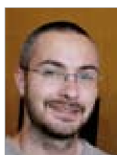
Minería de Repositorios de Software **para ayudar a los desarrolladores**



Romain Robbes
Profesor Asistente, DCC, Universidad de Chile. Doctor en Ciencias de la Computación, Universidad de Lugano, Suiza (2008); Diplome d'etudes approfondies, University of Caen, Francia (2003); Maitrise d'informatique, University of Caen, Francia (2002); DEUG and Licence, informatique (Bachelor), University of Caen, Francia (2001).
rrobbes@dcc.uchile.cl

Uno de los primeros estudios empíricos de desarrollo de software fue el estudio de Lehman y Belady sobre el OS/360 de IBM en los años '70: Ese trabajo fue el origen de las leyes de Lehman sobre la evolución del software [Lehman1985] en las que se establece lo siguiente:

- "Los sistemas de software deben ser continuamente adaptados o se vuelven cada vez menos satisfactorios" (Ley 1).
- "Cuando un sistema de software está cambiando, su complejidad aumenta a menos que algún trabajo se realice para mantener o reducir su complejidad" (Ley 2).
- "La calidad de los sistemas de software parece estar disminuyendo a menos que sean rigurosamente mantenidos y adaptados a los cambios de entorno operativo" (Ley 7).

Desde entonces, las leyes de la evolución del software han sido verificadas una y otra vez.

El mantenimiento de software, es decir, la fase donde los desarrolladores reaccionan a los cambios de requisitos, es la fase más costosa del desarrollo, porque cambiar un sistema grande es extremadamente difícil. ¿Cómo se puede juzgar entonces el impacto de un cambio en un proyecto que ha vivido varios años, ha sido escrito por un equipo de desarrollo de gran tamaño (potencialmente distribuido), y tiene miles -si no millones- de líneas de código?

En este contexto, cada información adicional sobre el sistema que se mantiene es útil.

Para hacer frente a estos desafíos, los investigadores en el área de investigación llamada Mining Software Repositories (MSR) exploran los repositorios de información más precisos que tenemos sobre el

desarrollo de software: los desarrolladores almacenarán discusiones de diseño, informes de problemas, y los cambios que realizan en herramientas especializadas, llamadas repositorios de software. Una vez recuperada – en un proceso que puede ser difícil - la información tiene un enorme valor para los desarrolladores, testeadores, mantenedores, arquitectos y administradores.

En este artículo, en primer lugar, presentaré el tipo de información que se encuentra en dos tipos de repositorios de software: el sistema de control de versiones y el sistema de seguimiento de errores. Luego presentaré cinco enfoques que podrían aprovechar esta información para ayudar al proceso de desarrollo de software de varias maneras distintas.

Por último, voy a presentar algunos de los trabajos recientes en que he estado involucrado, que comparten un objetivo común: ¿Cómo podemos hacer más? La respuesta está en el uso de diferentes repositorios de software, es decir, repositorios de correo electrónico, super-repositorios y repositorios que contienen muchos más detalles sobre las actividades de los desarrolladores.

LOS REPOSITORIOS DE SOFTWARE

El sistema de control de versiones

El Software Configuration Management nace de la necesidad de coordinar el cambio, cuando varios desarrolladores están trabajando en el mismo sistema de software. ¿Cómo podemos asegurarnos de que los cambios de uno no se oponen a los de otra persona? Sin la disciplina adecuada, romper el código de otro desarrollador es extremadamente fácil. Del mismo modo, hay que encontrar una manera eficaz para revisar e integrar los cambios realizados por otra persona en la base de código.

El sistema de control de versiones se encarga de estas cuestiones.

Limitamos la discusión a los sistemas de control de versiones centralizados, como CVS y Subversion. Los sistemas de control de versiones distribuidas como Git o Mercurial funcionan de forma ligeramente diferente y plantean nuevos desafíos para los investigadores de MSR [Bird2009].

Un escenario típico es el siguiente: un programador se conecta al servidor de control de versiones, y recupera la versión más reciente del sistema (check out). Luego trabaja en sus tareas asignadas. En cualquier momento, es libre de hacer un commit, (check in), con sus cambios pendientes. A continuación, se reflejarán los cambios en el servidor; otros desarrolladores pueden integrar estos cambios la próxima vez que hacen un "check out" del sistema. Esta serie de cambios se llama una transacción o un commit. La Figura 1 muestra un "change log" que es una lista de commits. La información típica que se encuentra en un commit - que puede ser explotada posteriormente por enfoques MSR-, es la siguiente:

- El número de la transacción, que da una identidad única para referencia futura.
- El autor del commit (¿quién?).
- La fecha del commit (¿cuándo?).
- Un comentario sobre el commit: texto libre, idealmente sería una razón y una descripción de los cambios (¿por qué?).
- La lista de ficheros cambiados en el commit (¿cuáles?).
- Las líneas añadidas y eliminadas, por cada uno de los ficheros modificados (¿cómo?).

Como veremos a continuación, incluso con estos datos, aunque limitados de información, ya podemos ayudar eficazmente a un profesional.

El sistema de seguimiento de problemas

Más allá de tener varios desarrolladores, los sistemas grandes cuentan con muchas tareas a realizar, y muchos errores que los desarrolladores tienen que corregir.

¿Cómo puede uno asegurarse de que tareas importantes se lleven a cabo a tiempo, y que nadie esté duplicando el trabajo de otro? El sistema de seguimiento de problemas se encarga de esto. Esta herramienta almacena todas las tareas a las que se hace referencia en el sistema, actuando como una especie de lista de tareas. Una tarea puede ser cualquier cosa, desde una solicitud hasta un defecto denunciado por un programador o usuario.

El sistema de seguimiento de problemas permite establecer prioridades, asignar tareas y discutir, con el fin de tomar decisiones informadas sobre cuándo, cómo y por quién, para cada tarea a realizar. La Figura 2 es un ejemplo de problema, o "bug", sobre el "Like Button" de Facebook. Para cada tarea, un sistema de seguimiento de problema tiene información sobre:

Figura 1

```
Fri Oct  8 10:52:25 2010  Nobuyoshi Nakada  <nobu@ruby-lang.org>

        * common.mk (RBCONFIG): depends on version.h due to
          RUBY_PATCHLEVEL.  [ruby-core:32709]

Fri Oct  8 00:24:54 2010  James Edward Gray II  <jeg2@ruby-lang.org>

        * lib/csv.rb: Fixing documentation typos.  [ruby-core:32712]

Thu Oct  7 09:14:28 2010  NARUSE, Yui  <naruse@ruby-lang.org>

        * vm_exec.c (vm_exec_core):  Treat clang as non gcc on this
          context: It has __asm__ but doesn't works well.

Wed Oct  6 12:28:22 2010  Tanaka Akira  <akr@fsij.org>

        * lib/uri/generic.rb (URI::Generic#hostname): new method.
          (URI::Generic#hostname=): ditto.
        * lib/open-uri.rb: use URI#hostname
        * lib/net/http.rb: ditto.
          reported by Adam Majer.  [ruby-core:32056]
```

- Un ID de remisión.
- El desarrollador o usuario que lo presenta.
- El desarrollador al que le fue asignada la tarea.
- Una descripción de la tarea.
- Las prioridades, que van de mayor a menor.
- Las gravedades, que van desde triviales hasta críticas.
- Un espacio para las discusiones sobre el tema.
- Y muchos otros espacios con más detalles.

Cuando un commit corrige un problema en el sistema de seguimiento, la práctica común es mencionarlo en el comentario del commit, haciendo referencia al ID en el texto. Esto permite asociar cada error con los cambios reales que lo corrigieron [Fischer2003].

A continuación describimos varios enfoques que se proponen en la literatura que hacen uso de uno o ambos de estos repositorios.

ENFOQUES DE MINERÍA DE REPOSITORIOS DE SOFTWARE

Predicción de cambios

La predicción de cambios responde a la siguiente pregunta: si cambio esta entidad (por ejemplo, una clase o un método), ¿qué otras entidades tengo que cambiar? Si no se cambian estas entidades se puede provocar la introducción de errores en el sistema. La aproximación clásica al problema se basa en el análisis de impacto: uno tiene que explorar todas las entidades que llaman o son llamadas desde un método para determinar si es necesario cambiarlas. El problema es que, además de ser un conjunto potencialmente elevado de entidades a inspeccionar, esto no cubre todos los casos. Por ejemplo, una función que exporta un documento en un archivo no llama a la función de importación, pero por otro lado estos sí están intrínsecamente ligados y necesitan ser modificados conjuntamente.

Figura 2

Una alternativa al análisis de impacto es buscar reglas implícitas en la historia del desarrollo, como se ha almacenado en el sistema de control de versiones. Cuando los desarrolladores suben sus cambios, podemos formar una asociación entre todas las entidades que fueron cambiadas en el commit. Si estas asociaciones se repiten en el tiempo, es posible haber encontrado una regla implícita de programación, como por ejemplo: cuando el método a() cambia, el método b() tiene que cambiar también.

Supongamos que un programador cambia el método a(), pero no b(). Esto vendría a romper el patrón que encontramos. Un enfoque de predicción de cambios puede emitir advertencias cuando los patrones como el de arriba son violados. De hecho, el programador puede saber realmente si se ha olvidado de cambiar b(), lo que constituiría un error que impediría el enfoque. [Zimmerman2004]

Estos resultados pueden ser evaluados con precisión al repetir el desarrollo del sistema. Para cada commit en el sistema de control de versiones, dividimos el conjunto de ficheros que han cambiado en dos conjuntos A y B. Después podemos dar al conjunto del fichero A a uno de los algoritmos de predicción de cambio, y preguntarle cuál es el conjunto B. Como sabemos el contenido de B, podemos compararlo con las predicciones del algoritmo, y medir su performance [Hassan2006].

Predicción de problemas

Mediante el uso de los datos de los sistemas de seguimiento de problemas -es decir, los errores en el pasado- uno puede construir modelos de predicción de errores futuros. El escenario en este caso es el de la asignación de recursos: si un equipo no tiene suficiente tiempo y/o testeadores para verificar correctamente todo el sistema, debe concentrar sus esfuerzos en las partes del sistema que son las más propensas a tener errores. Una vez más, la historia pasada puede ayudar.

Si tenemos un sistema de seguimiento de defectos, sabemos cuántos errores afectan a cada fichero en cualquier momento. Mediante la formación de un modelo de predicción por parte de los datos, y la evaluación de eso sobre el resto de ella, se puede evaluar qué características son mejores predictores de errores en el futuro. Por ejemplo, mientras más grande sea un archivo o más compleja sea una clase, más errores tenderá a tener. Del mismo modo, los archivos que han cambiado más considerablemente en el pasado son más propensos a presentar errores. Hay muchas métricas y enfoques que se pueden usar para esto. Se puede encontrar una comparación de varios enfoques en uno de mis artículos recién publicados [D'Ambros2010].

Recomendación de expertos y triage de errores

Otra área en la que puede ayudar el MSR es la recomendación de expertos. En situaciones de mantenimiento, es común para los desarrolladores tener que realizar cambios en una parte de un sistema que no conoce bien. Sería de gran ayuda tener la opinión de expertos en el área, pero lo que se necesita primero es identificarlos.

Varios indicadores pueden cuantificar la experiencia de alguien, como la cantidad de cambios que realizó a través del tiempo, el número de errores que deberá ser fijado, o el número de veces que utilizó una entidad determinada en su código. Un sistema de recomendación puede ser construido de

modo que, al explorar el código fuente en un IDE, una lista del personal capacitado en el archivo actual se pueda generar con el fin de contactar fácilmente a los expertos, en caso de que sea necesario. La Figura 3, extraída del artículo de Gîrba [Gîrba2005], muestra cómo los expertos sobre archivos de un sistema cambian con el tiempo y sus actividades. Cada línea representa un archivo y cada punto de color un cambio de un autor. Cada línea tiene el color del autor que sabe más sobre el archivo.

Un problema relacionado es el triage de errores: ¿A quién en el equipo debemos culpar de este nuevo error? Una gran proporción de errores son asignados, efectivamente, a varias personas antes de encontrar a la persona adecuada. Al usar el repositorio defectuoso, podemos entrenar modelos predictivos basados en los errores del pasado. Puesto que sabemos quién en el pasado ha corregido un error, podemos obtener una buena idea del rendimiento real de los modelos de predicción [Anvik2006].

Delta debugging (Delta Depuración o DD) es una forma sistemática para buscar el cambio que causó el accidente basado sobre el sistema de control de versiones. DD realiza una búsqueda dicotómica, mediante la aplicación de la primera mitad de los cambios en el framework, ejecuta la aplicación, y prueba si la aplicación falla. Si es así, refina la búsqueda del error, aplicando el primer cuarto de los cambios. En caso contrario, aplica la segunda mitad de los cambios y hace búsqueda ahí. El proceso continúa sobre el completo historial de los cambios, hasta que DD sea capaz de localizar el commit que es responsable del accidente. Como es evidente, es mucho más fácil entender - y arreglar - un puñado de líneas en lugar de cientos de miles [Zeller1999].

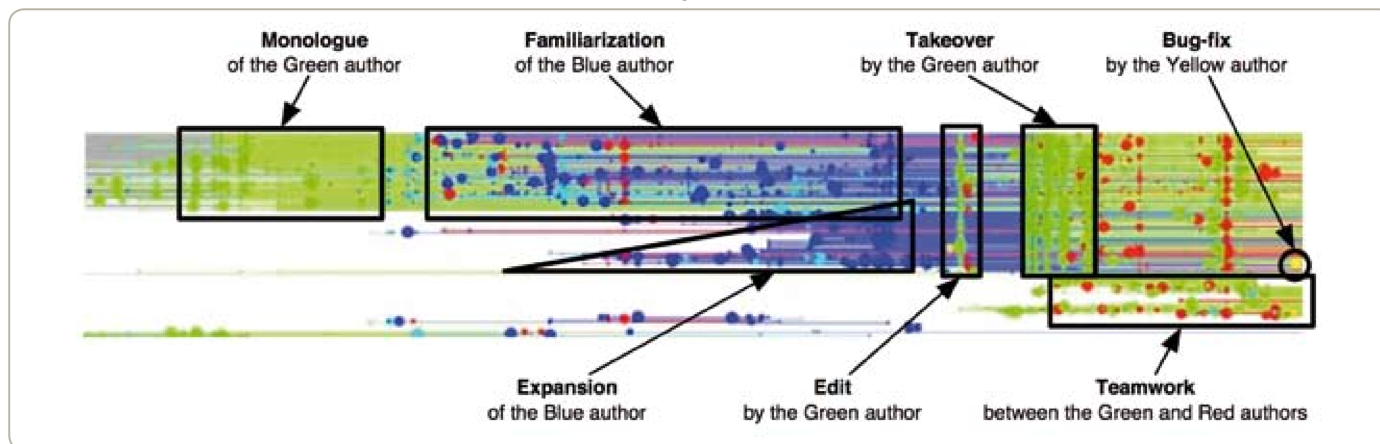
Version-sensitive editing

De la misma manera, una de las primeras herramientas que se dedicaba a explorar la información en un repositorio de versiones se desarrolladores usaron el editor (el editor de versiones firma cada archivo con una firma específica cuando se utiliza), fue posible medir un aumento de la productividad para los usuarios de la herramienta en comparación con los desarrolladores que no hicieron uso de la herramienta. Este incremento, estimado en un 40%, o 1,400 persona/año, se traduce en la cifra mencionada anteriormente [Atkins2002].

MI INVESTIGACIÓN EN MSR

Hasta ahora hemos visto que el MSR puede ser útil en una variedad de casos. Sin embargo, sólo hemos arañado la superficie de todos los enfoques que se han propuesto. Pero hay un principio general: los resultados del MSR son sólo tan buenos como los datos sobre los que se basan. En este contexto, mi objetivo de investigación es hacer frente a este problema, proporcionando datos más precisos, o tipos de datos adicionales.

Figura 3



Delta debugging

Este trabajo fue iniciado por el autor de la herramienta de depuración gráfica DDD que depende del depurador GDB, basado en texto. Entre dos versiones de GDB, DDD dejó de trabajar. Aproximadamente 200.000 líneas de código se cambiaron entre las dos versiones de GDB. La localización de los cambios que son la razón del error es muy difícil en este tipo de situación.

estima que ha salvado 270.000.000 dólares a una gran empresa en el transcurso de varios años. Esta sencilla herramienta, llamada editor de versiones, es un editor de texto que subraya código recientemente retirado, y pone en negritas código recientemente añadido, lo que permite ver más rápido los cambios entre dos versiones.

El editor de la versión se utilizó en una gran empresa durante varios años. Desde que fue posible determinar quiénes de los

Registro de cambios precisos

Los datos almacenados en los repositorios de versiones son notoriamente imprecisos: un sistema de control de versiones trabaja con archivos y no con programas. Si uno quiere hacer un análisis preciso sobre la evolución de los programas, hay que hacer el parsing de cada versión, y después unirlos. Esto representa un montón de trabajo: por

ejemplo, las clases o los métodos pueden cambiar de nombre entre las versiones, lo que es cada vez más común con las herramientas de refactoring en los IDEs. Hay que aceptar la imprecisión o detectarla con un algoritmo específico.

Además, el sistema de control de versiones registró cambios sólo cuando los desarrolladores hicieron un commit; pero ellos quizás hicieron sólo un commit al día, o tal vez uno sólo a la semana. Todos los cambios realizados durante ese tiempo se comprimen en un sólo evento, perdiéndose todas las informaciones sobre el orden de los eventos. Estas dos características de los sistemas de control de versiones reducen en gran medida la precisión de los enfoques MSR que necesitan este nivel de detalle [Robbes2005].

He propuesto una solución al problema de la exactitud de los datos mediante el registro de la actividad de los desarrolladores en el IDE, en lugar de recuperar los cambios de los sistemas de control de versiones. Esto nos permite pensar en las secuencias de cambios en el programa, en lugar de conjuntos desordenados de los cambios en archivos de texto. Esto a su vez nos da una visión mucho más precisa de la evolución del código fuente del sistema. La Figura 4 muestra una visualización de la actividad registrada durante una sesión de desarrollo en la izquierda, y la misma actividad cómo se recupera desde un repositorio de versiones en la derecha. Cada línea representa una método, y cada punto un cambio a un método (verde: creación de método; naranja: cambio; rojo: supresión; azul: otros). El efecto de "compresión" es evidente.

Doy un ejemplo de uso de esta información: un registro tan detallado de la actividad permite evaluar la eficacia de las herramientas en el IDE, de forma similar pero mucho más precisa que lo que la repetición de desarrollo nos permite (ver la sección sobre predicción de cambios). Esto nos permitió evaluar distintas variantes de herramientas comunes de IDE, como el "completador" de código, y proponer un algoritmo de "completado" mucho más preciso que el algoritmo inicial. De hecho, nuestro algoritmo es casi seis veces más exacto que el inicial. También tiene una interfaz gráfica mejor adaptada a su mayor precisión. Ese algoritmo e interfaz es empleado ahora por todos los desarrolladores que utilizan el entorno de programación Pharo. Hicimos una encuesta sobre las herramientas y los usuarios de Pharo prefieren en gran medida nuestra herramienta (Figura 5, izquierda: herramienta anterior; derecha: nuestra herramienta) [Robbes2010a].

Este trabajo sobre el registro de cambio fue la base de mi tesis de Doctorado, titulada "Of Change and Software" [Robbes2008a]. Esto dio lugar a varias publicaciones en los principales lugares de la ingeniería de software, tales como la ASE [Robbes2008b], ICSE [Robbes2008c] y el Journal de ASE [Robbes2010a], sin olvidar la conferencia MSR [Robbes2007a, Robbes2010b], entre otros.

Hacer un link entre e-mails y código fuente

Además, he investigado la utilidad de otras fuentes de datos tales como archivos de correo electrónico, a fin de tener una visión más completa de la evolución de un sistema. Los archivos de correo electrónico comprenden los debates y las peticiones que los desarrolladores hacen durante toda la vida del proyecto.

El primer paso para explotar estos datos es hacer un link entre los e-mails y los elementos del código fuente al que los e-mails hacen referencia. Uno puede pensar en muchas técnicas -sencillas o complicadas- para hacer esto. Nuestra evaluación de varias técnicas para encontrar links entre e-mails y código fuente se publicó en el ICSE de 2010 [Bacchelli2010] y en WCRE el año anterior [Bacchelli2009] donde obtuvo el premio al mejor artículo. Si el problema parece simple, evaluar sistemáticamente la exactitud de los métodos y sus variantes nos ha permitido encontrar resultados inesperados: enfoques simples basado en expresiones regulares terminó con mejores resultados que las técnicas avanzadas de minería de texto, tales como Vector Space Models y Latent Semantic Indexing.

Nuestro enfoque se centró en e-mails, pero puede funcionar con cualquier repositorio que contiene texto, por ejemplo conversaciones sobre el archivo defectuoso, comentarios de commit, archivos de programas de chat, o sobre la documentación del software.

Figura 4

Figura 5

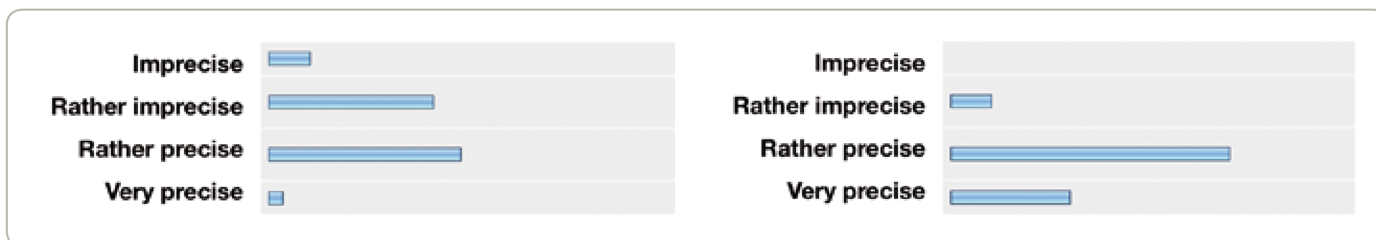
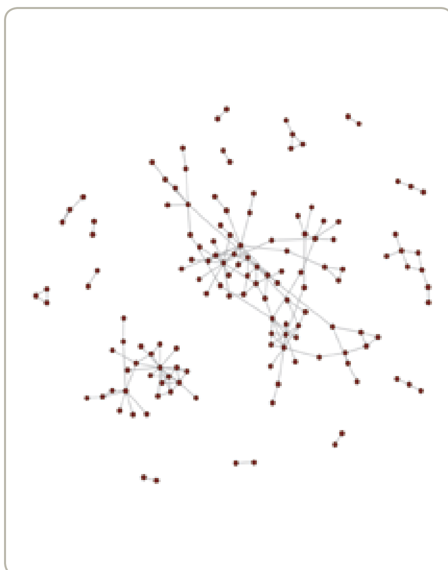


Figura 6



Minería de ecosistemas de software

Por último, la evolución de un sistema puede verse afectado por otros sistemas, como mostró el ejemplo sobre el Delta Debugging. Actualmente estoy trabajando en minería de la evolución de los ecosistemas de software. Un ecosistema de software es un conjunto de proyectos construidos por una comunidad de software, que evolucionan juntos. Si alguien cambia algo en un sistema - por ejemplo, el nombre de un método - eso puede afectar todos los sistemas que usan ese método. Hemos encontrado casos, incluso en pequeñas comunidades, donde un cambio en un sistema puede tardar hasta seis meses antes de ser adoptado en todos sus sistemas dependientes.

El primer paso en la minería de ecosistemas es recuperar las dependencias entre proyectos. Esto no es fácil, ya que la cantidad de datos es grande. Nuestra evaluación de técnicas ligeras para recuperar las dependencias entre proyectos, se ha publicado muy recientemente en la ASE de 2010 [Lungu2010]. En la Figura 6, se pueden ver las dependencias que hemos recuperado entre los proyectos de un ecosistema que tiene más de 200 proyectos distintos.

¡Busco estudiantes!

Con el fin de seguir trabajando en estos temas de investigación, busco estudiantes para supervisar. Por lo tanto, si eres estudiante de maestría o de licenciatura interesado en este ámbito y estás dispuesto a invertir parte de tu tiempo en estos temas de investigación, puedes ponerte en contacto conmigo en: rrobbes@dcc.uchile.cl . BITS

REFERENCIAS

[Lehman1985] M. M. Lehman, L. A. Belady: Program Evolution - Processes of Software Change. Academic Press, London, 1985, pp. 538.

[Bird2009] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. Germán, P. T. Devanbu: The promises and perils of mining git. MSR 2009: 1-10.

[Fischer2003] M. Fischer, M. Pinzger, H. Gall: Populating a Release History Database from Version Control and Bug Tracking Systems. ICSM 2003: 23-32.

[Zimmermann2004] T. Zimmermann, P. Weißgerber, S. Diehl, A. Zeller: Mining Version Histories to Guide Software Changes. ICSE 2004: 563-572.

[Hassan2006] A. E. Hassan, R. C. Holt: Replaying development history to assess the effectiveness of change propagation tools. Empir. Software Eng. 11(3): 335-367 (2006).

[D'Ambros2010] M. D'Ambros, M. Lanza, R. Robbes: An extensive comparison of bug prediction approaches. MSR 2010: 31-41.

[Gîrba2005] T. Gîrba, A. Kuhn, M. Seeberger, S. Ducasse: How Developers Drive Software Evolution. IWPSE 2005: 113-122.

[Anvik2006] J. Anvik, L. Hiew, G. C. Murphy: Who should fix this bug? ICSE 2006: 361-370.

[Zeller1999] A. Zeller: Yesterday, My Program Worked. Today, It Does Not. Why? ESEC / SIGSOFT FSE 1999: 253-267.

[Atkins2002] D. L. Atkins, T. Ball, T. L. Graves, A. Mockus: Using Version Control Data to Evaluate the Impact of Software Tools: A Case Study of the Version Editor. IEEE Trans. Software Eng. 28(7): 625-637 (2002).

[Robbes2005] R. Robbes, M. Lanza: Versioning Systems for Evolution Research. IWPSE 2005: 155-164.

[Robbes2010a] R. Robbes, M. Lanza: Improving code completion with program history. Autom. Software. Eng. 17(2): 181-212 (2010).

[Robbes2008a] R. Robbes: Of Change and Software. Ph.D. Thesis, University of Lugano, 210 pp.

[Robbes2008b] R. Robbes, M. Lanza: How Program History Can Improve Code Completion. ASE 2008: 317-326.

[Robbes2008c] R. Robbes, M. Lanza: SpyWare: a change-aware development toolset. ICSE 2008: 847-850.

[Robbes2007] R. Robbes: Mining a Change-Based Software Repository. MSR 2007: 15-23.

[Robbes2010b] R. Robbes, D. Pollet, M. Lanza: Replaying IDE interactions to evaluate and improve change prediction approaches. MSR 2010: 161-170.

[Bacchelli2010] A. Bacchelli, M. Lanza, R. Robbes: Linking e-mails and source code artifacts. ICSE (1) 2010: 375-384.

[Bacchelli2009] A. Bacchelli, M. D'Ambros, M. Lanza, R. Robbes: Benchmarking Lightweight Techniques to Link E-Mails and Source Code. WCRE 2009: 205-214.

[Lungu2010] M. Lungu, R. Robbes, M. Lanza: Recovering inter-project dependencies in software ecosystems. ASE 2010: 309-312.