaje de especificación de primer orden basado en el uso de relaciones. Está orientado a la especificación de propiedades estructurales de sistemas e intenta resolver la complejidad en la expresión de algunas propiedades comunes en lógica de primer orden utilizando operadores relacionales, definidos en la lóica relacional, el formalismo subyacente a Alloy. Las *signaturas* permiten describir dominios y sus estructuras, siguiendo el estilo de Z [12]. De manera similar, se pueden especificar operaciones mediante expresiones que relacionan el estado previo a la aplicación de una operación y el estado posterior correspondiente. Para ilustrar la definición de operaciones en Alloy, considere, por ejemplo,una operación que especifica la escritura de un valor en una dirección de memoria:

```
pred write(m, m': Memory, d: Data, a: Addr) {
    m'.map = m.map ++ (a -> d)
}
```

La operación *write*, descrita a través de un predicado, transforma una memoria, sobre-escribiendo el valor asociado a una dirección a con un dato d. Este predicado, como los esquemas para operaciones en Z, no está directamente asociado a ninguna signatura. Las variables primadas son usadas para denotar estados posteriores a la ejecución de la operación, aunque esto es simplemente una convención, ya que no se refleja en la semántica.

DynAlloy [20] es una extensión del lenguaje Alloy, que agrega nuevas características que permiten expresar mejor los cambios de estado. Unas de la deficiencias más importante de Alloy es que carece de una forma sencilla para expresar propiedades dinámicas de sistemas. DynAlloy, inspirado en la lógica dinámica, consigue especificar propiedades de trazas de manera más apropiada. También se caracteriza por la sencillez y rapidez de poder traducir modelos DynAlloy a Alloy sin requerir intervención por parte del usuario, para luego realizar validaciones utilizando el Alloy Analyzer. La sintaxis de DynAlloy extiende la de Alloy agregando una cláusula para definir programas:

```
formula ::= . . . |{formula} program {formula}
```

Con la incorporación de programas, los predicados que representaban operaciones sobre el modelo ahora se pueden definir como acciones dentro de un programa:

```
{ true }
write(m : Memory, d : Data, a : Addr)
{ m'.map = m.map ++ (a -> d) }
```

En la especificación anterior, la variable `m'` representa el valor de la variable `s` después de la ejecución de la acción `write`, y se asume que aquellas variables que no ocurren en la post-condición de forma primada, retienen su valor inicial.

3. El Lenguaje de Expresiones Cuantificadas

La comprensión precisa de un problema a resolver es un primer paso, ampliamente entendido como ineludible en la Ingeniería de Software moderna y las ciencias de la computación, en el camino para el desarrollo de software de manera efectiva. Es por esto que es importante que existan técnicas automáticas de software que den soporte al tipo de problemas que se dan al intentar capturar la especificación de un problema. Esta importancia se observa, por ejemplo, en el lugar que ocupan las técnicas basadas en lenguajes formales para especificación de programas en varias currículas de carreras de computación. En esta sección presentamos un lenguaje para especificaciones formales basado en una versión ecuacional de lógica de predicados con cuantificadores [2], utilizado en el marco de cursos de programación que introducen el concepto de especificación.

Las *expresiones cuantificadas* resultan útiles para expresar la reiteración de aplicaciones de un cierto operador sobre una expresión. Esta notación resulta muy práctica para especificar los problemas usuales que se plantean en los cursos introductórios a la programación. Consideremos un ejemplo sencillo: la suma de los n primeros números impares. Utilizando una notación matemática estándar, podemos describir el problema con la siguiente expresión:

$$\sum_{i=0}^{n-1} 2 \times i + 1 \tag{1}$$

En esta expresión distinguimos varias componentes: el operador usado (la sumatoria) se corresponde a la operación binaria suma, i es una variable con un rango de variación asociado (0 a $n-1$), y por último la expresión $(2 \times i + 1)$, que denota cuáles van a ser los términos de la sumatoria.

En el lenguaje aquí utilizado, este mecanismo de definición de expresiones es generalizado a operadores binarios asociativos y conmutativos. La noción de cuantificación aparece dada por variables formales definidas con un alcance delimitado específicamente por los paréntesis, y éstas se podrán usar para construir expresiones dentro del alcance especificado. La forma general de las especificaciones será:

$$(\oplus i : R : T) \tag{2}$$

Ésta consta de tres partes, delimitadas por ":". En la primera, el símbolo $\oplus$ representa el operador (e.g., $\vee$, $\wedge$, $+$, *Max*, *Min*, etc), donde i es la variable cuantificada (no hace falta definir el tipo de la misma). La segunda parte es el rango de la expresión cuantificada, denotado con el predicado R. Y finalmente T es una función que denota el término de la cuantificación. Utilizando esta notación, podemos, por ejemplo, escribir la expresión (1) de la siguiente manera:

$$(\sum i : 0 \leq i < n : 2 * i + 1) \tag{3}$$

El resultado de esta sumatoria depende del valor de n, una variable fuera del alcance de la expresión. Cabe destacar que una expresión cuantificada puede involucrar más de una variable ligada. Por ejemplo, si queremos expresar la propiedad que indica que una secuencia es capicúa, podríamos hacerlo a través de la siguiente especificación:

$$(\forall p, q : 0 \leq p \wedge 0 \leq q \wedge p + q = N - 1 \wedge N \leq \sharp xs : xs.p = xs.q) \tag{4}$$

donde p y q son dos variables cuantificadas y xs es una secuencia (el punto es el operador de indexación para secuencias, i.e., $xs.p$ será el elemento de xs en la posición p).

4. De Expresiones Cuantificadas a (Dyn)Alloy

En [18] se plantea la utilización de herramientas como Alloy y DynAlloy para la asistencia automática en la construcción de especificaciones, y se introducen ejemplos en los que se muestra cómo detectar errores comunes, tomados de exámenes y prácticas. El proceso de detección requiere de una "versión dorada", una especificación correcta (provista por el docente) del problema en cuestión, contra la cual contrastar la especificación candidata del estudiante, y por supuesto una *traducción* a Alloy, para poder contrastar las especificaciones automáticamente, usando Alloy Analyzer. En este trabajo estudiamos justamente esta traducción. El objetivo es poder brindar una traducción automática de expresiones cuantificadas a Alloy, preservando la semántica de las fórmulas originales. Como veremos, esta tarea implica una serie de desafíos técnicos, que intentamos sortear en el artículo.

4.1. Cuantificadores Booleanos

El lenguaje Alloy provee soporte directo para las cuantificaciones universal ($\forall$) y existencial ($\exists$), de la lógica de primer orden, a través de sus operadores `all` y `some`. Sin embargo, la traducción de las expresiones cuantificadas booleanas de un lenguaje a otro no es directa, ya que requiere el uso de propiedades lógicas que permitan pasar el predicado del *rango* al *término*. Mostramos la traducción de estas expresiones, las más simples del lenguaje, a través de ejemplos ilustrativos.

Para el cuantificador universal, el rango se reduce al *antecedente* de una implicación, que tiene al *término* como consecuente. Por ejemplo, la expresión cuantificada siguiente:

$$(\forall i : 0 \leq i < \#xs - 1 : xs.i = xs.(i + 1))$$

que indica que todos los elementos de una secuencia de enteros son iguales, se captura en Alloy de la siguiente manera[1]:

```
all i: Int | 0 <= i < #xs-1 implies xs[i]= xs[i+1]
```

Para la traducción de expresiones cuantificadas existencialmente, la traducción convierte al rango en una conjunción que acompaña al término. Por ejemplo, la expresión cuantificada siguiente:

$$(\exists i : 0 \leq i < \#xs : xs.i = 0)$$

que captura el hecho de que la secuencia `xs` contiene al menos un cero, se traduce en la siguiente expresión Alloy:

```
some i: Int | 0 <= i < #xs and xs[i]= 0
```

Como vemos, para el caso de las expresiones con dominio en los booleanos, la tradución a Alloy es bastante directa.

4.2. Cuantificadores Numéricos

La utilización de expresiones con sumatorias y productorias son naturales para los estudiantes, dado que las trabajan de manera intensiva en materias de matemática discreta y cálculo. El lenguaje de expresiones cuantificadas las introduce como una cuantificación más, incorporando una forma natural de capturar programas en esta notación. Esto genera sin embargo un problema al intentar utilizar Alloy como herramienta de análisis, ya que el lenguaje no provee soporte para sumatorias y productorias.

Para expresar sumatorias y productorias en Alloy, recurriremos a construir expresiones equivalentes que *computen* las sumas y productos correspondientes. Para este *cómputo*, utilizaremos el lenguaje DynAlloy, que extiende Alloy con la posibilidad de escribir *programas*, expresiones pseudo-declarativas que manejan

[1] En los ejemplos con los que se describe la traducción de las diferentes expresiones, se hace abuso de notación en la sintaxis Alloy a los fines de legibilidad.

estado. Esencialmente, dada una sumatoria o productoria, generaremos automáticamente un programa DynAlloy que "calcule" la expresión respectiva; esto estará oculto, por supuesto, al estudiante, dado que es parte de la traducción a Alloy (las sumatorias y productorias, capturadas a través de expresiones cuantificadas, seguirán siendo el "front-end" para el estudiante).

Para ilustrar la traducción, consideremos el siguiente programa DynAlloy, que calcula la sumatoria de los elementos de una secuencia:

```
program sumatoria[s:seq Int, suma:Int] var [indexSeq: Int] {
    indexSeq := 0;
    suma := 0;
    while (#s.elems > 0 and indexSeq <= lastIdx[s]) do {
        suma := s[indexSeq] + suma;
        indexSeq := indexSeq + 1
    }
}
```

Como vemos, la representación de una sumatoria en este caso está expresada con un programa, que recorre la secuencia y va acumulando en una variable la suma de los elementos en cada índice de la misma. La traducción sigue esta idea. Para capturar una expresión cuantificada aritmética arbitraria, nuestra traducción construye los elementos de la secuencia que sumará el programa DynAlloy, considerando el rango de las variables involucradas, las restricciones sobre la expresión y además el término que representa una expresión aritmética.

A continuación, a través de un ejemplo, describimos los detalles de la traducción de una especificación cuantificada con dominio numérico a DynAlloy. Consideremos la siguiente especificación, que captura la suma de los elementos pares de una secuencia:

$$(\sum i : 0 \leq i < \#xs \wedge par(xs.i) : xs.i)$$

Para traducir esta especificación introducimos en DynAlloy un predicado para representar el rango:

```
    pred rango[xs: seq Int, i: Int] {
        0 <= i and  i < #xs and par[xs.i]
    }
```

De manera similar, capturamos el término de la expresión cuantificada como una función:

```
    fun termino[xs: seq Int, i: Int]: one Int {
        xs[i]
    }
```

Finalmente, mediante un predicado, se construye la secuencia con los valores tomados de la función término, aplicada a las variables que cumplen con el rango:

```
pred buildSeq[xs: seq Int, res: seq Int] {
  #xs = 0 => #res = 0 else
  ( #xs <= 1 and #res <= #xs and
    (some k: inds[xs] | rango[xs,k]) implies #res > 0) and
    ( all i: inds[xs] |
      ( rango[xs,i] implies
        some j: Int | res[j]=termino[xs,i]  and j <= i
        and all e: res.elems | some k: inds[xs] | termino[xs,k] = e
        and # { z:inds[xs] | rango[xs,z] } = #inds[res]
      ) and
      !rango[xs,i] implies #{z:inds[xs] | rango[xs,z] } = #inds[res]
    )
  )
}
```

El predicado que construye la secuencia resultante considera primero el caso de una secuencia vacía, para el cual no construye ningún elemento. En caso que existan elementos para analizar, aparecen una serie de conjunciones de predicados que restringen los modelos con aserciones tales como que si algún índice cumple el rango, el resultado tiene elementos, para cada elemento del rango, el término correspondiente debe estar en el resultado, etc.

Continuando con el ejemplo, escribimos una aserción para comprobar que al aplicar el programa sumatoria a la secuencia resultante se obtiene lo esperado. Para esto definimos la siguiente aserción:

```
assert sumatoria[s, res: seq Int, suma: Int] {
    pre { buildSeq[s, res] }
    program { call sumSeq[res,suma] }
    post { par[suma'] }
}
```

Si la construcción de la secuencia es correcta, el resultado de la sumatoria de elementos pares de una secuencia debería dar como resultado un número par. Para comprobar si la aserción produce errores usamos el siguiente comando:

```
check sumatoria for 4  int, 2 seq lurs 4
```

Al ejecutarlo, obtenemos el siguiente resultado:

```
Executing "Check sumatoria for 4 int, 2 seq lurs 4 default"
    Solver=sat4j Bitwidth=4 MaxSeq=2 SkolemDepth=4 Symmetry=20
    4890 vars. 243 primary vars. 14379 clauses. 63ms.
    No counterexample found. Assertion may be valid. 94ms.
```

Como es de esperar, la aserción para una especificación correcta no muestra contraejemplos. Como parte de la evaluación de nuestra caracterización, sin embargo, observamos problemas que tienen que ver con la naturaleza no tipada de Alloy. Uno de estos surgió al analizar diferentes tipos de errores típicos de

estudiantes, en la construcción de especificaciones, como por ejemplo la definición de rangos inválidos. Por ejemplo, al indexar una secuencia, si un valor admisible por el rango para el índice no pertenece a la secuencia, el predicado que construye la secuencia con los elementos simplemente ignora los casos (las expresiones indefinidas se anulan en Alloy). Esto da como resultado que algunos errores no sean detectados. Para resolver este problema, incorporamos un programa DynAlloy adicional, que se encarga de recorrer los índices del rango y verificar que todos dan lugar a valores válidos del término:

```
program checkTermino[xs: seq Int, flag:Bool ]
 var [index,cantElems,i : Int, r: set Int] {
    flag := False;
    r := {z:Int| rango[xs,z]};
    index := min[r];
    i:=0;
    cantElems := #r;
    while (  lt[i,cantElems]  and (False = flag) ) do {
        if (  error [xs,index]) {
            flag := True
        } else {
            skip
        };
        r := r - {min[r]};
        i := add[i,1];
        if (! no r ) {
            index := min[r]
        }
    }
}
```

Utilizando este programa y el chequeo de la siguiente aserción:

```
    assert terminoValido[xs: seq Int, flag: Bool] {
        pre {  flag = False and (#s > 0) }
        program { call checkTermino[s,flag] }
        post { flag' = False }
    }
```

podemos distinguir los casos en los que la especificación permite indexar una secuencia en un índice inválido. Para ver cómo funciona el análisis, supongamos que la especificación tuviera el siguiente rango incorrecto:

```
    pred rango[xs: seq Int, i: Int] {
        0 <= i and i <= #xs and par[xs.i]
    }
```

Como vemos, el índice de la variable cuantificada abarca $\#xs$ mientras que los índices válidos de la secuencia no lo incluyen. Cuando chequeamos los términos válidos utilizando este nuevo rango, de la siguiente manera:

```
check terminoValido for 4 int, 3 seq
```

obtenemos lo siguiente:

```
Executing "Check terminoValido for 4 int, 3 seq"
    Solver=sat4j Bitwidth=4 MaxSeq=3 SkolemDepth=4 Symmetry=20
    3791 vars. 269 primary vars. 10600 clauses. 124ms.
    Counterexample found.  Execution. Assertion is invalid. 78ms.
```

Como vemos, la herramienta detecta un contraejemplo con instancias para las cuales, luego de ejecutar el programa que recorre los términos, la "bandera" se activa, indicando que existe un valor del rango para el cual el término está indefinido (y retornando un modelo que lo refleja). Esto permite detectar problemas de tipo rango inválido, divisiones por cero, etc., obteniendo situaciones concretas que lo ejemplifican, para que el estudiante las pueda analizar.

5. Conclusiones y Trabajo Futuro

En contextos educativos se hace cada vez más necesario contar con herramientas de análisis automático para asistir a los estudiantes en las diferentes etapas del desarrollo de software. En este trabajo tomamos un lenguaje formal de especificaciones, basado en la construcción de expresiones cuantificadas, y presentamos una traducción automática al lenguaje relacional Alloy, con la finalidad de utilizar el analizador automático Alloy Analyzer, un model finder, para analizar las especificaciones originales. El trabajo se basa en las ideas introducidas en [18], y muestra algunos desafíos técnicos en la construcción de la traducción, que demandó la utilización de un lenguaje operacional, DynAlloy, para capturar ciertas expresiones y propiedades de las mismas. Mostramos ejemplos en los cuales las herramientas de análisis proveen feedback sobre especificaciones, al estilo de rangos inválidos y problemas similares, equivalententes en cierta medida al tipo de feedback que se recibe al analizar programas. Cabe resaltar que las herramientas de análisis automático tienen limitaciones que están atadas a la expresividad de los lenguajes analizados. En nuestro caso, al trabajar sobre lenguajes de primer orden, no decidibles, es evidente que no se puede garantizar la equivalencia de dos especificaciones, ni garantizar la validez de ciertas fórmulas. El análisis aquí utilizado no es exhaustivo, sino acotado, pero aún así es sumamente útil, ya que en la práctica permite encontrar violaciones a propiedades esperadas dentro de las cotas de análisis (véase la *Hipótesis de la Cota Pequeña* [9] como argumento sobre este punto).

En cuanto a trabajo futuro, y de acuerdo a la experiencia obtenida en la representación de las cuantificaciones numéricas y los problemas que surgen durante la traducción a DynAlloy, consideramos importante analizar algunas herramientas tradicionales para el análisis de aritmética, como SMT (satisfiability modulo theories) solvers, que posiblemente tengan mejor soporte para el tipo de análisis que necesitamos.

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, *Cálculo de Programas*, Universidad Nacional de Córdoba, 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. D. Chaudhari y O. Damani, *Automated Theorem Prover Assisted Program Calculations*, en Proc. de IFM 2014, LNCS, Springer, 2014.
6. E. Clarke, O. Grumberg y D. Peled, *Model Checking*, MIT Press, 2000.
7. N. Eén, N. Sörensson, *An Extensible SAT-solver*, en Proc. de SAT 2003, LNCS 2919, Springer, 2004.
8. D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, The MIT Press, 2006.
9. D. Jackson, *Alloy: a lightweight object modelling notation*, in ACM Transaction Software Engineering and Methodology (ACM TOSEM), vol. 11, Nro. 2, 2002.
10. 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.
11. M. Frias, C. López Pombo y M. Moscato, *Alloy Analyzer+PVS in the Analysis and Verification of Alloy Specifications*, en Proc. de TACAS 2007, LNCS 4424, Springer, 2007.
12. J. Jacky, *The Way of Z: Practical Programming with Formal Methods*, Cambridge University Press, 1997.
13. J. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.
14. A. Levitin, M. Papalaskari, *Using puzzles in teaching algorithms*, en Proc. de SIGCSE 2002, USA, 2002
15. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik,*Chaff: Engineering an Efficient SAT Solver*, en Proc. de DAC 2001, ACM, 2001.
16. N. Tillmann y J. de Halleux, *PEX: White Box Test Generation for .NET*, en Proc. de TAP 2008, LNCS 4966, Springer, 2008.
17. 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.
18. C. Cornejo, M. Politano, F. Raverta, S. Permigiani, P. Ponzio, G. Regis, N. Aguirre, *Analizando el uso de (Dyn)Alloy como herramienta educativa*, In Proc. de CACIC 2015.
19. R. Singh, S. Gulwani and A. Solar-Lezama, *Automated Feedback Generation for Introductory Programming Assignments*,in Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2013.
20. G. Regis,; C. Cornejo, S.G. Brida, M. Politano, F. Raverta, P. Ponzio, N. Aguirre, J.P. Galeotti, and M. Frias, *DynAlloy analyzer: a tool for the specification and analysis of alloy models with dynamic behaviour*,in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017, pages 969–973, 2017. ACM.