Hacia un marco ontológico y epistemológico para una metodología de la verificación de los programas computacionales en ciencia de la computación

Martin Diller

1. Introducción

El problema de la "corrección" de los programas computacionales consiste en determinar si un programa computacional sastisface las especificaciones (el comportamiento deseado) del mismo. Para ello, algunas de las metodologías más comunes son el uso de la verificación formal y ciertos métodos empíricos como el "*testing*". En el caso de la verificación formal, se busca probar la corrección de manera deductiva empleando, por ejemplo, una lógica como la introducida por Hoare para razonar sobre programas imperativos (Hoare 1969). Esta consiste en un conjunto de axiomas y reglas de inferencia que hacen referencia al modo en que cambia el estado de un programa cuando se ejecutan las sentencias elementales que contiene el mismo. El "*testing*" constituye, a grandes rasgos, en ejecutar un programa con ciertos "casos de *test*", elegidos porque se espera que estos revelerán los errores del programa, comparando luego el resultado de la ejecución del programa con el comportamiento esperado del mismo en tales "casos de *test*".

El debate en torno de los alcances y ventajas del uso de la verificación formal para determinar la "corrección" de los programas computacionales ocupa un lugar central en la filosofía de la ciencia de la computación reciente, usándose incluso las diversas concepciones sobre esta cuestión como un criterio para distinguir entre distintas comunidades de investigación en ciencia de la computación con creencias ontológicas, valores epistémicos y metodologías que se suponen mutuamente excluyentes. Así, por ejemplo, en Eden (2007) se identifican tres "paradigmas" vigentes en ciencia de la computación (el "racionalista", "tecnocráta" y "científico") y se realiza una taxonomía de los mismos en torno de sus supuestos metodológicos, ontológicos y epistemológicos en la que los supuestos metodológicos y epistemológicos centrales de cada uno de los "paradigmas" giran en torno a la conveniencia y factibilidad de realizar verificaciones formales de los programas computacionales. También, en Colburn (2004) las diversas posiciones tomadas en el debate respecto de la verificación formal ocupan un lugar central para distinguir aquella comunidad de científicos de la computación que consideran a la ciencia de la computación como una rama de la matemática de aquellos que la consideran una disciplina ingenieril.

El "argumento de la confusión de categorías" (Eden 2007) presentado por Fetzer en la influyente "Communications of the ACM" (Fetzer 1988), con fines de marcar ciertos límites que tiene la verificación formal, ha tenido una influencia significativa en la literatura filosófica sobre los alcances de la verificación formal. En particular, ha tenido una marcada influencia sobre las ramificaciones

metodológicas de este debate: la discusión sobre qué métodos son los más adecuados para investigar a los programas computacionales.

Una consecuencia de la influencia del argumento presentado en Fetzer (1988) sobre el debate metodológico es que frecuentemente éste aparezca formulado en los términos ontológicos esbozados en este artículo. Por ejemplo, en Eden (2007) el "argumento de la confusión de categorías" se lo utiliza como un argumento a favor de las posturas metodológicas y epistemológicas del "paradigma" tecnócrata y científico en ciencia de la computación. Estos últimos "paradigmas", según la caracterización de Eden, defienden el uso de "*testing"* y una mezcla de métodos deductivos y experimentales como los métodos más adecuados para investigar la corrección de los programas computacionales respectivamente. También, de acuerdo con esta perspectiva, en Colburn (2004) se afirma:

> Fetzer revela que la aseveración de que es posible razonar de una manera completamente *a priori* sobre el comportamiento de los programas es cierta si el comportamiento es meramente abstracto; falsa y peligrosa en caso contrario. Esto garantiza la indispensabilidad de los métodos empíricos en el proceso de desarrollo de software, por ejemplo, el uso de testing para eliminar errores en los programas. (Colburn 2004, Pág. 322)

Un último ejemplo de este tipo de conclusiones proviene de la sección dedicada a la filosofía de la ciencia de la computación de la enciclopedia virtual de filosofía de la Universidad de Stanford:

> No hay suficiente cantidad de trabajo formal que pueda traspasar la barrera entre lo abstracto y lo físico: nunca podemos garantizar que cualquier ejecución particular de un programa en una máquina física se comportará de manera esperada. (…) Consistente con la naturaleza dual de los programas, podríamos decir que el programa como texto se encuentra sujeto a la corrección matemática, mientras que su contraparte física se encuentra sujeta a la verificación empírica. (Turner et al 2008, Pág. 18)

A partir de una revisión de los argumentos presentados por Fetzer (1988) y las reacciones posteriores al artículo de Fetzer de 1988, en este trabajo se busca establecer los rudimentos de un marco ontológico y epistemológico alternativo desde el cual encarar el debate metodológico. Se retoma, en primer lugar, los argumentos de Fetzer de 1988 en torno de los alcances de la verificación formal de la corrección de los programas computacionales. Se sugiere que una de las fuentes de mayor confusión generadas por el artículo de Fetzer es que en este artículo no se indica claramente a qué se refiere cuando se habla del "nivel físico" y "nivel lógico" de un programa computacional, siendo

que, en principio, para Fetzer la frontera entre ambos representaría una “barrera epistémica” infranqueable para la verificación formal. Se muestra cómo el precisar estas nociones atendiendo a los diversos componentes y niveles en que se puede analizar un sistema computacional y, en especial, a las relaciones entre estos, puede ofrecer una vía que permite precisar algunos de los límites de la verificación formal señalados por Fetzer en 1988 y que, a su vez, delimita un camino para resolver algunos de los puntos de mayor desacuerdo entre Fetzer y sus críticos.

En este contexto, se atiende, en particular, al uso del “argumento de la confusión de categorías” de Fetzer para argumentar a favor del uso del “*testing*” y ciertos métodos denominados “experimentales” como metodologías más adecuadas para investigar la corrección de los programas computacionales. A partir de la clarificación y reformulación de la perspectiva presentada por Fetzer sobre la naturaleza de los programas computacionales en 1988 descrita arriba, se mostrará que, en el modo en que se entienden habitualmente, tanto la verificación como los métodos empíricos propuestos como alternativas metodológicas en la literatura no difieren con respecto al “objeto” que se aplican (o “nivel” en que se puede analizar un programa computacional), por lo que los argumentos de Fetzer resultan tan (poco) destructivos para el uso de la verificación como para tales métodos empíricos.

Una vez establecida esta última conclusión se sugiere que, por un lado, algunas de las formas experimentales de investigar a los programas computacionales identificadas en la literatura reciente están orientadas a satisfacer objetivos distintos a los de la verificación formal y que, por ende, no deben ser vistas como métodos rivales para estudiar los programas computacionales sino, más bien, como métodologías que pueden complementarse. Por otra parte, el “*testing*” y la verificación formal en cuanto están orientados a satisfacer los mismos fines, sí podrían ser vistos como métodos potencialmente rivales para determinar la corrección de los programas computacionales. Se señala entonces al estudio del modo en que ambas metodologías pueden lidiar con diversas formas de complejidad de los sistemas computacionales, como el escollo principal a resolver para determinar las ventajas y desventajas de cada uno de éstos métodos. Finalmente, se hace eco de la tendencia cada vez más frecuente en ciencia de la computación de considerar, también, al “*testing*” y la verificación formal como métodologías complementarias y se señala una vía para avanzar en la construcción de un marco teórico desde el cuál elaborar y evaluar diversas estrategias metodológicas mixtas para determinar la corrección de los programas computacionales. Una convicción que estará en el trasfondo de este trabajo y que se perfilará a lo largo del mismo es la necesidad de alcanzar mayor claridad acerca de la naturaleza y relaciones entre el “*testing”*, verificación formal y ciertas formas experimentales de investigar los programas computacionales, además de los contextos de uso y los objetivos que se persiguen con cada una de estas metodologías con fines de avanzar en el debate metodológico en ciencia de la computación.

2. El argumento de la "confusión de categorías" de Fetzer

El artículo de Fetzer consiste, en parte, en una réplica a ciertos argumentos presentados por De Millo, Lipton y Perlis (1979) en contra de la verificación formal de los programas computacionales. Fetzer se concentra, en particular, en el argumento de De Millo, Lipton y Perlis en contra de la verificación formal basado en notar las prácticas sociales vinculadas a la aceptación de los teoremas en matemática. A grandes rasgos, estos autores realizan en su artículo una distinción entre "pruebas" y "pruebas formales" y argumentan que los objetos de evaluación para admitir nuevos teoremas en matemática son principalmente las "pruebas" y que los procesos que llevan a la aceptación de estas son, primariamente, de índole social. En particular, el rol jugado por las pruebas formales en matemática sería, según estos autores, prácticamente nulo. Procesos sociales importantes que determinan la aceptación de las pruebas en matemática, según DeMillo, Lipton y Perlis son su comunicación, diseminación, revisión por la comunidad, su transformación y uso en la prueba de otros teoremas y aplicaciones científicas y tecnológicas. Criterios epistémicos como la simplicidad jugarían, según estos autores, un rol importante en estos procesos de admisión de las pruebas por la comunidad de Matemáticos.

El que, por la naturaleza de las pruebas formales, sea imposible que existan procesos sociales semejantes para la aceptación de éstas haría fracasar, según De Millo, Lipton y Perlis, el proyecto de los verificacionistas en ciencia de la computación:

> Las verificaciones no son mensajes; una persona que saldría a la sala a comunicar su última verificación se encontraría rápidamente convertido en una paria social. Las verificaciones no se pueden leer; un lector puede hacer un intento heroico para leer algunas de las más cortas pero eso no es lectura. Siendo ilegibles y –literalmente- incomunicables, las verificaciones no pueden ser interiorizadas, transformadas, generalizadas, usadas, conectadas a otras disciplinas e incorporarse, eventualmente, a la conciencia comunitaria. (De Millo et al 1979, Pág. 275)

Fetzer argumenta, en contra de la posición de De Millo, Lipton y Perlis que ellos han encontrado una diferencia *en la práctica* pero no una *en principio*. De hecho, De Millo, Lipton y Perlis admiten implícitamente el valor epistémico que puede tener una prueba formal. Refiriéndose a las pruebas en matemática consideran que

> Luego de que estas hayan sido sometidas a un proceso de interiorización, generalización, transformación, uso, y suficientes conexiones [con otras ramas de la matemática hayan sido

> demostradas], la comunidad matemática eventualmente decide que los conceptos centrales en el teorema original, aunque a esta hora quizás algo modificados [por las transformaciones que sufre la prueba de un teorema hasta que éste es admitido], son estables. (…) El teorema es considerado verdadero en el sentido clásico- esto es, en el sentido que podría ser demostrado deductivamente mediante la lógica formal, aunque para la mayoría de los teoremas una deducción de este tipo no se lleva ni se llevará a cabo. (De Millo et al 1979, Pág. 274)

De este modo, si uno se imagina un cambio en las prácticas de los científicos de la computación, generado, quizás, como sugiere Fetzer, por un incentivo económico adecuado, este argumento de De Millo, Lipton y Perlis contra la verificación de programas fallaría. Por el otro lado, según Fetzer, habría razones, en principio más serias, para dudar de la posibilidad de verificación de los programas computacionales: "parece que De Millo, Lipton y Perlis han ofrecido algunos malos argumentos para algunas posiciones que requieren una mayor elaboración y que merecen más apoyo."(Fetzer 1988, Pág. 1049). Fetzer expresa sucintamente su argumento principal en el *abstract* de su artículo:

> Los algoritmos, como estructuras lógicas, son objetos apropiados para la verificación deductiva. Los programas, como modelos causales de estas estructuras, no. El éxito de la verificación de programas como un método generalmente aplicable y completamente confiable para garantizar el rendimiento de un programa no es siquiera una posibilidad teórica. (Fetzer 1988, Pág. 1048)

Fetzer realiza una distinción entre varios sentidos de "programa computacional" en su artículo, según, si uno se refiere a "programa" como i) un algoritmo, ii) una codificación de un algoritmo, iii) una codificación de un algoritmo que puede ser compilada o iv) una codificación de una algoritmo que puede ser compilada y ejecutada por una máquina. Según Fetzer, la diferencia crucial entre los programas en los sentidos i) y ii) y los sentidos iii) y iv) es que mientras los primeros pueden estudiarse meramente a través de una máquina abstracta, los últimos dos sentidos involucran una referencia a una máquina física. Así, las verificaciones de la corrección de los programas en el primer sentido serán "absolutos" del mismo modo en que, según Fetzer, lo serían las pruebas en la lógica y matemática pura: serán derivables a partir de axiomas primitivos que son verdaderos en virtud de sus significados únicamente y de reglas de inferencias que preservan la verdad. Por el otro lado, las verificaciones de los programas en el sentido de iii) y iv) serían verificaciones relativas: éstas involucrarán premisas y axiomas causales que pueden no resultar ciertos. Así, dada una relación de computación entre entradas E y salidas S de la forma EcS y un axioma de la forma "De EcS y E inferir S", según Fetzer:

> [Por un lado] las reglas y axiomas de la forma EcS pueden ser interpretados como verdades por definición que se aplican a la máquina abstracta que se define de esta manera. Por el otro lado, las reglas y axiomas de la forma IcS pueden ser consideradas como afirmaciones empíricas sobre el comportamiento posible de una máquina que se describe de esta manera. Interpretado de la primera forma, el desempeño de una máquina abstracta puede ser verificado de manera concluyente, pero no posee significancia alguna para el desempeño de cualquier sistema físico. Interpretado de la segunda forma, el desempeño de la máquina abstracta posee significancia para el desempeño de un sistema físico, pero no puede ser verificado de manera concluyente. Y la razón (…) debería resultar clara: los programas están sujetos a verificación "relativa", no "absoluta", en relación a 'reglas y axiomas' en la forma de generalizaciones en forma de leyes causales como premisas- afirmaciones empíricas cuya verdad no puede nunca establecerse con certeza. (Fetzer 1988, Pág. 1059)

3. ¿Qué establece una prueba formal de la corrección de un programa computacional respecto de la ejecución del programa en una máquina?

Como expresa el editor de la *Communications of the ACM* en una nota posterior a la publicación del artículo de Fetzer de 1988, ésta tuvo como consecuencia inmediata "una gran cantidad de correo negativo [fue enviado a la *Communications of the ACM* de parte] de investigadores trabajando en la verificación forma expresando dudas acerca de la responsabilidad de los autores, editores, evaluadores y la política de publicaciones de la *ACM*". Desde entonces, los debates en torno de los alcances de las críticas de Fetzer a la posibilidad de la verificación formal han continuado, también, en otros medios.

Según expresa Fetzer en una respuesta a una de las muchas críticas recibidas a su artículo, los objetivos del mismo eran más bien modestos. En particular, el artículo, según Fetzer, estaba dirigido, principalmente, a explicitar los límites de afirmaciones como la siguiente de Hoare:

> La programación de computadoras es una ciencia exacta en el sentido de que todas las propiedades de un programa y todas las consecuencias de su ejecución pueden, en principio, conocerse a partir del texto del programa por medio del razonamiento deductivo. (Fetzer 1989, Pág. 378)

Por otra parte, la polémica generada alrededor del artículo de Fetzer posiblemente se deba a la retórica un tanto infortunada del artículo, así como a algunas afirmaciones de Fetzer (inclusive el título) que parecen indicar que la verificación formal carece completamente de utilidad para asegurar

la corrección de los programas computacionales (en una máquina física). A esto se le suma el hecho de que, como se indicó arriba, Fetzer expresa en su artículo explícitamente que pretende ofrecer mejores argumentos a favor de la posición expresada por De Millo, Lipton y Perlis (1979) que se acercan bastante a avalar este último punto de vista (Fetzer 1988, Pág. 1049).

Uno de los aspectos de la argumentación de Fetzer que, posiblemente, genera mayor confusión y explica algunas de las afirmaciones más drásticas de Fetzer, así como también las críticas más fuertes a su artículo, es que Fetzer no distingue de manera muy clara entre los diversos componentes involucrados en la producción y ejecución de un programa computacional, las relaciones entre estos, y, por ello, tampoco entre los diversos niveles en que se puede estudiar y analizar un programa computacional. En particular, reduce en su artículo la diferencia entre estos diferentes niveles a la diferencia entre lo que llama "físico" y "abstracto" y no queda muy claro dónde y cómo trazar el límite entre estos dos niveles, cuya frontera resultaría infranqueable para la verificación formal. Más aún, la noción de "abstracto" implícita en la discusión de Fetzer es tomada de la filosofía de la lógica y matemática pura, cuando, como ha observado acertadamente Barwise (1989), el debate en torno a los alcances de la verificación formal gira en torno de la posibilidad de una forma específica de matemática aplicada.

En efecto, la perspectiva de Fetzer contrasta con la situación habitual en ingeniera de software donde, en primer lugar, se distingue entre varios niveles de abstracción en que se pueden analizar un programa o sistema computacional (Colburn 1999) y, por el otro lado, el objetivo, habitualmente, es el diseño de software en la forma de "capas de abstracción", donde la interfaz y, por lo tanto, la relación entre las capas se encuentra bien definida (Colburn 2011). Una consecuencia de ello es que las propiedades de alto nivel de los programas sean traducibles de manera relativamente precisa en las propiedades que tendrá la ejecución de los programas en una máquina.

Arkoudas y Bringsjord (2007) ofrecen una lista de aquellos componentes en que se debe confiar para poder fíarse en una prueba revisada por medios automáticos de un teorema matemático como el de los cuatro colores. Extendiendo y modificando en algunos puntos esta lista, se pueden identificar al menos 7 aspectos en que se debe tener confianza para poder confiar en un programa que haya sido verificado formalmente:

1. Funciones que son parte de otros componentes que usa o con los que se comunica el programa en cuestión.
2. El sistema operativo en el que se ejecuta el programa en cuestión. (En particular, esto significa, habitualmente, también el control de la comunicación entre los varios programas

cuando el programa en cuestión interactúa con otros programas, así como también el control de la interacción del programa con diversos dispositivos de hardware)

3. El compilador para el lenguaje en que esta escrito el programa (y todos los compiladores intermedios hasta la producción del código ejecutable)
4. La semántica del lenguaje en que está escrito el programa (así, como la semántica de todos los lenguajes de programación involucrados en la compilación a código ejecutable). También se debe confiar en que el sistema de prueba que se use para probar la corrección se encuentre adecuadamente conectado a la semántica usada para la construcción del compilador.
5. Si lo que ha sido verificado formalmente de los componentes anteriores *no* es el código fuente (por ejemplo, mediante la verificación asistida por medios automáticos o por algún enfoque de derivación de la implementación a apartir de las especificaciones), se debe confiar en la codificación de cada uno de los componentes de arriba (es decir, no debe haber errores producidos por la programación).
6. El hardware en el que se ejecuta el programa en cuestión.
7. La integridad de la máquina particular -hardware, sistema operativo y el/los compilador/es- en los que se ejecuta el programa (es decir, para confiar en el programa se debe confiar en que no hubo agentes maliciosos que hayan saboteado el sistema).
8. Factores “causales” generales: errores de hardware azarosos como influencia de rayos cósmicos, etc.

Los aspectos indicados en el punto 7 no atañen directamente al problema de la “corrección” de los programas computacionales, y los aspectos señalados en el punto 8 sirven para recordarnos de que siempre puede haber imprevistos. Por otra parte, el argumento de Fetzer, aunque podría ser interpretado como un recordatorio de esto último, en principio está orientado a señalar límites más específicamente vinculados al problema de la “corrección” de un programa computacional, como pueden ser aquellos aspectos detallados en los puntos 1-6. Esto queda más claro en un artículo más reciente de Fetzer (Fetzer 1999) en que se destaca al interpréte y compilador como posibles fuentes de error en el paso del código al “proceso”.

La perspectiva habitual sobre el diseño de los sistemas computacionales en ingeniera de software a la que se aludió anteriormente, sugiere que debería ser posible precisar las relaciones entre las diversas “transformaciones” involucradas en el paso del programa como “código” al programa como “proceso”. En conjunto con un analísis de la confianza que es razonable depositar en algunos de los aspectos en los puntos 1-6 en la lista anterior, esto, permitiría ver “lo que dice” la verificación formal

de un programa computacional respecto de la ejecución del mismo en una máquina. De este modo, podría precisarse y evaluarse los alcances de algunas afirmaciones de Fetzer en su artículo de 1988.

En lo que sigue ilustraré los rudimentos de un analísis de este tipo considerando lo que, a partir de la descripción de Fetzer de los límites de la verificación formal, pareciera son dos de los componentes más problemáticos a la hora de confiar en la verificación formal. Estos son, en primer lugar, el compilador, que hace de "puente" entre la "codificación del algoritmo que puede ser compilada" y la "codificación del algoritmo que puede ser compilada y ejecutada", según la descripción de Fetzer más arriba y, en segundo lugar, el hardware (en particular, el procesador), que ejecuta ésta última "codificación del algoritmo".

Prestando atención a la práctica de la construcción y de la verificación de compiladores, es de notarse que el proceso de compilación involucra alguna forma de "preservación de la semántica de los programas computacionales". Es decir, dadas semánticas que asocian comportamientos observables (en el caso más simple, los valores de entrada y los de salida) a los códigos fuente y código objeto, un compilador correcto traduce un programa en el código fuente a uno equivalente en el código objeto desde el punto de vista de las semánticas involucradas.

Así, por ejemplo, en el compilador verificado por medios semi – automáticos (usando el asistente de pruebas interactivo Coq) CompCert (Leroy, 2009) de código en Clight (un subconjunto de C) a un subconjunto de PowerPC assembly (lo cual involucra 14 traducciones intermedias usando 8 lenguajes distintos), la corrección del compilador significa concretamente que, dadas semánticas que asocian comportamientos observables B a los códigos fuente S y el código objeto C

$$\forall\, S,C,B \notin Error, Comp(S) = OK(C) \wedge C \downarrow B \Rightarrow S \downarrow B$$

donde Error es el conjunto de comportamientos con errores, (como, por ejemplo, la división por cero, donde la computación que involucra este error puede ser eliminada por el compilador si no es usada por el programa), Comp(S) = OK(C) significa que el compilador produce el código objeto y, finalmente $X \downarrow B$ significa que X se ejecuta con el comportamiento B.

Esto sugiere una vía concreta para establecer "lo que dice" la verificación formal de un programa computacional acerca de la codificación "ejecutable" de este programa[1]: en el caso de que el sistema

[1] Estrictamente hablando, CompCert no garantiza que la compilación al código ejecutable sea "correcta", ya que este sólo produce un código en lenguaje assembly. Sin embargo, como se sugiere en Leroy (2009, Pág. 114), aunque esto no sea muy interesante, el verificar formalmente el "assembler" y "linker" necesarios para traducir el código assembly a código binario, no presenta un desafío técnico mayor.

de prueba usado para la verificación respete la semántica usada para construir el compilador del programa y de que el compilador haya sido verificado formalmente, la codificación "ejecutable" del mismo también será correcta respecto a la relación de "preservación semántica" entre el código fuente y el código objeto (además del asistente de prueba interactivo Coq usado para realizar la prueba y derivar el compilador a partir de las especificaciones funcionales del mismo[2]). Los comportamientos observables, asignados por la semántica de los programas que preserva la "traducción" del código fuente al código objeto, realizada por CompCert son, por ejemplo, la terminación y divergencia de los programas, los errores que pueden llevar al programa a estrellarse y los comportamientos de entrada-salida ("la traza de ejecución"), es decir, en palabras de los constructores del compilador, "aquello que puede observar el usuario del programa o, de modo más general, el mundo exterior con el que interactua el programa" (Leroy 2009, Pág. 208).

El programa codificado en el lenguaje máquina es el que finalmente se ejecuta en una máquina concreta. Esta máquina se compone, habitualmente, de varios componentes discretos y la composición de éstos (el diseño del hardware) puede, nuevamente, ser objeto de verificación formal. En particular, es de esperarse que el procesador ejecute el programa codificado en el lenguaje máquina "preservando la semántica" de manera análoga a como el compilador "traduce" el código fuente al código objeto, es decir, preservando las propiedades observables asignadas al código por la semántica del lenguaje máquina. Por otra parte, la verificación formal del hardware se realiza habitualmente en lo que suele llamarse el "nivel lógico" de la máquina: se supone que los circuitos digitales que componen una computadora típica, por ejemplo, manejan "bits" discretos de información que pueden modelarse adecuadamente mediante el álgebra booleano, ignorándose las propiedades "físicas" de los sistemas en que se implementan tales circuitos. Así, en una interpretación posible de la tesis de Fetzer de 1988 es en este punto, fundamentalmente, donde se introduce la incertidumbre vinculada al "mundo físico" que no es tenida en cuenta en la verificación formal de un programa. Ésta es una interpretación que Fetzer admite implícitamente (Fetzer 1989, Pág. 380) en una respuesta a una de las críticas publicadas en Communications of the ACM, en que se argumenta que una descripción matemática del hardware al nivel de las compuertas lógicas reduciría a la brecha entre el algoritmo y su ejecución hasta tal punto que sólo la relación entre la descripción matemática de las

[2] Esto, como se sugiere en Leroy (2009, Pág. 114) no es tan problemático como puede parecer. Por un lado, una prueba que es verificada mecánicamente por un asistente de pruebas es más confiable que una prueba moderadamente compleja inspeccionada por un ser humano. En todo caso, el estatus epistémico de esta prueba es analógo al estatus epistémico descrito por Arkoudas y Bringsjord (2007) de la nueva prueba del teorema de los cuatro colores, realizada por Gonthier utilizando Coq en 2004 : el programa que genera esta prueba emite un "certificado" en la forma de los pasos de la prueba que puede luego ser chequeado por un *proof checker* el cual, a su vez, puede estar verificado mediante una prueba que es suficientemente corta para poder ser inspeccionada fácilmente por un ser humano. Por otra parte, como también se describe en Leroy (2009), puede verificarse el mecanismo de extracción de código a partir de las especificaciones funcionales escritas en Coq y construirse un compilador de Caml, que es el lenguaje en que se encuentra el código generado por el mecanismo de extracción, a Cminor en cuyo caso se tiene un compilador que, a su vez, se auto-compila a través de una forma de *bootstrapping*.

compuertas lógicas y las propiedades físicas del silicio estarían sujetas a la distinción de Fetzer.

La vía seguida en Scheutz (1999) para elucidar la noción de "implementación física" en términos de la noción de "realización de una función de manera práctica", inspirada en la práctica de la construcción de computadoras, sugiere una forma muy general en que se podría precisar la relación entre la descripción matemática del hardware (por ejemplo, una compuerta lógica) y la descripción "física" del mismo (por ejemplo, utilizando la teoría de circuitos electrícos la cual, a su vez, resulta de la abstracción de propiedades eléctricas y leyes de la teoría de campos electromagnéticos que surgen al investigar diversas combinaciones espaciales de materiales conductores e aislantes). La idea básica acerca de cuándo un sistema físico S, que se encuentra descrito por una teoría física P (que permite, entre otras cosas, identificar las entradas y salidas en S), realiza una función f es que se puede definir una función F en términos del lenguaje y de las leyes de P que preserve el comportamiento de entrada-salida que prescribe la función f[3].

Sin embargo, esta noción básica es precisada en relación a varios aspectos en Scheutz (1999) para arribar a la noción de "realización de una función *de manera práctica*", la cual toma en en cuenta los problemas relacionados con la precisión con que se producen y se pueden medir las señales de entrada, confiabilidad y rango de funcionamiento de los sistemas físicos, el ruido generado por el medio ambiente, etc. Así, los sistemas que soportan sistemas digitales tienen la propiedad de que existen procedimientos fiables para aplicar las entradas y medir las salidas dentro de los límites operativos del sistema y para los cuales la cantidad de valores discretos que se puede distinguir dentro de tales limites es finita[4].

---

[3]En términos formales, esto se puede expresar de la siguiente manera:
Una función f con dominio D y rango R es realizada por un sistema físico S (descrito en una teoría P) si y sólo si se dan las siguientes condiciones:

1. Existe un isomorfismo I entre el 'dominio de entrada' de S a D
2. Existe un isomorfismo O entre el 'dominio de salida' de S a R
3. Existe una función F que describe la propiedad física (el comportamiento) de S para las propiedades de entrada-salida dadas (es decir, F es un mapeo del 'dominio de entrada' de S a su 'dominio de salida', descrito en el lenguaje y por las leyes de P) tal que para todo $x \in D$ se cumple lo siguiente: $O(F(I^{-1}(x)) = f(x)$

[4] La siguiente definición formal caracteriza un sistema físico que *"soporta sistemas digitales"* y tiene en cuenta el tiempo y rango de funcionamiento en el que opera el sistema:
Una función discreta f(x,t) con dominio D × *Tiempo* y rango R × *Tiempo* es *realizada de forma práctica dentro de* $[x_{min}, x_{max}] \subseteq$ D por un sistema físico S (descrito en una teoría P) con errores e_D para las magnitudes de entrada y salida y e_T para los tiempos de entrada-salida, si y sólo si se dan las siguientes condiciones:

1. Existe un isomorfismo I entre intervalos disjuntos del 'dominio de entrada' de S (donde el tamaño de los intervalos es de $> 2*e_D$) a D
2. Existe un isomorfismo O entre intervalos disjuntos del 'dominio de salida' de S (donde el tamaño de los intervalos es de $> 2*e_D$) a R
3. Existe un isomorfismo T entre intervalos disjuntos de *Tiempo Real* (donde el tamaño de los intervalos es $> 2*e_T$) y *Tiempo*.
4. Existe una función F que describe la propiedad física (el comportamiento) de S para las propiedades de entrada-salida dadas sobre el tiempo (es decir, F es un mapeo del 'dominio de entrada' × *Tiempo Real* de S a su 'dominio de salida' × *Tiempo Real* descrito en el lenguaje y por las leyes de P) tal que para todo $t \in$ *Tiempo* y $x \in [x_{min}, x_{max}]$ se cumple lo siguiente: si $F(I^{-1}(x), T^{-1}(t)) = \langle X, Z \rangle$, entonces $\langle X, Z \rangle = \langle O^{-1}(x'), T^{-1}(t') \rangle$ (donde $f(x,t) = \langle x', t' \rangle$).

Las definiciones presentadas por Scheutz (1999) pueden ser generalizadas a casos en que se consideren sistemas más complejos (realizando composiciones de funciones), como, por ejemplo, un registro o un contador. Precisada para tener en cuenta las límites de los sistemas físicos y así generalizada, la noción de "realización de una función de manera práctica" permite ver de manera bastante clara cómo se conectan las descripciones matemáticas, - sujetas a la verificación formal-, y las descripciones "físicas" del hardware.

La discusión anterior no deja de ser un atisbo superficial de lo que puede ser un análisis muy detallado de la confianza que es razonable depositar en los diversos componentes involucrados en la producción y ejecución de un programa computacional y, en particular, de la relación entre las diversas formas que toma un programa computacional hasta ser ejecutado en una máquina. Sin embargo, se espera que sirva para mostrar que es posible realizar un analísis bastante más minucioso que el que aparece en Fetzer (1988) acerca de las implicancias que tiene la verificación formal de un programa para la "corrección" de la ejecución del programa en una máquina y que, un analísis de este tipo revelaría que, contra lo afirmado por Fetzer, la verificación formal puede tener, también, aunque de manera derivada, "significado empírico".

De acuerdo con ésto último, el analísis anterior también sugiere que la implementación de un "algoritmo" en una "máquina" puede ser visto como un proceso de refinamiento sucesivo en que cada una de las fases de refinamiento de la implementación del algoritmo genera una "entidad" que puede, en principio, ser objeto de verificación formal y, si la conexión entre estas diversas fases se realiza de manera adecuada, finalmente resulta cierta la siguiente afirmación de Hoare, también citada por Fetzer en su artículo de 1988:

> Cuando la corrección de un programa, su compilador, y el hardware de la computadora se han establecido con certeza matemática, será posible depositar una gran confianza en el resultado del programa y predecir sus propiedades con una confianza limitada únicamente por la confiabilidad en la electrónica (Fetzer 1988, Pág. 1059)

Más aún, se vió como sería, en principio, posible establecer con bastante precisión el grado de confianza que se puede tener en la electrónica (aunque siempre queda abierta la posibilidad de imprevistos que tienen que ver, entre otras cosas, con el grado en que se encuentre corroborada la teoría "física" usada para guíar la implementación del hardware).

Por otra parte, resulta claro que si, como es muy común en la práctica, el compilador y el hardware,

por ejemplo, no se encuentran verificados formalmente, se reduce considerablemente la confianza que se puede depositar en la verificación formal de un programa que se ejecuta en tal entorno de ejcución. Sin embargo, incluso si este escenario ideal no concuerda con la realidad puede verse la utilidad que puede tener la verificación formal ya que, como mínimo, es probable que el compilador, hardware, etc. hayan sido diseñados con la *intención* de adecuarse a la noción de corrección que supone la prueba de verificación formal (desde otra perspectiva, la noción de corrección formal pretende *modelar* la noción de corrección informal[5]).

4. ¿Implica el argumento de la confusión de categorías que el uso de métodos "empíricos" es indispensable para determinar la "corrección" de los programas computacionales?

En la sección anterior se ofrecieron algunos indicios respecto a cómo sería posible traducir la verificación formal de un programa en términos de la "corrección" del mismo cuando éste se ejecuta en una máquina, así como bajo que condiciones la verificación formal permite depositar bastante confianza en que el programa satisfaga las especificaciones del mismo. Por otra parte pudo apreciarse a partir de la discusión anterior que, aunque matizada, la conclusión general del artículo de 1988 de Fetzer es correcta: la verificación formal de un programa computacional por sí misma no ofrece garantías acerca de la "corrección" de la ejecución del programa en una máquina. Se identificaron al menos dos fuentes posibles de "error" de naturaleza distinta que hacen que la verificación formal no pueda ofrecer garantías acerca de la "corrección": una es la construcción del hardware y la otra se debe a que los programas computacionales se construyen habitualmente en capas y cada una de estas capas (y la conexión entre ellas) introduce la posibilidad de error.

Esta situación puede llevar a la conclusión de que ciertos métodos empíricos serán indispensables para determinar la corrección de los programas computacionales. Específicamente, en la literatura se han propuesto al "*testing"* así como ciertas formas de experimentación como metodologías alternativas que vendrían a suplir las deficiencias de la verificación formal. En el marco de una forma general de argumentación a favor del uso de tales métodos empíricos en lugar de la verificación formal con el fin de establecer la "corrección" de los programas computacionales, el uso del argumento de la "confusión de categorías" de Fetzer con este propósito involucra, implícitamente, las siguientes premisas (algunas de las cuales son conclusiones de otros argumentos):

[5]Esta perspectiva sugiere que pueden introducirse "errores" si el modelo de "corrección" no se ajusta a la noción informal de "corrección" considerada, por ejemplo, por los programadores. Esta cuestión tiene que ver con la confianza que se puede depositar en la semántica formal de los lenguajes de programación a la que se ajusta el sistema de prueba que se use para la verificación formal. De hecho, Leroy (2009, Pág. 114) sugiere que establecer la adecuación de la semántica de los lenguajes de programación posiblemente sea el aspecto más díficil para garantizar la "corrección" del compilador CompCert, aunque se sugiere, también, algunas maneras de resolver el problema: inspección de la semántica por expertos (la semántica es relativamente chica), "testing" de formas ejecutables de la semántica y probar conexiones con semánticas formales formuladas independientemente.

1. La verificación formal no tiene la ventaja principal X que se le atribuye.
   En Fetzer (1988) (y también en De Millo et al (1979)) esta ventaja es ofrecer garantías absolutas acerca de la corrección del programa computacional.
2. La verificación formal no ofrece otra ventaja significativa sobre los métodos empíricos.
3. Los métodos empíricos sí ofrecen alguna otra ventaja significativa.
   En esta sección se considerará que la ventaja en cuestión es que tales métodos permiten obtener conocimiento sobre la ejecución del programa en una máquina, mientras que la verificación formal no. En la sección siguiente se evaluará el argumento de que la ventaja, en particular, del "*testing"* es que este es un método más adecuado que la verificación formal debido a la complejidad de los sistemas comptuacionales y a la naturaleza informal y a la "simplicidad" del "*testing"* en contraposición a la verificación formal.

Todas las cartas con críticas al argumento de Fetzer publicadas en la Communications of the ACM luego de haberse publicado este artículo cuestionan la caracterización que el mismo hace de los objetivos de la verificación formal (ofrecer garantías absolutas de la corrección de un programa). Una objeción relacionada a cuestionar la primera premisa explícitada arriba y que también esta implícita en la mayoría de las críticas al artículo de Fetzer publicadas, es que plantear el problema de la "corrección" de los programas en términos de la dicotomía "garantía absoluta" vs "ninguna garantía" equivale a rechazar la posibilidad de cualquier tipo de conocimiento (sobre el mundo). En particular, en un lenguaje más cercano al de la matemática aplicada, puede argumentarse que la verificación formal, a pesar de no ofrecer "garantías absolutas", sí ofrece *mayores* garantías que el "*testing*", por ejemplo. Esta es una manera de defender el uso de la verificación formal que es prometedora y parte de la cual involucra analizar la relación entre las propiedades de alto nivel sujetas a la verificación formal y las de la ejecución del programa en una máquina, además de las condiciones bajo las cuales la verificación formal ofrece garantías absolutas, módulo el estado de arte del hardware actual. Otra parte se la retomará brevemente en las próximas secciónes cuando se señalan algunas de las relaciones entre "*testing*" y verificación formal e involucra mostrar porqué la verificación formal ofrece mayores garantías que el "*testing*" en particular.

Más adelante se verán algunas otras formas en las que se puede dar contenido a la premisa 3 de arriba usando algunos argumentos contra el uso de la verificación formal que aparecen en De Millo et al (1979) y que han sido reelaborados por otros autores. Sin embargo, un analísis de las formas de "*testing*" y ciertas formas de experimentación propuestas como alternativas a la verificación revela que, en la forma en que aparece detallada arriba, la premisa 3,- esto es, la concepción implícita en algunas de las citas de la introducción de este trabajo de que el "*testing*" y tales formas de experimentación permiten obtener conocimiento sobre la ejecución del programa en una máquina,

mientras que la verificación formal no,- debe ser revisada.

Como fue afirmado en la introducción del trabajo presente, el "*testing*" constituye, a grandes rasgos, en la ejecución de un programa con ciertos "casos de *test*", elegidos porque se espera que estos revelerán los errores del programa, comparando luego resultado de la ejecución del programa con el comportamiento esperado del mismo en tales "casos de *test*". En el caso de los "*tests*" de caja negra, que se encuentran dirigidos a testear la funcionalidad del programa, los "*tests"* a realizar se eligen en base a las especificaciones del programa. Por ejemplo, si se quiere testear el comportamiento de un programa que controla el funcionamiento de un motor, el cual puede ser seteado a cualquier valor entero entre 0 y 100 ciclos por segundos en una dirección y en la dirección contraria, se lo ejecuta con lo que, en base a las especificaciones se determinan son valores representativos -50, 0 y 50, tomados de los subconjuntos [-100,0), [0], (0,100], además de los valores -101, -100, 99, -1, 0, 1, 99, 100, 101, los cuales se encuentran en los lugares en que se espera que cambie el comportamiento del sistema (Bogdanov et al, 2009). En el caso de los "*tests*" de caja blanca, diseñados con fines de testear la implementación del programa, los "*tests*" se realizan en base a la estructura del programa (habitualmente el código mismo). Por ejemplo, un objetivo en la elección de los conjuntos de casos de tests de caja blanca es que los conjuntos de valores elegidos cubran la mayor cantidad de las ramas posibles del grafo de flujo de control (una representación de todos los caminos que pueden atravesarse del programa), el cual puede ser derivado a partir del código del programa.

Ahora, en general el resultado de un experimento se interpreta en el marco de ciertos conocimientos y expectativas previas a la realización del experimento. En el caso de los "*tests*" de caja negra se ve que tales expectativas (los "casos de *test*") se formulan en base a las especificaciones del programa mientras que en el caso de los "*tests*" de caja blanca éstas se formulan en base a la estructura del programa. Tales especificaciones y estructura pueden serlo del programa en las diversas fases de refinamiento sucesivo de la implementación del mismo. De este modo, se ve que aquello que se investiga en el caso del "*testing*" no es fundamentalmente distinto de lo que se investiga mediante la verificación formal: se investiga al funcionamiento del programa en algún nivel de abstracción que, como tal, puede ser representado en forma textual (como "código") en algún lenguaje de programación de mayor o menor generalidad. En ningún momento, por ejemplo, el "*testing*" de programas computacionales hace referencia a la implementación "física" del programa en el sentido descrito antes.

Frecuentemente, en el contexto de la validación de software, se usa "*testing*" y experimentación como intercambiables, sin embargo puede distinguirse al menos tres formas de experimentación en ciencia de la computación que difieren de lo que habitualmente se considera como "*testing*". Una

forma de experimentación es la considerada por Eden (2007, Pág. 155) en el que se usa un programa para explorar dominios de otras ciencias, siendo el uso de simulaciones computacionales para estudiar fenómenos físicos, astronómicos, biológicos, etc el caso paradigmático. Otro caso particularmente interesante, ya que involucra una forma radicalmente distinta de "programación" (Partridge, 2000), es el de los programas construidos mediante "programación inductiva", por ejemplo mediante redes neuronales o programación lógica inductiva. En éstos, en lugar de especificaciones, se producen ejemplos del comportamiento deseado del programa (por ejemplo, imagenes de caras asociadas a determinados nombres o descripciones de los estados de algún dispositivo según si estos estados representan fallas o no) y se induce un programa que se comporta según el comportamiento deseado (por ejemplo, el reconocer caras o fallas en un dispositivo) con cierto grado de confianza que puede determinarse ejecutando el programa varias veces y utilizando metodologías estadísticas para evaluar su "éxito".

En todos estos casos, resulta claro que el uso de la experimentación no es para fines de comprobar, por ejemplo, que el sistema operativo, el compilador o el hardware se encuentre bien implementados, sino que es incluso más importante aún que se confie en que estos componentes se encuentren funcionando adecuadamente (¿que interpretación recibe un error de compilación en una simulación de la formación de una galaxia?). En particular, se debe resistir la tentación en que se incurre en Eden (2007) de concluir que estos usos de experimentación en ciencia de la computación, sugieren que esta representa, en algunos casos, una metodología *alternativa* a la verificación formal para estudiar la "corrección" de los programas. En este caso, el estudio del programa debe ser distinguido del estudio del modelo que se explora mediante el programa; en particular, el hecho de que el resultado de la simulación no pueda predecirse con antelación a la ejecución del programa no significa que no se pueda probar que el programa sea correcto en el sentido de que cumple con la funcionalidad de ser una simulación del fenómeno bajo estudio. Incluso en el caso de la programación mediante métodos inductivos es posible verificar formalmente el algoritmo usado para inducir el programa en cuestión. Por ejemplo, puede verificarse que la red neuronal se encuentre implementada correctamente. Estos ejemplo muestran que, al menos en estos casos, la experimentación y la verificación formal representan actividades complementarias, las cuales se encuentran dirigidas a satisfacer diversos objetivos metodológicos.

Casos que también demuestran, como se arguye en Eden (2007), que hay lugar tanto para el uso de métodos deductivos como experimentales en ciencia de la computación, son aquellos en que la "verificación formal" supone alguna forma de experimentación previa (dejando de lado la cuestión de la corroboración de la teoría física usada para guíar la implementación del hardware considerada antes). Este es el caso considerado en Simon (1996) de los dominios computacionales

(principalmente, aquellos muy complejos), como sucedió con la construcción de los sistemas de tiempo compartido, para los cuales no existe una teoría que sirva para guiar el diseño de tales sistemas y para los cuales la única vía para desarrollar y mejorar tales sistemas es el de "construirlos y ver cómo se comportan" (Simon 1996, Pág. 20). En estos casos, la verificación formal de un sistema involucra axiomas y reglas que deben ser interpretados de manera hipotética, aunque debe distinguirse, también en estos casos, a la actividad de corroboración de tales axiomas y reglas (la teoría que subyace a la verificación formal) de la actividad de asegurar la "corrección" de un programa computacional particular, usando la teoría en cuestión (Blanco et al 2008).

5. Verificación formal, "testing" y la complejidad de los sistemas computacionales

Las secciones anteriores tuvieron como propósito principal desembarazar la discusión metodológica acerca de qué métodos conviene usar para investigar la "corrección" de los programas computacionales de los términos abstractos y un tanto equívocos en que se presenta el "argumento de la confusión de categorías" de Fetzer en su artículo de 1988. En primera instancia, se vio que en la práctica este argumento no ofrece suficientes razones para negarle valor a la verificación formal como medio de investigar a los programas computacionales, aunque quede en pie el determinar si este valor justifica el uso de la verificación por sobre otras metodologías como el "*testing*".

Luego, en la sección anterior, se vio, en primer lugar, que algunas de las formas de experimentación usadas en ciencia de la computación pueden ser vistas como metodologías complementarias cuyos objetivos difieren del de la verificación formal y, en algunos casos, que la verificación formal representa una buena vía para aumentar la confianza en los resultados de las mismas. En segundo lugar, se ofrecieron indicios de que, como se entienden habitualmente a estas metodologías, la diferencia entre "*testing*" y verificación formal, no consiste en el hecho de que estas se aplican a "objetos" distintos, sino que representan caminos metodológicos alternativos que se pueden emplear en la investigación de los programas computacionales en las diversas fases de refinamiento que sufre la implementación de un algoritmo. Por ello, en particular, los argumentos de Fetzer de 1988 resultan tan (poco) destructivos para el "*testing*" como para la verificación formal. Esto sugiere que, para evaluar los méritos relativos del "*testing*" y la verificación formal, se debe prestar mayor atención a algunos otros presuntos problemas para la verificación formal planteadas en la literatura. Éstos son independientes de los problemas que insinua Fetzer en su artículo de 1988 y se describirán en lo que sigue. Tales problemas mostrarían que, para la mayoría de los programas computacionales "reales", la verificación formal resulta inviable y que, por ende, la alternativa es la utilización de métodos empíricos más simples como el "*testing*". Casi todos estos problemas aparecen entremezclados con los argumentos a partir de las prácticas sociales en De Millo et al (1979) y han sido recuperados,

redescubiertos y/o (re)elaborados por otros autores en comentarios posteriores al debate generado por el artículo de Fetzer.

Especialmente importante es el hecho de que a los argumentos de Fetzer se le puede añadir un problema más grave aún para la verificación de programas. Éste es que las pruebas de corrección de los programas son relativas a las especificaciones, ignorándose el problema de la formulación y la adecuación de las especificaciones en las definiciones formales de corrección. Como todo modelo, las especificaciones son incompletas y esto provoca que los programas basados en las mismas, a pesar de haber sido verificados, puedan realizar acciones inesperadas ((De Millo et al 1979, Pág. 275), (Smith 2003), (Blanco et al 2008)). Desde este punto de vista, uno puede preguntarse respecto a la utilidad de verificar formalmente un programa, cuya especificaciones probablemente contienen errores. En particular, en De Millo et al (1979) y Eden (2007) se argumenta que la formalización de las especificaciones puede, incluso, introducir una mayor cantidad de errores.

Otro problema es que los “verificacionistas” parecen suponer, en general, que las especificaciones son relativamente estables y muy distintas de los programas. Pero, en general, la formulación de las especificaciones está sujeta a los mismos tipos de errores que los programas y, habitualmente, hay un proceso de ajuste entre las especificaciones y los programas en base a la experiencia que se obtiene con el programa ((De Millo et al, Pág. 275), (Smith, Pág. 285)). Incluso, las verificaciones formales son díficiles de “reciclar”: una vez cambiadas las especificaciones, debe verificarse el programa desde cero y esto es bastante costoso si se cambian las especificaciones de manera continua.

Especialmente interesante es el hecho de que en la literatura sobre el problema de la verificación de los programas se ha ido decantando una diferenciación entre “corrección de programas” y “corrección de sistemas computacionales” (Barwise 1989, Pág. 7) la cual también aparece implícita en la argumentación inicial de DeMillo, Lipton y Perlis. Los sistemas computacionales tienen la particularidad de que suelen ser complejos, están embebidos en el mundo, realizan acciones e interactúan con usuarios. En el caso de estos tipos de programas computacionales, algunos de los argumentos en contra de la posibilidad de verificación de los programas anteriores aparecen con mayor fuerza y, además, se suman algunos nuevos:

- Las especificaciones para los sistemas computacionales son muy complejas (De Millo et al 1979, Pág. 275) y, dados los errores que pueden aparecer en esta fase y por todos los siguientes argumentos, es improbable que resulte en un beneficio el formalizarlas (De Millo et al 1979), (Barwise 1989), (Eden 2007)).

- Los sistemas computacionales, como están embebidos en el mundo, dependen de manera esencial de los modelos que subyacen a su construcción y es imposible, por ende, garantizar la completitud de las especificaciones de estos sistemas (Smith 2003), (Barwise 1989).
- En particular, la interacción (en especial, con humanos) hace que estos sistemas involucren un nivel de impredecidibilidad que no se encuentra en los "programas" (Smith 2003).
- Por su complejidad, los sistemas computacionales siempre están construidos en forma de capas. Esto aumenta considerablemente el problema de verificar que estos sistemas hagan lo que deben hacer. Puede haber fallas en todos los niveles (Smith, 2003).
- Los sistemas computacionales son dinámicos: como están embebidos en sistemas sociales complejos, deben cambiar constantemente para reflejar los cambios en estos sistemas (Smith 2003, Pág. 277).
- Aunque hay avances, en el estado del arte actual de la verificación de programas estas técnicas se aplican a programas más bien simples. Aunque es erróneo suponer que la complejidad de un programa computacional se puede medir como función lineal de su longitud, el estado actual del arte de la verificación de programas es que estas técnicas se aplican a programas de no más de cientos de líneas. Por el otro lado, los sistemas computacionales pueden llegar a tener hasta 10,000,000 líneas de código (Smith 2003). No hay ninguna razón *a priori* para suponer que las estrategias que son aplicables a un nivel se apliquen a otro: "cuidar a un niño dormido por una hora no escala a criar una familia de diez" (De Millo et al 1979, Pág. 278), o alternativamente: "compara la diferencia entre resolver una disputa entre dos personas y encontrar una solución a los problemas políticos de Medio Oriente" (Smith 2003, Pág. 277).

Aunque conviene distinguir entre los diversos problemas para la verificación formal identificados en la literatura reciente, la mayoría de estos problemas, como se indica en Eden (2007), están relacionadas con diversas formas de "complejidad" de los sistemas computacionales actuales, ya sea en lo que se refiere a las especificaciones, los sistemas mismos o al desarrollo y mantenimiento de tales sistemas. Sin embargo, debe tenerse presente, en contra de la caracterización que se hace en Eden (2007) de este problema como un problema esencialmente práctico, que muchos de los autores que denuncian estos problemas consideran que, por el contrario, hay una diferencia ontológica entre los "programas" y los "sistemas computacionales" (ver, por ejemplo, (De Millo, et al 1979, Pág. 275)). Para tales autores, además, como se indica en el último argumento enumerado, los métodos de la verificación formal no "escalan" a los sistemas complejos.

Por otra parte, en una ciencia en la que la forma en que se diseña los sistemas tiene una influencia determinante sobre la "naturaleza" de estos, la pregunta fundamental central no debería ser tanto si existen sistemas altamante complejos y, en muchos casos, caóticos, sino si pueden diseñarse sistemas que tengan la misma funcionalidad pero de modo tal que sea posible predecir el comportamiento de estos programas con cierto grado de confianza. En general, esto se traduce en la cuestión de si pueden construirse tales sistemas complejos utilizando combinaciones de programas ("módulos") simples, y de modo tal que la complejidad del sistema pueda derivarse de la complejidad de las partes.

De acuerdo a esto último, varios autores consideran que se puede realizar una distinción entre dos tipos de "complejidad" de los sistemas computacionales, cuyo comportamiento también resulta más o menos predecible. Por ejemplo, en Fetzer (1988, Pág. 1055) se distingue entre lo que Fetzer llama "*cummulative complexity*" y "*patchwork complexity*" y en (Arkoudas 2007, Pág. 199) se realiza una distinción similar en términos de "*a fox view of computer programs*" y "*hedgehog view of computers*". En Van Roy et al (2004) se realiza un iluminador analísis de las relaciones entre los diversos paradigmas de lenguajes de programación y de su evolución en función de las necesidades de la construcción de software, el cual sugiere que muchos de los sistemas se ajustan a una visión intermedia entre ambos tipos de "complejidades". Según este analísis, la evolución de los lenguajes de programación puede explicarse, a grandes rasgos, como un ajuste a las (diversas) necesidades de la construcción de software en función de ciertas dimensiones de "expresividad" de los lenguajes de programación de las cuales las principales son si tienen estado explícito o no, si el estado es no-determinista o determinista, y si son secuenciales o admiten concurrencia. El grado de "expresividad" de los paradigmas de programación es, según este punto de vista, también, inversamente proporcional a la facilidad con la que se puede razonar sobre los programas escritos en los lenguajes que soportan estos paradigmas, estando la mayoría de los lenguajes de programación usados en la industria (como Java y C++) en la zona epistémicamente más opaca de la taxonomía de paradigmas de programación que presentan los autores. Sin embargo, estos autores proponen también que la mayoría de los paradigmas de programación complejos pueden ser vistos como poseedores de un diseño en capas, lo cual posibilita así un diseño de los programas en el cual los constructos más transparentes epistémicamente (aquellos más declarativos) pueden aislarse de aquellos menos transparentes, facilitando así el razonamiento sobre los programas.

Por otra parte, para cada uno de los argumentos que pondrían en jaque el uso de la verificación formal arriba existe un contra-argumento que favorece el uso de la verificación. En primer lugar, y como quedará más claro en la siguiente sección, la explosión en la combinatoria de estados posibles

de un sistema computacional moderadamente complejo limita considerablemente el alcance de un enfoque, como el "*testing*", que consiste en probar lo que sucede en un subconjunto muy reducido de estados posibles. En segundo lugar, aunque la formalización de las especificaciones puede ser díficil, el grado de precisión alcanzado puede compensar los costos involucrados e incluso ayudar a testear la adecuación de las especificaciones a las necesidades de los usuarios de los programas. La formalización de las especificaciones, de hecho, posibilita el uso de varias técnicas automáticas para explorarlas. Además, se puede argumentar que la formalización de las especificaciones, contra lo indicado arriba, puede ayudar a identificar errores. Finalmente, aunque probablemente sea imposible formalizar las dimensiones sociales, estéticas, etc. de los sistemas computacionales, las dificultades involucradas con el "cambio" de las especificaciones pueden solucionarse mediante distintos grados de automatización de la verificación formal[6] . En todo caso, debe tenerse en cuenta los avances realizados en el desarrollo de métodos formales para determinar la "corrección" de los programas. Como se muestra en un "*survey*" reciente sobre el uso de métodos formales en la industria (Woodcock et al 2009), ya existen actualmente proyectos de hasta 1000000 líneas de código que han sido verificados formalmente.

6. Hacia un marco ontológico y epistemológico alternativo para una metodología de la verificación de los programas computacionales

Se espera haber mostrado, en el presente trabajo, que la diferencia entre "*testing*" y verificación formal no se encuentra en que el "objeto" al que se aplican estas metodologías (alternativamente, el "nivel" en que se puede estudiar un programa computacional) sea distinto y que, por ende, la evaluación de los méritos relativos de cada una de estas metodologías puede proceder con relativa independencia de la consideración de los problemas que se esbozan en Fetzer (1988) y se intentaron precisar en algunos puntos aquí. En particular, en la sección anterior se señaló el rol fundamental que tienen los problemas relacionados con la "complejidad" de los sistemas computacionales y, en especial, de los "costos" de aplicar tales metodologías en el contexto de la investigación de los sistemas "complejos", en la ponderación de los méritos relativos que tienen el "*testing*" y la verificación formal.

En nuestra discusión sobre la relación entre algunas formas de experimentación en ciencia de la computación, indicamos que la verificación formal y tales formas de experimentación pueden ser vistas como actividades complementarias destinadas a satisfacer diversos objetivos metodológicos. En el caso del debate respecto del "*testing*" y la verificación formal, el atender a los diversos objetivos, limitaciones prácticas, costos, etc. que se hallan en los distintos contextos en que se debe decidir

[6]Hoare (2003) arguye, por ejemplo, que un compilador que verifique el código que compila es una posibilidad real dado el estado de arte actual alcanzado en el desarrollo de métodos formales en ciencia de la computación.

respecto del uso de estas metodologías puede, también, evitar discusiones fútiles. En todo caso, mientras que el "ideal verificacionista" en que todos los componentes de un sistema complejo de software no se encuentren verificados no haya sido realizado (o no hagan falta, por las necesidades del contexto de aplicación, garantías muy fuertes de "corrección"), es inevitable el uso de una mezcla de "*testing*" y verificación formal para determinar la "corrección" de los programas computacionales[7].

El marco teórico desarrollado en Gaudel (1995) establece un camino para elaborar en mayor detalle algunas de las relaciones y diferencias entre "*testing"* y verificación formal y sugiere, también, una forma de encarar la estructuración y justificación de una metodología "mixta" en que la verificación formal y el "*testing*" pueden ser vistos como metodologías complementarias, más que antagónicas (Bogdanov et al, 2009). En este marco, el "*testing*" exhaustivo del sistema bajo estudio y la verificación formal del sistema entero son dos formas de arribar a la misma situación epistémica ideal: la garantía de que el programa es correcto (en el nivel de implementación del programa que se este estudiando y en relación a las especificaciones del programa). Sin embargo, tanto la verificación formal de todo el sistema como el "*testing*" exhaustivo del mismo son procedimientos cuya complejidad aumenta en función de la complejidad de las especificaciones[8] y esto hace que ambos enfoques para determinar la "corrección" de los programas comptuacionales sean en muchos casos díficles, sino imposibles, de llevar a cabo en la práctica.

Mientras que una solución parcial a este dilema en el caso de la verificación formal puede ser el de introducir supuestos que simplifiquen la verificación[9], la complejidad del "*testing*" puede reducirse, como se vio arriba, seleccionando "casos de *tests*" de acuerdo a las siguientes criterios contrapuestos: 1) se debe elegir "casos de *tests"* que revelen la mayor cantidad de errores del programa, 2) se debe elegir la cantidad mínima posible de "casos de *tests*". La actividad de selección de casos de "*tests*" se ve beneficiada si existe una especificación formal para el programa y esto puede permitir también, entre otras cosas, la generación automática de "casos de *test"*.

La selección de los "casos de *test"* está basada en ciertas hipótesis como, por ejemplo, que los valores seleccionados para realizar el "*testing*" sean representativos de todos los valores posibles con que se puede correr el programa o que el resultado de atravesar los diversos caminos de ejecución posibles del programa no cambia sustancialmente si se cambian los datos que se eligen para realizar

---

[7] Incluso en el escenario ideal del "verificacionista" podría recomendarse el uso del "*testing*" con fines de aumentar la confianza en que la verificación formal no contenga errores o no se hayan pasado por alto algún aspecto de la "corrección" del sistema y, de hecho, podrían diseñarse "casos de *tests*" dirigidos a esta tarea subsidiaria a la verificación formal. Relacionado con esto, en un escenario menos ideal las verificaciones formales muchas veces contienen simplificaciones y podría concebirse de "*tests*" que revelan errores en tales supuestos simplificadores.

[8] La complejidad no se reflejará necesariamente en la forma sintáctica de las especificaciones.

[9] Ver nota al pie de página 7.

el recorrido. El que estas hipótesis no hayan sido demostradas es lo que hace a la naturaleza heurística del "*testing*". Por otra parte, si tales hipótesis fuesen probadas se reestablecería la situación epistémica ideal que implica el "*testing*" exhaustivo o la verificación formal completa del sistema en cuestión. Esto sugiere que un estudio como el que se realiza en Gaudel (1995) de las diversas hipótesis implícitas en la selección de "*tests*" y las maneras en que se podría hacer para probar tales hipótesis puede ser una de las formas en que se podría establecer una metodología mixta para determinar la "corrección" de los programas computacionales.

La visión anterior, en conjunto, también, con un analísis de las diversas fases de refinamiento de la implementación de un sistema computacional (y la relación entre estas) y de los objetivos y limitaciones prácticas que surgen en los diversos contextos de diseño de software, ofrece una perspectiva más rica desde la cual elaborar y evaluar diversas estrategías metodológicas para determinar la "corrección" de los programas computacionales (en ciertos contextos y según ciertos objetivos y restricciones prácticas). Tal perspectiva pone en duda, también, a las divisiones de las comunidades de investigación en ciencia de la computación en términos de sus supuestos ontológicos, epistemológicos y metodológicos a las que se aludió en la introducción de este trabajo. Sugiere, más bien, que, si existen tales comunidades bien diferenciadas, esto se deba, probablemente, a causas ajenas a los fundamentos ontológicos y epistemológicos que subyacen a las metodologías que emplean los investigadores que componen a las mismas.

Referencias

Arkoudas, K; Bringsjord, S. Computers, Justification, and Mathematical Knowledge. Minds & Machines, 17: 185-202, 2007

Barwise, J. Mathematical Proofs of Computer System Correctness. Notices of the American Mathematical Society, 36 (7): 844-851, 1989

Blanco, J; García, P. A Categorial Mistake in the Formal Verification Debate. En European Conference on Computing and Philosophy (ECAP), 2008

K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, R. M. Hierons, K. Kapoor, P. Krause, G. Luettgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan . Working together: Formal Methods and Testing ACM Computing Surveys 41(4): 1–36, 2009

Colburn, T.R. Software, Abstraction, and Ontology. The Monist, 82 (1): 3-19, 1999

Colburn, T.R. Methodology of Computer Science. En L. Floridi (ed.), The Blackwell guide to the philosophy of computing and information. Cornwall, UK: Blackwell Publishing, 2004

Colburn Decoupling as a Fundamental Value in Computer Science, Minds and Machines, 21: 241-259, 2011

De Millo, R D; Lipton, R; Perlis, A Social Processes and Proofs of Theorems and Programs. Communications of the ACM, 22 (5):271-280, 1979

Eden, A H. Three Paradigms of Computer Science. Minds & Machines, 17: 135-167, 2007

Fetzer, JH. Program verification: The very idea. Communications of the ACM, 37 (9): 1048-1063, 1988

Fetzer, JH. Technical Correspondence, Communications of the ACM, 34 (3): 374-381, 1989

Fetzer, JH. The Role of Models in Computer Science, The Monist, 82(1):20-36, 2009

Gaudel, MC. Testing can be Formal Too. Lecture Notes in Computer Science, 915: 82-96, 1995

Hoare, CAR. An Axiomatic Basis for Computer Programming. Communications of the ACM, 12(3): 335-355, 1969

Hoare, CAR. The Mathematics of Programming. Byte, 8(11): 115-130, 1986

Hoare, CAR. The Verifying Compiler: A Grand Challenge for Computer Science Research. Communications of the ACM 50(1): 63-69, 2003.

Jalote, P. An Integrated Approach to Software Engineering. Springer, 2010

Leroy, X. Formal Verification of a Realistic Compiler. Communications of the ACM 52(7): 107-115, 2009

Partridge, D. Non-Programmed Computation. Communications of the ACM, 43: 293-302, 2000

Scheutz, M. When Physical Systems Realize Functions. Minds and Machines, 9: 161-196, 1999

Simon, H.A. The Sciences of the Artificial. The MIT Press, 1996

Smith, BC. Limits of Correctness in Computers. En Timothy R. Colburn et al. (ed) Program Verification, Kluwer Academia Publishers, 2003

Turner, R.; Eden, A.H. "The Philosophy of Computer Science", *The Stanford Encyclopedia of Philosophy* (Edición 2008), Edward N. Zalta (ed.), URL = < http://plato.stanford.edu/entries/computer-science/>

Van Roy, P; Haridi, S., Concepts, Techniques, and Models of Computer Programming. The MIT Press, 2004

Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal methods: Practice and experience. ACM Computing Surveys 41(4): 1–36, 2009