Method to identify corrections of defects on product line models

Luisa Rincón *, Gloria Giraldo †, Raúl Mazo ‡, Camille Salinesi ‡ and Daniel Diaz ‡
*Departamento de Electrónica y Ciencias de la Computación,
Pontificia Universidad Javeriana, Cali, Colombia
†Departamento de Ciencias de la Computación y la Decisión,
Universidad Nacional de Colombia, Medellín, Colombia
‡CRI,Universidad Paris 1 Panthéon Sorbonne, París, Francia

***Abstract*—Software product line engineering is a promising paradigm for developing software intensive systems. Among their proven benefits are reduced time to market, better asset reuse and improved software quality. To be successful, software product line engineering represents the collection of products of the product line by means of product line models. Feature Models (FMs) are a common way to represent product lines by expressing the set of feature combinations that software products can have. However, these models might have defects. Defects in FMs might be inheriting to the products configured from these models. Consequently, defects must be early identified and corrected. In scientific literature, several works deal with identification of defects in FMs. However, only few of these proposals are able to explain how to fix found defects, and they only identify some corrections. This paper proposes a new method to detect all possible corrections that can be found when the method systematically eliminates features from the FMs. In particular, the proposed method was applied on 78 distinct FMs with sizes up to 120 dependencies. Preliminary evaluation indicates that the method proposed in this paper is accurate and potentially useful in real scenarios.**

***Keywords*—Software product lines, Features Models, Corrections, Defects, Software Engineering**

I. Introducción

La ingeniería de líneas de productos, es un prometedor paradigma que permite administrar eficientemente un conjunto de productos con elementos comunes y variables que pertenecen a un dominio en particular. Este paradigma ofrece beneficios como la reutilización, la disminución de errores y la disminución de tiempos y costos de producción [1]. Los beneficios propuestos para las Líneas de Productos (LP) pueden ser extensibles al software, pues en el desarrollo de software es necesario administrar la reutilización y la variabilidad. Las líneas de productos aplicadas al software se conocen como Líneas de Productos de Software (LPS) [2].

Todos los productos que se derivan a partir de una LP, pueden representarse de manera intesiva con modelos. Una de las notaciones que existen para expresar esos modelos de LP es la notación de modelos de características. Los modelos de características se diseñan en una de las primeras etapas del desarrollo de la LP y son un insumo importante para identificar los elementos comunes y variables que contendrá la LP [3]. En un modelo de características, cada característica corresponde a un elemento distintivo para el usuario. Así, estos modelos son un punto de comunicación intermedia entre los usuarios finales y los desarrolladores de la LP [3], ya que permiten, por ejemplo, identificar cuáles son las características comunes a todos los productos y cuáles son las características variables; cuáles características deben ir juntas en todos los productos y cuáles características no pueden ir al mismo tiempo en un producto [3].

A medida que aumenta la complejidad de los modelos de características, es posible que quienes los elaboran introduzcan en ellos, sin intención, defectos semánticos. Los defectos semánticos son imperfecciones que afectan la capacidad del modelo de características para representar de manera correcta el dominio que se está modelando [4].

Diferentes trabajos se proponen en la literatura para identificar automáticamente defectos semánticos en los modelos de características [4]–[13]. Sin embargo, pocos de ellos proponen cómo podría corregirse cada defecto, y los que lo hacen [9]–[13], identifican solamente algunas de las correcciones disponibles. La corrección de los modelos de características depende entonces de la experiencia y habilidad del diseñador del modelo, quien debe decidir qué cambios del modelo corregirían cada defecto. Sin embargo, hacer esta operación manualmente es tan complicado como encontrar el defecto mismo, ya que a medida que aumenta la complejidad del modelo de características, el número de dependencias y la transitividad entre ellas, hacen que la corrección se vuelva una tarea lenta y propensa a errores [6], [7], [11].

Nuestro objetivo general es encontrar una técnica genérica para identificar las causas y correcciones de diferentes defectos de los modelos de líneas de productos expresados en diferentes notaciones. En este artículo proponemos un trabajo que contribuye a ese objetivo, al presentar un método que no sólo identifica defectos semánticos de los modelos características, sino que también identifica para cada defecto posibles correcciones que lo solucionarían. Concretamente, las contribuciones de este trabajo son las siguientes:

1) Se desarrolló un método para identificar defectos semánticos en los modelos de características y sus correcciones.
2) Se aprovechó en el área de la ingeniería de líneas de productos el concepto de los *Subconjuntos Mínimos de Corrección (MCSes)* para identificar correcciones a los modelos de características con defectos. La identificación de MCSes se utiliza en la verificación de diseños de hardware [14], y en el diagnóstico de

circuitos [15]. Sin embargo, de acuerdo a la revisión realizada de la literatura, no se encontraron evidencias del uso de este concepto para identificar correcciones en modelos de características.

3) Se creó una herramienta que implementa el método propuesto.
4) Se aplicó exitosamente el método propuesto en un caso de estudio, en modelos de características tomados de la literatura y en modelos de características generados automáticamente con BeTTy [16]. Los resultados indican que nuestra propuesta es exacta y potencialmente útil en escenarios reales.

Este artículo se organiza de la siguiente manera: en la Sección II se presenta el marco conceptual de este artículo y en la Sección III se presenta el método propuesto. En la Sección IV se presentan algunos detalles de la implementación y en la Sección V se presenta la evaluación preliminar del método propuesto. Luego, en la Sección VI se presentan los trabajos relacionados y finalmente, en la Sección VII se presentan las conclusiones y el trabajo futuro.

II. Conceptos Preliminares

A. Modelos de Características

Las características de los modelos de características se relacionan entre sí con dependencias que pueden ser *obligatorias*, *opcionales*, *de cardinalidad grupal*, de *inclusión* o de *exclusión*. En esta subsección se explican estas dependencias, de acuerdo a las definiciones presentadas por Czarnecki *et al.* [18]. Para ello, se toman como ejemplo las dependencias del modelo de características de la Fig. 1. Este modelo de características se presentará en detalle en la Subsección II-C.

- **Obligatoria:** Se representa con una línea cuyo origen es la característica padre y cuyo destino, formado por un círculo oscuro, es la característica hija. Esta dependencia indica que la característica hija debe ser incluida en los productos que tengan seleccionada su característica padre y viceversa. La dependencia *D1* de la Fig. 1 es obligatoria, por lo tanto la característica *Rendimiento* debe ser incluida en todos los productos que tengan la característica *Sitio Web* y viceversa. Otras dependencias obligatorias del modelo de la Fig. 1 son: *D3, D7, D8, D9, D11 y D15*.
- **Opcional:** Se representa con una línea cuyo origen es la característica padre y cuyo destino, formado por un círculo blanco, es la característica hija. Esta dependencia indica que la característica hija puede ser o no incluida en los productos que tengan su característica padre. Si la característica hija es incluida en un producto, su característica padre también debe ser incluida en el mismo producto. La dependencia *D2* de la Fig. 1 es opcional, por lo tanto la característica *Servicios Adicionales* puede o no ser incluida en los productos que tengan la característica *Sitio Web*. Otras dependencias opcionales del modelo de la Fig. 1 son *D5, D6, D10, D12, D13 y D14*.
- **Cardinalidad grupal:** Representa el mínimo y máximo número de características que un producto puede tener cuando la característica padre es incluida en un producto. Si al menos una característica hija es incluida en un producto, la característica padre también debe ser incluida. La dependencia *D18* de la Fig. 1 es una cardinalidad grupal, la cual requiere mínimo una de sus tres características hijas y máximo las tres características hijas. Otras cardinalidades grupales del modelo de la Fig. 1 son: *D4, D16 y D17*.
- **Requerida:** Se representa con una flecha punteada unidireccional. El origen de la fecha es la característica que requiere, y el destino es la característica requerida. Esta dependencia indica que la característica requerida debe ser incluida en todos los productos que tengan la característica que la requiere. La dependencia *D19* de la Fig. 1 es una dependencia de tipo requerida, por lo tanto la característica *Segundos* es requerida por la característica *Búsquedas*. Otras dependencias requeridas del modelo de la Fig. 1 son *D20, D21, D23, D24, D25, D26 y D27*.
- **Exclusión:** Se representa con una flecha punteada bidireccional que relaciona dos características. Esta dependencia indica que las características relacionadas no pueden estar juntas en ningún producto derivado de la LP. La dependencia *D22* de la Fig. 1 es una dependencia de tipo exclusión, por lo tanto las características *Milisegundos* y *HTTPS* no pueden estar simultáneamente en un mismo producto derivado de la LP.

Las dependencias de tipo obligatorio y opcional son conocidas como dependencias estructurales, mientras que las dependencias de tipo requerido y exclusión son conocidas como dependencias transversales [18].

B. Defectos Semánticos en Modelos de Características

La semántica de un modelo de características corresponde al conjunto de productos válidos que pueden derivarse a partir del modelo de LP [18]. En este artículo, nos interesamos en los defectos semánticos de los modelos de características. Estos defectos son imperfecciones que afectan la capacidad del modelo de características, para representar de manera correcta el dominio que se está modelando [4]. A continuación, se presentan los defectos semánticos más comunes que se pueden encontrar en los modelos de características. En adelante, en este artículo, nos referiremos a los defectos semánticos, simplemente como defectos.

- **Modelos vacíos:** no puede derivarse ningún producto válido del modelo de características [4], [8], [11], [19].
- **Falso modelo de línea de productos:** sólo es posible derivar un producto válido del modelo de características [4].
- **Características muertas:** son características que aunque están consideradas en el modelo de características, no pueden estar presentes en ningún producto válido derivado del modelo de características [4], [8].
- **Características falsas opcionales:** son características representadas como opcionales en el modelo, pero que deben ser incluidas en cualquier producto válido derivado del modelo de características [4], [8], [11].
- **Redundancias:** son dependencias que no modifican la semántica del modelo de características y se dan

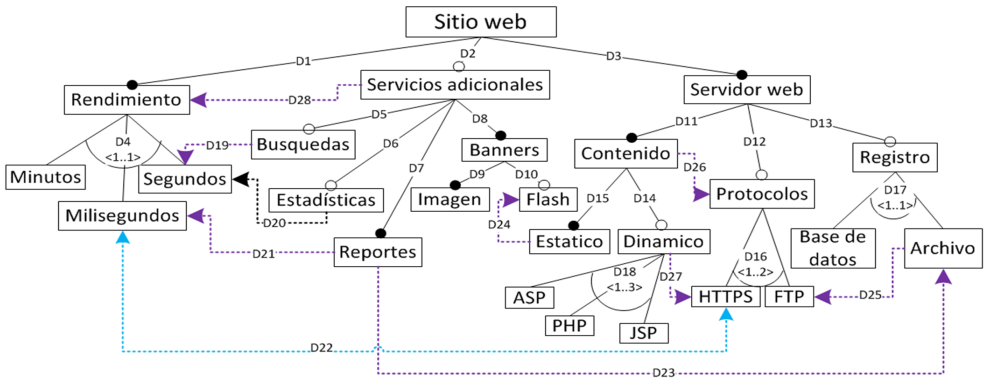

Figura 1: Modelo de características de sitios web. Versión adaptada del modelo propuesto por Mendoça *et al.* [17]

cuando éste tiene la misma información modelada de diferentes maneras [4], [8]. Si bien pueden darse casos en los que el diseñador del modelo de características introduce a propósito las redundancias, en este trabajo, cualquier redundancia identificada se considera un defecto, puesto que las redundancias constituyen un problema para la evolución de los modelos [4]. Determinar cuándo una redundancia es un defecto y cuándo no es un defecto, es un trabajo futuro de está investigación.

C. Ejemplo de aplicación

Una línea de productos de sitios web, es una solución para desarrollar rápidamente sitios web a partir de características comunes y variables a los sitios web. La Fig. 1 presenta una versión adaptada del modelo de características de sitios web propuesto por Mendoça *et al.* [17]. Para ilustrar nuestra propuesta, simplificamos el modelo original y le introdujimos 10 características muertas, 7 falsas características opcionales, 3 redundancias, y lo volvimos un falso modelo de líneas de productos. Por razones de espacio, en este artículo únicamente se hará referencia a la característica *ASP*, que es una de las características muertas introducidas en el modelo de características. Esta característica es muerta, entre otras cosas, porque al seleccionar *ASP* en un producto derivado de la línea de productos, debe ser seleccionada también su característica padre *Dinámico*, pero *Dinámico* es también una característica muerta. Una explicación más detallada de este defecto se presenta en la Subsección III-C. En Internet [1] se encuentra disponible la lista de defectos introducidos en el modelo y las correcciones identificadas por el método propuesto para estos defectos.

Nuestra adaptación del modelo de características de sitios web tiene tres características importantes: el *Rendimiento*, los *Servicios adicionales* y el *Servidor Web* (ver Fig. 1). El *Rendimiento*, puede estar dado en *Milisegundos*, *Segundos* o *Minutos*, según la característica que sea seleccionada. Los *Servicios adicionales* disponibles son: *Búsquedas* de información en el sitio, *Estadísticas* de visitas, *Reportes* y *Banners*. Los *Banners* son siempre *Imágenes* y opcionalmente animaciones en *Flash*. Si un sitio web tiene *Servicios adicionales*, entonces también tendrá *Reportes* y *Banners*. En cuanto a la característica *Servidor web*, ésta agrupa el *Contenido* del sitio web, los *Protocolos* de transferencia y la capacidad de *Registro*. El *Contenido* es *Estático*, sin embargo, los sitios web derivados, de manera opcional pueden tener contenido *Dinámico* en *PHP*, *JSP* o *ASP*. En cuanto a los *Protocolos* de transferencia, el modelo de características cuenta con los protocolos *HTTPS* y *FTP*. En una configuración, pueden seleccionarse algunos de los dos protocolos o ambos al mismo tiempo. Opcionalmente, los sitios web derivados de esta LP soportan el *Registro* de información (*Logs*), lo cual es útil para auditar las acciones de los usuarios en el sitio web. En este caso, el registro es almacenado en *Archivos* externos o dentro de una *Base de datos*.

Existen otras dependencias que limitan las combinaciones de características que generan productos válidos en este modelo de características. Por ejemplo, cuando un sitio web soporta el *Registro* de información en *Archivos*, entonces requiere el protocolo *FTP* (ver dependencia *D25*); y no es posible que un sitio web tenga un *Rendimiento* en *Milisegundos*, si soporta el protocolo *HTTPS* (ver dependencia *D22*).

D. Programación de Restricciones

La programación de restricciones es un paradigma de programación para la resolución de problemas combinatorios y de optimización. En este paradigma, las relaciones entre las variables son expresadas en términos de restricciones y cada variable tiene valores de dominio que indican sus posibles valores. La programación de restricciones se ha aplicado con éxito en ámbitos como la planificación, la configuración de redes, y la bioinformática [20].

Los problemas que se resuelven mediante la programación de restricciones pueden modelarse como problemas de satisfacción de restricciones (CSP por su nombre en inglés *Constraint Satisfaction Problem*). Resolver un CSP, es encontrar una solución en la que todas las variables tienen asignado un valor del dominio y se satisfacen todas las restricciones del problema [21]. Cuando esta asignación no es posible, entonces el CSP no tiene solución y el programa de restricciones que representa al CSP es irresoluble, en caso contrario el programa de restricciones es resoluble [20]. Los programas de

[1] https://github.com/lufe089/CLEI2014

restricciones se resuelven con un *solver*[2]. GNU Prolog [23] y SWI Prolog [24], son dos de los *solvers* disponibles para resolver programas de restricciones que representan CSPs.

E. Subconjuntos Mínimos de Corrección (MCSes)

Los Subconjuntos Mínimos de Corrección, en inglés *Minimal Correction Subsets* (MCSes), son utilizados para identificar correcciones a programas de restricciones irresolubles [25]–[28]. De manera informal, un MCS es un subconjunto de restricciones de un programa de restricciones irresoluble, que al ser eliminado hace que el programa de restricciones se vuelva resoluble. Si existe más de un subconjunto de elementos que pueden eliminarse para corregir el programa irresoluble, entonces existe más de un MCS [26].

Formalmente,

$$\begin{gathered}\text{Dado un programa de restricciones irresoluble } \alpha \\ M \subseteq \alpha \text{ es un MCS} \equiv \alpha - M \text{ es irresoluble} \wedge \\ \forall Ci \in M,\ \alpha - (M - \{Ci\}), \text{ es resoluble}\end{gathered}$$

Cada MCS es mínimo, ya que todos sus elementos deben ser eliminados del programa irresoluble para volverlo resoluble. Lo anterior garantiza que cada MCS contiene sólo restricciones relevantes para corregir el programa de restricciones [26]. Sin embargo, aunque los MCSes son mínimos, esto no quiere decir que deban tener un número máximo de elementos. Por ejemplo, como se explicará en la Subsección III-C, para que la característica *ASP* de la Fig. 1 no sea muerta se identificaron MCSes con uno, dos y tres elementos.

III. Propuesta

En esta sección se describe el método propuesto para identificar los defectos presentados en la Subsección II-B y sus respectivas correcciones. Estas correcciones corresponden a los subconjuntos mínimos de dependencias que deben ser eliminados del modelo de características para solucionar al menos un defecto (MCSes). Para ello, el método recibe como entrada un modelo de características y luego realiza tres pasos: transformar el modelo de características (Subsección III-A), identificar en el modelo de características los defectos considerados en este artículo (Subsección II-B) y finalmente, identificar las correcciones de cada defecto (Subsección III-C). Este último paso es nuestra principal contribución. Para terminar, los defectos del modelo y sus respectivas correcciones se presentan al diseñador.

A. Paso 1. Transformar Modelos de Características

La representación gráfica de los modelos de características no permite razonar sobre ellos. Por esta razón, para analizar los modelos de características, éstos deben ser transformados a algún lenguaje que pueda se analizado por un computador, como por ejemplo la programación de restricciones [29]. En un programa de restricciones que representa un modelo de características, las variables son las características del modelo y, en nuestro caso, tienen un dominio definido de uno ("1") o cero ("0"), mientras que las restricciones son las dependencias del modelo de características [29].

Durante este primer paso del método, el modelo de características de entrada es transformado en un nuevo modelo que llamamos *modelo transformado*. El *modelo transformado* sirve para construir, en los siguientes pasos del método, programas de restricciones a partir del modelo de características de entrada. En particular, en el *modelo transformado* se almacenan las características y las dependencias del modelo de características. Además, para cada dependencia se almacena su representación en términos de restricciones (obtenida al aplicar las reglas de transformación propuestas por Mazo *et al.* [29]) y una descripción de la dependencia en lenguaje natural.

B. Paso 2. Identificar Defectos en Modelos de Características

Salinesi y Mazo [4] proponen algoritmos para identificar defectos en modelos de LP representados como programas de restricciones. Estos defectos, corresponden a los defectos considerados en este artículo (ver Subsección II-B). Nuestro método transforma el modelo de características que se va a analizar en un programa de restricciones a partir del *modelo transformado* obtenido en el paso 1 (ver Subsección III-A). Luego, el método analiza el programa de restricciones resultante con los algoritmos propuestos por Salinesi y Mazo [4]. De esta manera al terminar este paso, el método ha identificado los defectos del modelo de características analizado y queda listo para identificar las correcciones de cada defecto.

C. Paso 3. Identificar Correcciones

El método propuesto en este artículo, toma de la programación de restricciones el concepto de los MCSes, y lo aplica en el contexto de las líneas de productos. De esta manera, identificar correcciones a los defectos de los modelos de características, corresponde a identificar los MCSes de un programa de restricciones irresoluble. En términos de los modelos de características, cada MCS es un subconjunto mínimo de dependencias, el cual debe ser eliminado de un modelo de características con el fin de corregir un defecto.

El concepto de los MCSes requiere de programas de restricciones irresolubles. Por esta razón, este paso toma como entrada el *modelo transformado* y un defecto a analizar del modelo de características. Luego, con esta información, el método crea un programa de restricciones irresoluble que es analizado para identificar sus MCSes, es decir las correcciones. Este paso se debe repetir mientras existan defectos del modelo de características, para los cuales no se hubieran identificado sus MCSes.

A continuación se explica cómo se lleva a cabo, en este tercer paso, la identificación de los MCSes. Para ello, primero se presentan los tipos de restricciones que se requieren para identificar MCSes según el tipo de defecto. Después, se presenta el algoritmo propuesto para identificar MCSes. Luego, se detalla cómo se obtienen las correcciones de cada defecto, al transformar los MCSes identificados de subconjuntos de restricciones, a subconjuntos de dependencias del modelo de características. Finalmente, se analizan las correcciones identificadas por el método para la característica muerta *ASP* del modelo de características de la Fig. 1.

[2]Un *solver* es un término genérico que indica una pieza de software matemático que 'resuelve' un problema matemático tomando la descripción del problema y luego calculando su solución [22].

1) Tipos de Restricciones: Para identificar las correcciones de cada defecto el Algoritmo 1, el cual se explicará más adelante, analiza sistemáticamente programas de restricciones irresolubles. Estos programas de restricciones están formados por la unión de tres tipos de restricciones que llamamos: *las restricciones del modelo, la restricción fija y la restricción de verificación*. A continuación se explica cada tipo de restricción.

- ***Restricciones del modelo:*** son las restricciones que representan las dependencias del modelo de características. El Algoritmo 1 elimina estas restricciones del modelo de características, para identificar los MCSes de cada defecto.
- ***Restricción fija:*** es una restricción que siempre debe incluirse en los programas de restricciones porque es requerida por la notación. Una restricción fija de los modelos de características es que la característica raíz debe ser seleccionada en cualquier producto derivado de la línea de productos [3].
- ***Restricción de verificación:*** es una restricción que se adiciona para representar el defecto que se analizará. Esta restricción sirve para volver irresoluble el programa de restricciones, puesto que un programa de restricciones irresoluble es el punto de partida para identificar las correcciones de un defecto del modelo de características.

2) Restricción de verificación por tipo de defecto: La restricción de verificación necesaria para identificar, con el Algoritmo 1, los MCSes cambia según el tipo de defecto así:

- **Modelo vacío:** en el método propuesto identificar correcciones para un modelo vacío, consiste en identificar las dependencias del modelo, que al eliminarse hacen que se puedan derivar productos del modelo de características. Para identificar las correcciones de este defecto, no es necesario adicionar ninguna *restricción de verificación*, pues cuando el modelo de características es vacío, el programa de restricciones formado por las *restricciones del modelo* y la *restricción fija* ya es irresoluble.
- **Características muertas y falsas características opcionales:** en el método propuesto, identificar correcciones para una característica muerta, es identificar las dependencias del modelo, que al eliminarse hacen que se puedan derivar productos que contengan esta característica. De igual manera, identificar correcciones para una falsa característica opcional, consiste en identificar las dependencias del modelo, que al eliminarse hacen que se puedan derivar productos sin esta característica (i.e., con la característica falsa opcional deseleccionada). Para ello, en el caso de las características muertas, se crea una *restricción de verificación* que obliga a que la característica muerta esté seleccionada en al menos un producto derivado del modelo de características. Por su parte, en el caso de las falsas características opcionales, se crea una *restricción de verificación* que obliga a que la característica falsa opcional no esté seleccionada en al menos un producto derivado del modelo de características. Por ejemplo, `ASP #=1`, es la restricción de verificación que adiciona el método, automáticiamente, para identificar las correcciones de la característica muerta *ASP* del modelo de características de la Fig. 1. Esta restricción obliga a que la característica *ASP* sea seleccionada en un producto derivado del modelo de características de la Fig. 1 y hace que la unión de las *restricciones del modelo, la restricción fija y la restricción de verificación* dé como resultado un programa de restricciones irresoluble.
- **Falso modelo de LP:** un modelo de LP es falso cuando no permite derivar más de un producto [4]. Esto puede darse porque el modelo de características no tiene variabilidad (todas sus dependencias son obligatorias), o porque el modelo tiene características falsas opcionales y/o características muertas. Cuando en un modelo de características se corrigen las características falsas opcionales y las características muertas, entonces pueden derivarse más productos de la LP. Por lo tanto, cuando un modelo de características es un falso modelo de LP, porque tiene características muertas o falsas opcionales, entonces corregir estos defectos también corrije los falsos modelos de LP. En cambio, si el modelo de características es un falso modelo de LP, porque todas sus dependencias son obligatorias, entonces en ese caso, el método propuesto no identifica ninguna corrección, pues consideramos que el modelo de características fue diseñado así intencionalmente.
- **Redundancias:** en el método propuesto, identificar correcciones para una dependencia redundante, consiste en identificar las dependencias del modelo de características que cuando son eliminadas, hacen que la dependencia redundante deje de serlo. Para identificar las correcciones de una dependencia redundante, se debe eliminar de las *restricciones del modelo* aquella que representa la dependencia redundante. Además, la *restricción de verificación*, debe representar la negación de la dependencia redundante a analizar. Así, la unión de las *restricciones del modelo, la restricción fija y la restricción de verificación*, da como resultado un programa de restricciones irresoluble.

3) Algoritmo para Identificar Subconjuntos Mínimos de Corrección: En este trabajo proponemos el Algoritmo 1 para identificar los MCSes de un programa de restricciones irresoluble. Este algoritmo identifica los MCSes quitando sistemáticamente *restricciones del modelo* de un programa de restricciones irresoluble, hasta que el programa de restricciones resultante tenga al menos una solución. Cada restricción que al eliminarse hace que el conjunto de restricciones restante sea resoluble pertenece a un MCS. Para encontrar todos los MCSes, primero se eliminan las *restricciones del modelo* de una en una, luego de dos en dos y así sucesivamente.

El Algoritmo 1 identifica los MCSes de un programa de restricciones irresoluble que representa un modelo de características con uno de sus defectos. Por lo tanto, este paso del método crea un programa de restricciones y ejecuta el Algoritmo 1 para cada uno de los defectos identificados en el paso 2 (ver Subsección III-B).

Las entradas del Algoritmo 1 son las restricciones del

modelo (rm), la restricción fija (rf), y la restricción de verificación (rv). Como salida, el algoritmo entrega la colección de todos los MCS identificados (MCSes) para el defecto analizado. La variable k representa el tamaño de los subconjuntos generados a partir de las *restricciones del modelo*. Dado que el algoritmo es incremental, k inicia en 1.

El algoritmo se ejecuta con un ciclo que se invoca mientras que la variable bandera continuar tenga asignado el valor 'verdadero' (línea 4). Esta condición se cumple mientras que existan subconjuntos candidatos a ser MCSes. El ciclo inicia invocando la función obtSubsetsCandidatos. Esta función se encarga de construir los subconjuntos candidatos a ser MCS, lo cual se explicará más adelante con un ejemplo. Esta función recibe tres entradas. La primera entrada es la variable k, que indica el tamaño de los subconjuntos que se generarán. La segunda entrada es el conjunto de las *restricciones del modelo* (rm), a partir del cual se identificarán los subconjuntos candidatos a ser MCSes. Finalmente, la tercera entrada es la colección de los MCSes identificados en ejecuciones anteriores. Estos subconjuntos corresponden a todos los subconjuntos de tamaño k (a excepción del conjunto vacío), que pueden formarse con las restricciones que pertenecen al conjunto (rm) y que no son super conjuntos de los (MCSes) identificados previamente.

Por ejemplo, para la característica muerta *ASP* del modelo de características de la Fig. 1, el algoritmo identifica un MCS de dos elementos (k=2), formado por las restricciones que representan las dependencias *D11* y *D14* (ver MCS10 Tabla I). Suponiendo que el algoritmo vaya a identificar los MCSes de tres elementos (k=3), la función obtSubsetsCandidatos obtiene los subconjuntos candidatos a ser MCSes. Estos son subconjuntos de 3 elementos que no contienen al mismo tiempo las restricciones correspondientes a las dependencias *D11* y *D14*. En términos del modelo de características, esto significa que al eliminar las dependencias *D11* y *D14* se corrige la característica muerta *ASP*. Por lo tanto, no tiene sentido evaluar si también sería una corrección, eliminar estas dos dependencias más otras dependencias del modelo, pues la corrección dejaría de ser mínima.

Una vez se han construido todos los subconjuntos candidatos a ser MCSs, el algoritmo almacena en la variable subsets la colección de subconjuntos resultantes y evalúa cada subconjunto candidatoMCS ∈ subsets de la siguiente manera:

El algoritmo crea un programa de restricciones irresoluble α, formado por la unión de las *restricciones del modelo* (rm), la *restricción fija* (rf) y la *restricción de verificación* (rv) (línea 8). Luego, el algoritmo elimina del programa de restricciones irresoluble α, las restricciones del subconjunto candidatoMCS y evalúa si el conjunto de restricciones resultante α' es resoluble (líneas 9 y 10). Sólo si el conjunto α' es resoluble, el subconjunto candidatoMCS es un MCS. En ese caso, se adiciona el conjunto candidatoMCS a la lista de MCSes (línea 11).

Una vez el algoritmo termina de evaluar todos los subconjuntos candidatos a ser MCSes de tamaño k, incrementa el valor de k en uno (línea 14) y repite nuevamente la creación y evaluación de subconjuntos candidatos (líneas 6-17). Cuando la función obtSubsetsCandidatos entrega un conjunto vacío, esto significa que no existen subconjuntos de tamaño k candidatos a ser MCSes. El algoritmo entonces asigna a la variable continuar el valor 'falso' (línea 16), retorna la colección de MCSes identificados (línea 19) y termina. Es importante resaltar que en cada iteración de los subconjuntos candidatos, el algoritmo construye nuevamente el conjunto α (línea 8). Esto se hace para que la identificación de cada MCS inicie siempre con el mismo programa de restricciones irresoluble, de manera que los MCS identificados sean independientes entre sí.

Para el caso de la característica muerta *ASP*, el Algoritmo 1 identifica MCSes de un elemento, dos elementos y tres elementos, mientras que no identifica MCSes de cuatro elementos o más (ver Tabla I). Esto quiere decir, que no existen MCSes con más de 3 elementos que sigan siendo mínimos para la característica muerta *ASP*.

Algoritmo 1 algoritmo propuesto para identificar MCSes

Entrada: rm:restric modelo,rf:restric fijas, rv:restric verificación
Salida: $MCSes$: Colección de MCSes

1: $k \leftarrow 1$ // Tamaño subconjuntos a analizar
2: $MCSes \leftarrow \emptyset$
3: $continuar \leftarrow$ **verdadero**
4: **mientras** ($continuar =$ **verdadero**) **hacer**
5: $subsets \leftarrow obtSubsetsCandidatos(k, rm, MCSes)$
6: **si** $subsets \neq \emptyset$ **entonces**
7: **para todo** $candidatoMCS \in subsets$ **hacer**
8: $\alpha \leftarrow rm \cup rf \cup rv$
9: α' $\leftarrow \alpha \backslash \{candidatoMCS\}$
10: **si** α' $es\ resoluble$ **entonces**
11: $MCSes \leftarrow MCSes \cup \{candidatoMCS\}$
12: **fin si**
13: **fin para todo**
14: $k \leftarrow k + 1$
15: **si no**
16: $continuar \leftarrow$ **falso**
17: **fin si**
18: **fin mientras**
19: **retornar** $MCSes$

Cuando el Algoritmo 1 identifica los MCSes de cada defecto, éstos se encuentran expresados en forma de restricciones, lo cual no es muy claro para el usuario final. La idea del método propuesto es que los diseñadores puedan entender las correcciones identificadas por el método, aún sin conocer la programación de restricciones. Por esta razón, al terminar de identificar los MCSes de un defecto, el método busca en el *modelo transformado* a qué dependencia del modelo de características pertenece cada restricción de los MCS. Luego, el método busca en el *modelo transformado* para cada dependencia implicada en un MCS su representación en lenguaje natural. Finalmente, el método reemplaza en cada MCS las restricciones por lenguaje natural.

La Tabla I muestra los MCSes identificados en este tercer paso del método para la característica muerta *ASP* del modelo de características de la Fig. 1. Gracias a la descripción en lenguaje natural de cada MCS de la tabla, es posible identificar qué dependencias deben ser eliminadas del modelo para corregir ésta característica muerta, tan solo con leer cada MCS. Cada MCS es una corrección que soluciona la característica

Tabla I: Subconjuntos Mínimos de Corrección para la Característica Muerta *ASP* del Modelo de Características de Sitios Web

Id	Corrección	IdDepen	#Elem
MCS1	Dependencia transversal entre :HTTPS y Milisegundos	D22	1
MCS2	Dependencia transversal entre :Estatico y Flash	D24	1
MCS3	Dependencia transversal entre :Reportes y Milisegundos	D21	1
MCS4	Dependencia transversal entre :Dinamico y HTTPS	D27	1
MCS5	Dependencia obligatoria entre Servicios Adicionales y Reportes	D7	1
MCS6	Dependencia obligatoria entre Servicios Adicionales y Banners	D8	1
MCS7	Dependencia opcional entre Banners y Flash	D10	1
MCS8	Dependencia obligatoria entre Contenido y Estático	D15	1
MCS9	Dependencia grupal entre Dinamico y [ASP, PHP, JSP]	D18	1
MCS10	Dependencia obligatoria entre Servidor web y Contenido, Dependencia opcional entre Contenido y Dinámico	D11,D14	2
MCS11	Dependencia obligatoria entre Sitio web y Servidor web, Dependencia opcional entre Servidor web y Protocolos, Dependencia opcional entre Contenido y Dinámico	D3,D12,D14	3
MCS12	Dependencia obligatoria entre Sitio web y Servidor web, Dependencia grupal entre Protocolos y [FTP, HTTPS], Dependencia opcional entre Contenido y Dinámico	D3,D16,D14	3

muerta *ASP*. Por esta razón, aplicar una de las correcciones identificadas es suficiente para solucionar el defecto. Así por ejemplo, el primer MCS indica que se debe eliminar la dependencia transversal entre las características *HTTPS* y *Milisegundos*. Esta corrección tiene sentido, ya que por la dependencia de exclusión entre las características *HTTPS* y *Milisegundos*, la característica *HTTPS* no puede ser seleccionada en ningún producto derivado de este modelo de características. Por lo tanto, no se cumple que si la característica *Dinámico*, padre de la característica *ASP*, está en un producto derivado de la línea de productos, también está la característica *HTTPS*. En consecuencia, la característica *Dinámico* y sus características hijas (entre ellas *ASP*) no pueden ser seleccionadas en ningún producto derivado de la línea de productos (son características muertas). Al eliminar la dependencia sugerida por el MCS1 esta contradicción entre las dependencias se solucionaría y la característica *ASP* dejaría de ser muerta.

4) Identificación de correcciones- análisis: Para la característica muerta *ASP*, el método en el tercer paso identifica 12 correcciones. Esta variedad de posibilidades genera preguntas como: ¿cuál de todas las correcciones es mejor? ¿es preferible la corrección que implique la menor cantidad de cambios en el modelo?, ¿es mejor tener diferentes alternativas de corrección?.

En nuestra opinión, son los diseñadores de los modelos de características quienes pueden responder estas preguntas, pues son ellos quienes conocen el dominio que representa el modelo de características. Por esta razón, nuestra aproximación, a diferencia de lo propuesto en la literatura [7], [10], [11], presenta todos los MCSes que pueden ser identificados al eliminar sistemáticamente dependencias del modelo de características. Con esta información, de acuerdo a sus intereses los diseñadores pueden decidir qué corrección aplicar. Esta decisión puede relacionarse, por ejemplo, con el MCS que implique la menor cantidad de cambios, o con el MCS que implique mantener en el modelo de características alguna dependencia en particular.

Las 12 correcciones identificadas por el método para la característica muerta *ASP* son mínimas pues son MCSes (ver Subsección II-E). Identificar únicamente correcciones mínimas es importante, pues una corrección que implique eliminar todas las dependencias del modelo, no es útil para los diseñadores de los modelos de características [19]. Por el contrario, lo verdaderamente útil es identificar, como en el método que proponemos, correcciones mínimas. Es decir, correcciones formadas únicamente por dependencias que en realidad tienen relación con los defectos del modelo.

IV. Detalles de Implementación

Todos los pasos del método propuesto son sistemáticos y por esa razón pudieron ser implementados de forma automática en una herramienta computacional. Para ello, fue desarrollada en Java una herramienta en la que se implementaron los tres pasos del método propuesto. La funcionalidad de esta herramienta será luego integrada en *VariaMos* [30], nuestra principal herramienta de análisis de modelos de variabilidad.

La herramienta que implementa el método propuesto está integrada con librerías Java para ejecutar programas de restricciones en GNU-Prolog [23] o SWI-Prolog [24]. La herramienta no tiene presentación gráfica, sino que debe ser invocada programáticamente. Los modelos de características que analiza deben estar expresados en el formato *sxfm* (Simple XML Feature Model)[3]. Los resultados del análisis se exportan en un archivo en formato *"xls"*. Este archivo contiene en la primera hoja los defectos identificados para el modelo de características y en la segunda hoja las correcciones identificadas para cada defecto. La herramienta y su manual de instalación están disponibles en Internet [4].

V. Evaluación Preliminar

El método propuesto en este artículo se evaluó de manera preliminar en 78 modelos de características diferentes, con tamaños de hasta 120 dependencias. Los modelos fueron analizados con la herramienta que implementa el método propuesto. La evaluación preliminar se focalizó en dos aspectos: la exactitud y el rendimiento. En las siguientes subsecciones se presentan los detalles de los experimentos desarrollados y se discuten los resultados obtenidos.

A. Exactitud

La exactitud se refiere a la proximidad de los resultados obtenidos por el método en comparación con los valores esperados [31]. Para evaluar la exactitud fueron definidos dos criterios: el porcentaje de falsos positivos y el porcentaje de aciertos. A continuación, se explican ambos criterios.

- **Falsos positivos:** los falsos positivos corresponden a subconjuntos que fueron identificados por el método,

[3]El formato sxfm, es una representación bien conocida por la comunidad de líneas de productos de software, para representar modelos de características. Información sobre su sintaxis está disponible en http://www.splot-research.org/

[4]https://github.com/lufe089/CLEI2014/tree/master/Herramienta

pero que no son una corrección. Particularmente, una corrección es un falso positivo, si al eliminar sus elementos del modelo de características, el defecto para el cual fue identificada la corrección, no desaparece. Además, una corrección es un falso positivo si no es mínima. Es decir, si no es necesario eliminar del modelo todos los elementos de la corrección para corregir el defecto.

- **Aciertos:** los aciertos son valores correctamente identificados por el método propuesto. Particularmente, una corrección es un acierto, si es un subconjunto de dependencias mínimo que corrige al menos un defecto del modelo de características al ser eliminado del modelo.

Durante la evaluación se hizo una inspección manual de los resultados para detectar si habían falsos positivos. Esta inspección se llevó a cabo para cada corrección de la siguiente manera:

1) Se creó manualmente un programa de restricciones irresoluble que representara el modelo de características con el defecto para el que fue identificada la corrección. Luego, a ese programa de restricciones se le eliminó manualmente las restricciones que hacían parte de la corrección.
2) Se ejecutó el programa de restricciones resultante. Si el programa de restricciones era resoluble, entonces la corrección solucionaba el defecto.
3) Se evaluó si la corrección era mínima. Según la definición de los MCSes (ver Subsección II-E), si a un subconjunto irresoluble α, se le elimina únicamente una parte de un MCS, el subconjunto α debe seguir siendo irresoluble. En efecto, si el MCS es mínimo, todos sus elementos deben ser eliminados de α para volver α resoluble. Para esta parte de la inspección manual se siguió la misma idea. La inspección consistió en crear nuevamente un programa de restricciones irresoluble que representara el modelo de características con el defecto para el que fue identificada la corrección. Luego, únicamente una parte de la corrección era eliminada de ese programa de restricciones. Si como se esperaba, el programa de restricciones seguía siendo irresoluble, entonces la corrección era mínima. Por el contrario, si el programa de restricciones se volvía resoluble, entonces parte de la corrección era innecesaria para solucionar el defecto y la corrección no era mínima.

Cuando la corrección solucionaba el defecto y era mínima, entonces era un acierto. De lo contrario era un falso positivo.

Con el fin de evaluar la exactitud del método con los criterios anteriormente presentados, se definieron tres niveles de evaluación: *controlado, semi controlado y aleatorio*. Todos los modelos de características usados para la evaluación se encuentran disponibles en Internet[5].

1) **Controlado:** en este nivel se evaluó la exactitud del método en dos modelos de características de la literatura para los que ya se conocían los resultados: el modelo de características que representa un sistema de integración de hogar propuesto por Trinidad *et al.* [11] y el modelo de características que representa una versión simplificada de las características del sistema operativo Ubuntu, propuesto por Felfering *et al.* [32]. La exactitud en este nivel se evaluó comparando las correcciones obtenidas por el método propuesto, con los resultados presentados en los trabajos de investigación de donde fueron tomados los modelos. Los resultados obtenidos en este nivel indican que el método propuesto es exacto, ya que los resultados coinciden con los resultados de los trabajos de la literatura y al inspeccionarlos manualmente se constantó que eran aciertos.
2) **Semi-Controlado:** en este nivel se evaluó la exactitud del método con un caso de estudio, del cual se conocían los defectos y algunas de las correcciones. El caso de estudio corresponde al modelo de características de sitios web presentado en la sección II-C. Debido a que los defectos fueron introducidos adrede en el modelo, se conocían algunas de las correcciones que se esperaban identificar. Sin embargo, dado que el método identificó más correcciones de las previstas, los resultados se inspeccionaron manualmente. Para los defectos introducidos en el modelo fueron identificadas 187 correcciones, de las cuales 20 correcciones eran comunes a más de un defecto. Este resultado es interesante pues indica que eliminando cierto subconjunto de dependencias podría solucionarse más de un defecto del modelo de características. Por ejemplo, al eliminar la dependencia transversal entre Estático-Flash se corrigen del modelo las características muertas *ASP, Base de datos, Dinámico, HTTPS, JSP, Minutos, PHP, Segundos*; las falsas características opcionales *Archivo, Flash, FTP, Milisegundos, Servicios adicionales y Registro*; y las redundancias *Dependencia transversal: Archivo - FTP y Dependencia transversal Contenido - Protocolos*.
3) **Aleatorio:** en este nivel se evaluó la exactitud del método en 25 modelos, para los que no se conocían con anterioridad las correcciones de los defectos. De una parte, tres de los 25 modelos fueron propuestos en literatura relacionada con líneas de productos y están disponibles en el repositorio de modelos de características SPLOT [33]. De otra parte, los restantes 22 modelos, fueron creados con el generador automático de modelos de características BeTTy [16] Todos los modelos tenían menos de 35 dependencias. Esto con el fin de facilitar la inspección manual de los resultados. Además, cada modelo tenía al menos uno de los defectos presentados en la Subsección II-B y había al menos un modelo para cada tipo de defecto. Es importante resaltar que en este nivel se evalúo el método en modelos de características que tenían originalmente defectos y no en modelos de características modificados. De esta manera, se evitó introducir patrones en las correcciones de los defectos. Además, los modelos evaluados tenían entre sí, diferente cantidad de defectos, de características y de dependencias. Estas variaciones se hicieron con la intención de validar que la exactitud del

[5]https://github.com/lufe089/CLEI2014/tree/master/Modelos\%20probados

método, no estuviera influenciada, ni por patrones predeterminados de correcciones, ni por la cantidad de dependencias de los modelos de características, ni por el tipo de defecto, ni por la cantidad de defectos. Los resultados obtenidos en este nivel indican que el método propuesto es exacto pues tuvo 100 % de aciertos y 0 % de falsos positivos. Por razones de espacio no se detallan aquí los modelos analizados, pero los resultados detallados están disponibles en Internet[6].

B. Rendimiento

Una evaluación empírica del rendimiento del método propuesto fue llevada a cabo. La evaluación consistió en medir el tiempo que tomó la herramienta que implementa el método en identificar las correcciones de 50 modelos de características generados automáticamente con BeTTy [16]. A partir de esta información, fue comparado el tiempo de ejecución del método contra cuatro variables: el número de dependencias, el número de defectos, el número de correcciones, y el número máximo de elementos de las correcciones identificadas (ver Fig. 2). Estas variables se seleccionaron debido a que se consideró que podían tener alguna incidencia en el tiempo de ejecución del método. Las pruebas fueron ejecutadas en un computador portátil con Windows 7 Ultimate de 32 bits, procesador Intel Core i5-2410M, memoria RAM de 4.00 GB, de los cuales 2.66 GB son usadas por el sistema operativo. Cada modelo tenía al menos uno de los defectos presentados en la Subsección II-B de este artículo y había al menos un modelo para cada tipo de defecto. El modelo más pequeño tenía 10 dependencias y el más grande 120 dependencias.

Para facilitar el análisis de los resultados, las subfiguras de la Fig. 2 muestran en el eje *y* el tiempo de ejecución en escala logarítmica en base 10, pues había una gran diferencia en las magnitudes de los tiempos obtenidos para los 50 modelos considerados.

Como se puede observar en la Fig.2a, al parecer el tiempo de ejecución no tiene relación con el número de dependencias del modelo de características, pues los datos no presentan ninguna tendencia. Así por ejemplo, el método tomó menos tiempo en identificar los defectos y las correcciones en un modelo de características con 120 dependencias que en un modelo de características con 80 dependencias (ver Fig. 2a). Esto puede explicarse porque si un modelo de características con 120 dependencias tiene menos defectos que un modelo de características con 80 dependencias, es posible que el método analice más rápido el modelo con menos defectos.

Por su parte, al parecer el tiempo de ejecución tiene relación con el número de defectos, por lo que en diferentes modelos, a medida que aumentó el número de defectos también aumentó el tiempo de ejecución que tardó la herramienta desarrollada en analizar los modelos (ver Fig. 2b). No obstante, hubo modelos de características con un solo defecto para los el método tardó más tiempo que cuando analizó modelos que tenían mayor número de defectos. En cuanto al tiempo y el número de correcciones, al parecer, el tiempo de ejecución tiene alguna relación con la cantidad de correcciones identificadas. Como se puede observar en la Fig. 2c, en varios modelos el tiempo de ejecución aumentó a medida que aumentó el número de correcciones. Lo anterior quiere decir, que es de esperar que el método tome más tiempo en analizar modelos de características con 200 correcciones que modelos de características con 20 correcciones. Finalmente, al parecer, el tiempo de ejecución tiene relación con el número máximo de elementos de las correcciones (ver Fig. 2d). En efecto, el tiempo de ejecución aumentó a medida que aumentó el número máximo de elementos de las correcciones, por lo que es de esperar que el método tome más tiempo en analizar un modelo cuyas correcciones tengan cuatro elementos, que en analizar un modelo cuyas correcciones tengan un solo elemento. Esta relación tiene sentido, ya que el método no terminará hasta que identifique todas las correcciones mínimas del modelo de características, sin importar la cantidad de elementos que éstas tengan.

La evaluación preliminar del rendimiento indica entonces que es díficil predecir cuánto tiempo tardará el método en identificar los defectos y las correcciones de un modelo de características. En efecto, de acuerdo a lo encontrado en los diagramas de dispersión, al parecer el tiempo de ejecución del método tiene relación con el número de defectos, el número de correcciones y el tamaño de las correcciones, pero esta información no se conoce sino después de analizar el modelo de características. Antes de analizar un modelo de características con el método aquí propuesto, el diseñador de los modelos sólo conoce el número de dependencias que tiene el modelo. Sin embargo, según el diagrama de dispersión de la Fig. 2a, el tiempo de ejecución parece tener baja relación con el número de depencias del modelo y por lo tanto, modelos de características con diferente número de dependencias, pueden tener el mismo tiempo de ejecución, y modelos con el mismo número de dependencias pueden tener diferentes tiempos de ejecución.

VI. Trabajos Relacionados

En la literatura existen diferentes trabajos que proponen identificar automáticamente defectos semánticos en modelos de características [4]–[8], [34]. Sin embargo, ninguno de ellos explica cómo corregir los defectos identificados. En ese sentido, nuestra propuesta es una solución más completa que integra no sólo la identificación de los defectos, sino también la identificación de posibles correcciones para cada defecto identificado.

En cuanto al uso de MCSes, Reiter [35] fue pionero en usar los MCSes para identificar correcciones a programas de restricciones irresolubles. En adelante, otros trabajos han propuesto algoritmos para identificar MCSes en programas de restricciones booleanos [25], [26], [36], [37], o enteros [38], [39]. Sin embargo, estos trabajos están enfocados en proponer algoritmos para identificar MCSes, mientras que nuestra propuesta se enfoca en aplicar el concepto de los MCSes en modelos de características defectuosos, los cuales pueden ser representados como programas de restricciones. Uno de nuestros trabajos futuros consiste en probar y comparar los algoritmos propuestos en la literatura para optimizar la identificación de los MCSes.

En cuanto a la identificación de defectos y sus correcciones en modelos de características, Trinidad *et al.* [11]

[6]https://github.com/lufe089/CLEI2014/tree/master/Resultados

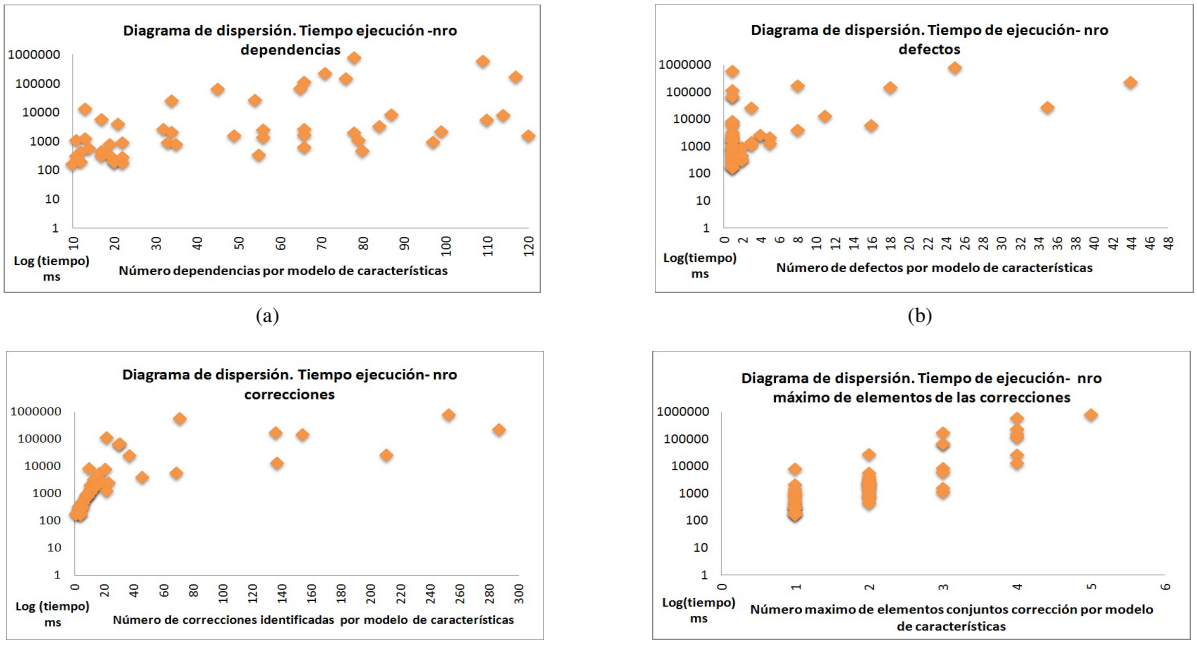

Figura 2: Diagramas de dispersión tiempo vs [nro dependencias, nro defectos, nro correcciones y nro máximo elementos de las correcciones]

transforman el modelo de características en un problema de diagnóstico y luego resuelven un programa de satisfacción de restricciones para identificar las correcciones de menor tamaño para cada defecto identificado. Los defectos soportados son: modelos vacíos, características muertas y falsas características opcionales en modelos de características. La propuesta fue automatizada en FaMa [40], un framework desarrollado en Java para el análisis automático de modelos de características. Sin embargo, este trabajo no le presenta al diseñador del modelo de características, otros subconjuntos de dependencias (no necesariamente del menor tamaño) que al eliminarse del modelo también podrían corregir el defecto analizado. Así por ejemplo, de la característica muerta *ASP*, la propuesta de Trinidad *et al.* [11] identifica las correcciones que tienen un elemento, pero no identifica las correcciones de dos y tres elementos. En un trabajo posterior, Trinidad y Ruiz-Cortés [12], proponen utilizar lógica abductiva para explicar por qué se presentan características muertas, falsas opciones y modelos vacíos. Sin embargo, los autores no presentan ni algoritmos, ni otros detalles que permita usar o implementar su propuesta.

Wang *et al.* [13] propusieron identificar correcciones para modelos de características vacíos. En este trabajo, el diseñador del modelo de características asigna prioridades a las dependencias del modelo. Luego, por medio de una herramienta, el trabajo identifica un subconjunto mínimo de dependencias, de menor prioridad, que deben eliminarse del modelo de características para que el modelo deje de ser vacío y permita derivar al menos un producto. Sin embargo, este método sólo identifica una corrección al tiempo y debe aplicarse nuevamente si la solución propuesta no se ajusta a los intereses del usuario. Además, este trabajo no identifica, como si lo hace el método aquí propuesto, correcciones para las características muertas, las falsas características opcionales o las redundancias.

Noorian *et al.* [9] proponen un framework basado en lógica descriptiva [41] para identificar si un modelo de características es vacío y para proponer posibles correcciones. El framework transforma automáticamente un modelo de características en formato *sxfm* a OWL-DL y usa el razonador ontológico Pellet [42] para verificar si el archivo OWL-DL que representa el modelo de características es inconsistente. Si es así, el framework invoca la funcionalidad de Pellet para extraer subconjuntos mínimos de axiomas OWL que deben quitarse para volver consistente el archivo OWL. Esta propuesta identifica correcciones para modelos vacíos y configuraciones inválidas, pero no considera otros defectos como las características muertas, las falsas características opcionales, las redundancias, o los falsos modelos de LP. Además, las correcciones se presentan en OWL-DL puro, por lo que son difíciles de entender para el diseñador del modelo de características.

Thüm *et al.* [10] proponen Feature IDE, una herramienta que soporta el desarrollo de software orientado a características. Cuando un modelo de características es vacío, tiene falsas características opcionales o tiene características muertas, FeatureIDE automáticamente identifica las dependencias de inclusión y exclusión, que al eliminarse corrigen cada defecto. Sin embargo, la identificación no es completa. Si corregir un defecto requiere eliminar más de una dependencia, FeatureIDE no identificará ninguna corrección para ese defecto. Además, FeatureIDE no identifica ni las redundancias ni los falsos modelos de LP, ni sus correcciones, como sí lo hace el método propuesto en este artículo.

Rincón *et al.* [43] usan ontologías y un conjunto de

reglas en un lenguaje de consultas para ontologías, con el fin de: (i) representar modelos de características, (ii) identificar características muertas y falsas características opcionales, (iii) identificar algunas de las causas que originan estos defectos, y (iv) formular una explicación en lenguaje natural. Sin embargo esta propuesta únicamente identifica algunos defectos y algunas de sus causas. Si el modelo tiene defectos que no se acogen a las reglas definidas, entonces estos no serán detectados. Además, este trabajo identifica algunas causas, mientras que el método propuesto en este artículo, identifica correcciones. Por lo tanto, ambas propuestas son complementarias y al integrarlas se podrían identificar los defectos de un modelo, sus causas y sus correcciones.

En un trabajo anterior Rincón *et al.* [44] propusieron un método semiautomático para identificar las correcciones de las características muertas de los modelos de características. En ese trabajo, el método recibe como entrada un modelo de características expresado como un programa de restricciones. Luego, a ese programa de restricciones se le identifican las características muertas, y para cada característica muerta se identifican, como en este artículo, los MCSes que corresponden a las correcciones. Ese trabajo es semiautomático, puesto que se debe realizar manualmente, la transformación del modelo de características en un programa de restricciones en Prolog y por cada característica muerta se debe adicionar manualmente la restricción de verificación que permite identificar los MCSes. El método presentado en este artículo, es una continuación de ese trabajo de investigación. Por esta razón, en esta propuesta se identifican también los MCSes, pero con un método completamente automático. Además, el método presentado en este artículo soporta también otros tipos de defectos de los modelos de características y entrega las correcciones en lenguaje natural para facilitar la comprensión de los resultados.

VII. Conclusiones y Trabajos Futuros

Garantizar la calidad de los modelos de características es de gran importancia para aprovechar los beneficios propuestos por las LP. En este artículo se presenta un método que no sólo identifica los defectos semánticos de los modelos de características, sino que también identifica correcciones para cada defecto. La corrección consiste en presentar a quien analiza el modelo de características, todos los subconjuntos mínimos de dependencias del modelo de características, que podrían ser eliminados para corregir cada defecto identificado.

A diferencia de los trabajos encontrados en la literatura y de nuestros trabajos previos, la propuesta presentada en este artículo, permite identificar automáticamente los defectos de los modelos de características y un amplio número de correcciones posibles para cada defecto. Además, estas correcciones están expresadas en un lenguaje fácilmente entendible para ayudar a comprender los resultados. Como se indica en [3], [11], [45]–[47], toda información que permita guiar el proceso de corrección, ofrece un ahorro de tiempo y costo en el desarrollo de la LP. En efecto, conocer rápidamente las correcciones de los defectos, permite que los diseñadores de los modelos de características puedan enfocarse en hacer una buena representación del dominio de la LP, más que en encontrar cómo corregir los defectos de los modelos.

Nuestra propuesta es una primera aproximación para identificar defectos y sus correcciones en modelos de características. Sin embargo, en el método únicamente identificamos las correcciones que implican eliminar dependencias en el modelo. Otras correcciones que involucren por ejemplo modificar el modelo, ampliar el dominio de las variables, o adicionar nuevas dependencias, aún no son identificadas. Adicionalmente, consideramos que es importante definir criterios que le faciliten al diseñador del modelo, la selección de la mejor corrección entre las correcciones propuestas. Algunos criterios podrían ser, por ejemplo, si la corrección genera nuevos defectos, si restringe la cantidad de productos derivables del modelo de características, o si aplicar la corrección soluciona más de un defecto del modelo. Además, este trabajo aún no identifica la(s) causa(s) que explican por qué los defectos ocurren. Ampliar el método para identificar también las causas de los defectos hace parte de nuestra línea futura de investigación.

Agradecimientos

El trabajo de investigación presentado en este artículo, hace parte del proyecto RC Nro 0634-2013 titulado *Desarrollo de soluciones para soportar la completitud y la corrección de lineas de productos con aplicación a la ingeniería de software*. Este proyecto es financiado por el Departamento Adminstrativo de Ciencia Tecnología e Innovación COLCIENCIAS de la República de Colombia.

Referencias

[1] K. Pohl, G. Böckle, and F. J. van Der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., 2005.

[2] P. Clements and L. M. Northrop, *Software Product Lines: Practices and Patterns*, 1st ed. Addison-Wesley Professional, 2001.

[3] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and S. P. Peterson, "Feasibility Study Feature-Oriented Domain Analysis (FODA). Technical Report," Software Engineering Institute,Carnegie Mellon University, Tech. Rep., 1990.

[4] C. Salinesi and R. Mazo, "Defects in Product Line Models and how to identify them," in *Software Product Line - Advanced Topic*, A. Elfaki, Ed. InTech, 2012, ch. 5, pp. 97–122.

[5] D. Batory, "Feature models, grammars, and propositional formulas," in *Proceedings of the 9th international conference on Software Product Lines SPLC'05*, ser. SPLC'05. Rennes, France: Springer-Verlag, 2005, pp. 7–20.

[6] A. Osman, S. Phon-Amnuaisuk, and C. Kuan Ho, "Knowledge Based Method to Validate Feature Models," in *First International Workshop on Analyses of Software Product Lines*, 2008, pp. 217–225.

[7] H. Wang, Y. Li, J. Sun, H. Zhang, and J. Pan, "Verifying feature models using OWL," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 5, no. 2, pp. 117–129, Jun. 2007.

[8] T. Von der Massen and H. Lichter, "Deficiencies in Feature Models," in *Workshop on Software Variability Management for Product Derivation - Towards Tool Support*, T. Mannisto and J. Bosch, Eds., 2004.

[9] M. Noorian, A. Ensan, E. Bagheri, H. Boley, and Y. Biletskiy, "Feature Model Debugging based on Description Logic Reasoning," in *DMS'11*, 2011, pp. 158–164.

[10] T. Thüm, C. Kastner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, "FeatureIDE: An extensible framework for feature-oriented software development," *Science of Computer Programming*, 2012.

[11] P. Trinidad, D. Benavides, A. Duran, A. Ruiz-Cortés, and M. Toro, "Automated Error Analysis for the Agilization of Feature Modeling," *Journal of Systems and Software*, vol. 81, no. 6, pp. 883–896, 2008.

[12] P. Trinidad and A. Ruiz-Cortés, "Abductive Reasoning and Automated Analysis of Feature models: How are they connected," in *Proceedings of the Third International Workshop on Variability Modelling of Software-Intensive Systems*, 2009, pp. 145–153.

[13] B. Wang, Y. Xiong, Z. Hu, H. Zhao, W. Zhang, and H. Mei, "A dynamic-priority based approach to fixing inconsistent feature models," in *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I*, ser. MODELS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 181–195.

[14] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah, "Reveal: A Formal Verification Tool for Verilog," in *Logic for Programming Artificial Intelligence and Reasoning (LPAR-2008)*, A. Cervesato, Iliano and Veith, Helmut and Voronkov, Ed., 2008, vol. 5330, pp. 343–352.

[15] S. Safarpour, H. Mangassarian, A. Veneris, M. H. Liffiton, and K. A. Sakallah, "Improved Design Debugging Using Maximum Satisfiability," in *Proceedings of the Formal Methods in Computer Aided Design*, ser. FMCAD '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 13–19.

[16] S. Segura, J. A. Galindo, D. Benavides, J. A. Parejo, and A. Ruiz-Cortés, "BeTTy: benchmarking and testing on the automated analysis of feature models," in *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, ser. VaMoS '12. New York, NY, USA: ACM, 2012, pp. 63–71.

[17] M. Mendonça, T. T. Bartolomei, and D. Cowan, "Decision-making coordination in collaborative product configuration," in *Proceedings of the 2008 ACM symposium on Applied computing*, ser. SAC '08. New York, NY, USA: ACM, 2008, pp. 108–113.

[18] K. Czarnecki, S. Helsen, and U. Eisenecker, "Formalizing Cardinality-based Feature Models and their Specialization," *Software Process: Improvement and Practice*, vol. 10, no. 1, pp. 7–29, 2005.

[19] D. Batory, D. Benavides, and A. Ruiz-Cortés, "Automated analysis of feature models: challenges ahead," *Communications of the ACM*, vol. 49, no. 12, pp. 45–47, 2006.

[20] F. Rossi, P. Van Beek, and T. Walsh, *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc, 2006, vol. 35.

[21] M. Gavanelli and F. Rossi, "Constraint Logic Programming," in *A 25-year perspective on logic programming*. Springer-Verlag Berlin, Jan. 2010, vol. 6125, no. 4, pp. 64–86.

[22] R. Mazo, "A Generic Approach for Automated Verification of Product Line Models," Ph.D.thesis, Ph.D.dissertation.Paris 1 Panthéon Sorbonne University, Paris, France, 2011.

[23] D. Diaz and P. Codognet, "The GNU prolog system and its implementation," *Design and Implementation of the GNU Prolog System*, 2001.

[24] J. Wielemaker, "SWI Prolog Reference Manual (Version 6.2.2)," 2012. [Online]. Available: http://www.swi-prolog.org

[25] J. Bailey and P. J. Stuckey, "Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization," in *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages (PADL'05). Lecture Notes in Computer Science*. Long Beach,USA: Springer Berlin Heidelberg, 2005, pp. 174–186.

[26] M. H. Liffiton and K. Sakallah, "Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints," *Journal of Automated Reasoning*, vol. 40, no. 1, pp. 1–33, 2008.

[27] A. Morgado, M. H. Liffiton, and J. Marques-silva, "MaxSAT-Based MCS Enumeration," in *Haifa Verification Conference (HVC 2012)*. Springer, 2012.

[28] B. O'Sullivan, A. Papadopoulos, B. Faltings, and P. Pu, "Representative explanations for over-constrained problems," in *Proceedings of the 22nd national conference on Artificial intelligence - Volume 1*, ser. AAAI'07. AAAI Press, 2007, pp. 323–328.

[29] R. Mazo, C. Salinesi, D. Diaz, O. Djebbi, and A. Michiels, "Constraints: the Heart of Domain and Application Engineering in the Product Lines Engineering Strategy," *International Journal of Information System Modeling and Design IJISMD*, vol. 3, no. 2, pp. 33–68, 2011.

[30] R. Mazo, C. Salinesi, and D. Diaz, "Variamos: a tool for product line driven systems engineering with a constraint based approach," in *CAiSE Forum*, 2012, pp. 147–154.

[31] ISO-5725, "International Standard ISO 5725-1, Accuracy (trueness and precision) of measurement methods and results General Principles and definitions," 1994.

[32] A. Felfernig, D. Benavides, J. Galindo, and F. Reinfrank, "Towards Anomaly Explanation in Feature Models," in *Proceedings of the Workshop on Configuration*, no. 827587, Vienna, Austria, 2013, pp. 117–124.

[33] M. Mendonça, M. Branco, and D. Cowan, "S.P.L.O.T.: software product lines online tools," in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, ser. OOPSLA '09. New York, NY, USA: ACM, 2009, pp. 761–762. [Online]. Available: http://doi.acm.org/10.1145/1639950.1640002

[34] W. Zhang and H. Zhao, "A Propositional Logic-Based Method for Verification of Feature Models," in *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM'04)*, J. Davies, W. Schulte, and M. Barnett, Eds., Seattle-USA, 2004, pp. 115–130.

[35] R. Reiter, "A theory of diagnosis from first principles," *Artificial Intelligence*, vol. 32, no. 1, pp. 57–95, Apr. 1987.

[36] M. H. Liffiton and A. Malik, "Enumerating Infeasibility: Finding Multiple MUSes Quickly," in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, ser. Lecture Notes in Computer Science, C. Gomes and M. Sellmann, Eds. Springer Berlin Heidelberg, 2013, vol. 7874, pp. 160–175.

[37] J. Marques-Silva, F. Heras, M. Janota, A. Previti, and A. Belov, "On Computing Minimal Correction Subsets," in *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*, Beijing, China, 2013.

[38] A. Felfernig, M. Schubert, and C. Zehentner, "An efficient diagnosis algorithm for inconsistent constraint sets," *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, vol. 26, no. 1, pp. 53–62, 2012.

[39] J. Marques-Silva and I. Lynce, "On improving MUS extraction algorithms," in *Proceedings of the 14th international conference on Theory and application of satisfiability testing*, ser. SAT'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 159–173.

[40] P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, and A. Jimenez, "FAMA Framework," in *Proceedings of the 12th International Software Product Line Conference*, ser. SPLC '08. Washington, DC, USA: IEEE Computer Society, 2008, p. 359.

[41] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, Eds., *The description logic handbook: theory, implementation, and applications*. New York, NY, USA: Cambridge University Press, 2003.

[42] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical OWL-DL reasoner," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 5, no. 2, pp. 51–53, 2007.

[43] L. Rincón, G. L. Giraldo, R. Mazo, and C. Salinesi, "An ontological rule-based approach for analyzing dead and false optional features in feature models," in *XXXIX Latin American Computing Conference (CLEI)*, 2013.

[44] L. F. Rincón Perez, G. L. Giraldo Gómez, R. Mazo, C. Salinesi, and D. Diaz, "Subconjuntos Minimos de Correccion para explicar caracteristicas muertas en Modelos de Lineas de Productos. El caso de los Modelos de Caracteristicas," in *Proceedings of the 8th Colombian Computer Conference (CCC)*, Armenia-Colombia, 2013.

[45] K. Lauenroth, A. Metzger, and K. Pohl, "Quality Assurance in the Presence of Variability," in *Intentional Perspectives on Information Systems Engineering*, S. Nurcan, C. Salinesi, C. Souveyet, and J. Ralyté, Eds. Springer Berlin Heidelberg, 2010, pp. 319–333.

[46] A. Osman, S. Phon-Amnuaisuk, and C. Kuan Ho, "Investigating Inconsistency Detection as a Validation Operation in Software Product Line," in *Software Engineering Research, Management and Applications*, N. Lee, Roger and Ishii, Ed. Springer Berlin Heidelberg, 2009, vol. 253, pp. 159–168.

[47] J. Sun, H. Zhang, and H. Wang, "Formal Semantics and Verification for Feature Modeling," in *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, ser. ICECCS '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 303–312.