DeloreanJS: Un Debugger en el Tiempo para JavaScript

Paul Leger
Escuela de Ingeniería
Universidad Católica del Norte
Coquimbo, Chile
pleger@ucn.cl

Felipe Ruiz
Escuela de Ingeniería
Universidad Católica del Norte
Coquimbo, Chile
felipe.ruiz@alumnos.ucn.cl

Guillermo Victorero
Escuela de Ingeniería
Universidad Católica del Norte
Coquimbo, Chile
guillermo.victorero@alumnos.ucn.cl

***Resumen*—Aplicaciones Web, usando JavaScript, son desarrolladas cada vez con mayor frecuencia. Como en la mayoría de los entornos de desarrollos, una aplicación Web puede adquirir defectos de software (conocido como *bugs*), cuyos síntomas se aprecian durante el desarrollo e incluso, siendo peor, en producción. Para ello, el uso de debuggers es sumamente útil para detectar bugs. Lamentablemente, los actuales debuggers solamente avanzan hacia adelante en la ejecución para detectar un bug y no permiten retornar hacia un punto anterior en la ejecución para tomar acciones asociadas al bug detectado. Por ejemplo, probar si el mismo bug podría gatillarse con otro valor de una variable. Usando el concepto de continuaciones, este artículo presenta un debugger para JavaScript, llamado DeloreanJS, que permite a un programador volver en el tiempo en la ejecución con el fin pueda volver y probar el contexto alrededor de un bug. Este debugger ha sido implementado como una prueba de concepto para ser probado online.**

***Index Terms*—DeloreanJS, JavaScript, Debugger, Web application, Continuations**

I. Introducción

Para construir aplicaciones Web, uno de los lenguajes más usados es JavaScript, cuya presencia en el entorno de la Web es de alrededor del 95 % [1]. Asimismo, la industria del software se mueve fuertemente hacia el desarrollo de este tipo de aplicaciones, siendo testigo un gran número de migraciones de aplicaciones *standalone* a la Web; los ejemplos van desde convertir un documento Word a un formato PDF [2] hasta un sistema online ERP (*Enterprise Resource Planning*) [3]. Por ello, cada vez estas aplicaciones se han vuelto más complejas y con mayor riesgo de introducir defectos de software (*a.k.a. bugs*).

Detectar y reparar bugs representa una de las tareas más costosa en el proceso de desarrollo de software, y las aplicaciones Web no son la excepción. Para aliviar el proceso de tratamiento de bugs, un conjunto de debuggers han sido propuestos, desde el que incluye un navegador en su distribución (*e.g.,* Mozilla Firefox developer tools) hasta debuggers omniscientes que pueden recorrer a través de la traza de ejecución de una aplicación para encontrar las causas de un bug [4], [5]. Considerando estos últimos debuggers, ellos son *post-morter*, y es decir, solo permiten mostrar la ocurrencia de un bug sin entregar la posibilidad de retroceder *en el tiempo* las veces necesarias para que un programador pueda repararlo mientras la aplicación se ejecuta o manipular los valores de las variables alrededor del bug para mejorar la comprensión de su causa.

Este artículo científico presenta DeloreanJS, un debugger que permite al desarrollador de una aplicación Web escrita en JavaScript expresar al insercción de *timepoints* (en cambio de *breakpoints*) para entregar la posibilidad retornar en el tiempo a esos puntos y modificar valores de variables para continuar la ejecución del programa. Este novedoso enfoque de DeloreanJS permite tres características:

1. **Reparación de un bug durante una ejecución.** Mantener una aplicación Web funcionando usando los timepoints para potencialmente reparar un bug durante la ejecución. Lo anterior significa que no es siempre necesario detener la ejecución de la aplicación con el fin de realizar un analísis post mortem.
2. **Comprensión de un bug.** Modificar reiteradamente los valores de las variables asociado a un bug encontrado para mejorar la comprensión de éste. Lo anterior puede ahorrar un gran número de ejecuciones usando un contexto similar para descubrir la verdadera razón del bug.
3. **Experimentación de hipotéticos escenarios.** Probar escenarios hipotéticos de la ejecución de una aplicación Web usando un timepoint de DeloreanJS. Esto permite explorar diversas evoluciones de la ejecución de la aplicación cambiando valores de algunas de sus variables.

Para construir DeloreanJS, utilizamos la abstracción *continuación* [6] de los lenguajes de programación funcional como Scheme [7]. Una continuación permite al programador capturar y guardar, como valor del lenguaje, un momento en la ejecución (*contador de programa* y *pila*) de una aplicación. Extendiendo esta abstracción para lenguajes no funcionales como JavaScript, nosotros habilitamos a DeloreanJS para que ofrezca la posibilidad a un programador de expresar la insercción de timepoints en una aplicación Web, y así poder volver esos momentos de ejecución y modificarlos cuando es requerido.

El artículo está organizado como sigue. La sección II presenta diferentes ejemplos de aplicación de DeloreanJS, donde se puede apreciar en ejecución de nuestra propuesta. Luego, se presenta DeloreanJS, detallando cómo se usa y extiende con-

tinuaciones. En la sección IV se discuten debuggers similares. Finalmente, sección V concluye y describe lineamientos sobre el trabajo futuro de nuestra propuesta.

Disponibilidad. El código fuente de la implementación de DeloreanJS se encuentra en `http://github.com/fruizrob/delorean`. Además, los ejemplos presentados aquí se pueden probar online sin la necesidad de una ninguna extensión en el sitio Web de nuestra propuesta `http://pleger.cl/sites/delorean` [8].

II. Un Tour por DeloreanJS

Nosotros presentamos DeloreanJS a través de tres ejemplos concretos. Cada ejemplo aborda una de las tres características de nuestra propuesta y pueden ser probados (y modificados) online en [8].

II-A. Reparando un Bug

Para introducir DeloreanJS, usaremos un simple código fuente (Listing II-A), el cual está escrito en JavaScript. Esta pieza código muestra que al inicio un programador debe definir qué variables desea observar (ej. x). Luego el programador puede insertar *timepoints* para permitir que la ejecución de una aplicación Web vuelva en el tiempo cuando una excepción es gatillada. En este código, la excepción es gatillada cuando una función inexistente es llamada.

```
DeloreanJs.watch(x,z);
a = 1;
x = 5;
z = 2;

DeloreanJs.insertTimePoint("TP");
a = a + 1;
x = 0;
z = a + 3;

if (z == 5)
  inexistenteFuncion(); //excepcion gatillada
```

II-B. Mejorando la Comprensión de un Bug

PL *Explicar con imagénes el ejemplo 2* □

II-C. Experimentando con Escenarios Hipotéticos

PL *Explicar con imagénes el ejemplo 3* □

III. DeloreanJS

Dado que DeloreanJS propone un novedoso enfoque con respecto a los existentes debuggers debido a característica de volver en el tiempo, esta sección describe cómo funciona.

La figura 1 muestra las multiples trazas de ejecución, llamadas *timelines*, que pueden ser producidas por los timepoints en una aplicación Web. Cuando un programador inserta un timepoint, se puede volver a ese timepoint durante la ejecución en el momento que una excepción es gatillada, y así crear una nueva timeline. Cada timeline representa una traza de ejecución distinta, por ejemplo, las variables x e y de la figura contienen distintos valores. A su vez, cada timeline puede generar otras timelines debido al uso de los timepoints insertados. A continuación, se describe cómo se crea, inserta y usa un timepoint en una aplicación.

III-A. Construyendo Timepoints

Para permitir volver en el tiempo en una ejecución en DeloreanJS, la abstracción de un timepoint es crucial (Figura 1). Esto es porque un timepoint captura y almacena el estado de la ejecución de una aplicación Web en términos del *program counter* (contador de programa), *stack* (pila), y *heap* (datos). En la figura 2 se puede apreciar que el program counter y stack es capturado usando continuaciones [6] y el heap es capturado a través de un análisis estático usando Babel [9].

III-A1. Capturando Program Counter y Stack: Para entender cómo estos dos elementos son capturados y almacenados, es necesario entender continuaciones. Los lenguajes de programación funcionales como Scheme [7], los cuales se caracterizan por tener variables *no mutables*, proveen una abstracción llamada continuación. Esta abstración captura el program counter y stack de un programa funcional, y lo guarda como un *valor de primera clase*, es decir, un valor que soporta operaciones de asignación e invocación (ej. funciones en JavaScript). Cuando una continuación es creada y guardada, ésta puede ser invocada para reemplazar la actual ejecución del programa por la que está guardada en la continuación. Unwinder [11] es una librería en JavaScript que soporta continuaciones a través de una función llamada callCC. Nosotros ejemplificamos continuaciones con Unwinder usando una pieza de código que captura la ejecución de una función que suma dos números:

```
1  var kont;
2
3  function sumar(x,y) {
4    return x + (
5          function() { kont = callCC(cont => cont);
6                       return typeof(kont) == "number"? kont:y;})();
7  }
8
9  mostrar(sumar(5,1));                          //muestra 6
10 if (typeof(kont) == "function") kont(20);     //muestra 25
```

La figura 3 muestra el diagrama de flujo de la pieza de código anterior. En la línea 5, una continuación kont es creada antes de sumar la variable y. Esta captura toma lugar en la línea 9 cuando la función sumar es llamada. El resultado de sumar es pasado a mostrar, y se muestra el número 6. La línea 10 invoca la continuación asociada a kont con el parámetro 20, produciendo que el valor retornado de la función anónima entre las lineas 5-6 sea 20 y no 1. Como resultado, se muestra 25 (25 = (x = 5) + (y = 20)). Notar que el *if expression* (línea 6) y *if statement* (línea 10) son usadas para diferenciar entre la creación de una continuación y su invocación. Una continuación es una función (línea 9) cuando es creada, y cuando es invocada es enlazada al valor pasado como parámetro (ej. el valor 20 en esta pieza de código).

III-A2. Capturando el Heap: Un timepoint también necesita capturar y almacenar las variables que se encuentran en el heap, pero lamentablemente una continuación no lo

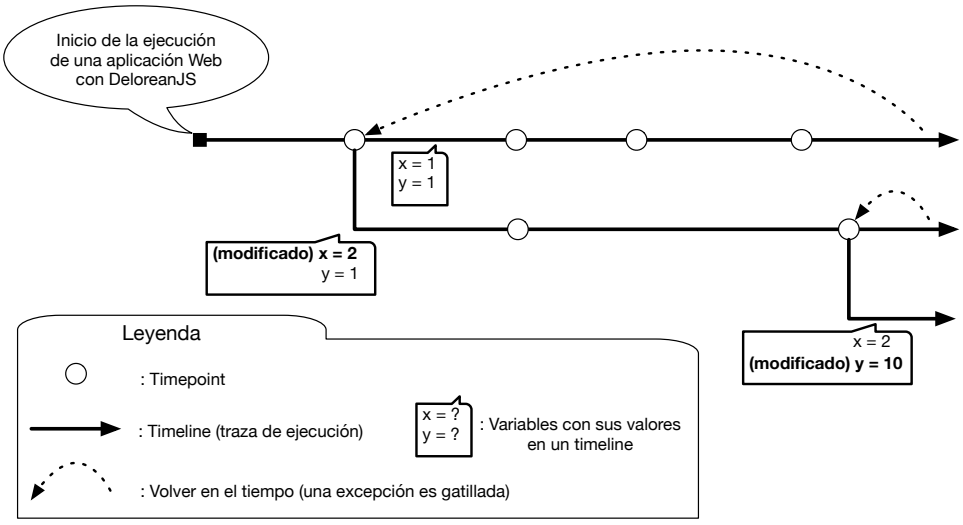

Figura 1. Multiples trazas de ejecuciones de la ejecución de una aplicación con DeloreanJS.

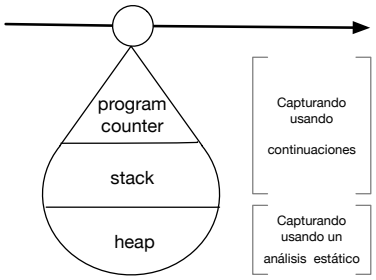

Figura 2. Composición de un timepoint, el cual es creado usando continuaciones y analísis estático del código.

realiza. Para lograrlo, antes de ejecutar una aplicación Web, DeloreanJS realiza un análisis estático para capturar las *variables observables*, definidas por un programador, del heap. La figura 4 ejemplifica las tres etapas que debe realizar el análisis estático para capturar las variables. En la etapa 1, se definen las variables del heap que deben ser observadas (ej. las variables `x` y `z`). Luego en la etapa 2, se captura y almacena los valores de las variables previamente definidas. Finalmente en la etapa 3, se puede apreciar que las variables que modifican a las variables observadas también son capturadas y almacenadas (la variable `a`). Esta última etapa es necesaria porque se debe asegurar que las variables observadas deben *en lo posible*[1] comportarse igual (es decir, con los mismos valores) cuando una ejecución comienza desde un timepoint seleccionado. En el ejemplo de la figura, si la variable `a` no es capturada, entonces la variable `z` tendría un valor distinto cuando una ejecución vuelva a comenzar desde el timepoint `TP`.

Para lograr las etapas de la figura 4, DeloreanJS usa el *Visitor pattern* [12] para realizar un análisis estático, usando la librería Babel [9], como se muestra en la figura 5 . Con esta librería, nuestra propuesta examina cada sentencia del código fuente de la aplicación Web para agregar en un objeto en JavaScript las modificaciones de las variables observadas y otras variables que las modifican.

[1]No es posible si el valor de la variable depende de un evento no determinístico, ej. uso de una función random.

III-B. Insertando y Usando Timepoints en una Aplicación

Durante la ejecución de una aplicación Web usando DeloreanJS, un desarrollador puede *insertar* timepoints, lo cuales se pueden *usar* posteriormente cuando una excepción es gatillada. La figura 6 muestra lo que ocurre cuando un programador llama al método `DeloreanJs.insertTimePoint(String)` y luego posteriormente lo usa. Cuando se inserta un timepoint, nuestra propuesta crea y encapsula en un objeto `TimePoint` una nueva continuación y el objeto que almacena las variables del heap observadas con sus dependencias. Cuando el programador selecciona un particular timepoint (ej. `TP`), DeloreanJS invoca la continuación guardada en el timepoint y posteriormente modifica los valores de las variables observadas junto a sus dependencias.

IV. Trabajos Relacionados

Debido a que JavaScript es un lenguaje muy ampliamente usado, existe un interés constante en crear avances en términos de investigación [13]–[17] y desarrollo [9], [18]–[20]. En estos avances, varios tipos de propuestas de debuggers son posible encontrar en la literatura. Algunos de ellos [21]–[23] ofrecen un amplio conjunto de características como modificar los valores de variables mientras la aplicación se está ejecutando (ej. FireBug [21]). Aunque, en lo mejor de nuestro conocimiento, no es encontrar debuggers que sigan un enfoque similar a DeloreanJS, existen algunos que consideran la historia de ejecución de una aplicación Web:

Debuggers omniscientes. Estos tipos de debuggers se encargan de registrar cada evento que ocurre en la ejecución de un programa, creando un historial de la traza de ejecución. Estos debuggers han sido implementados en lenguajes como en

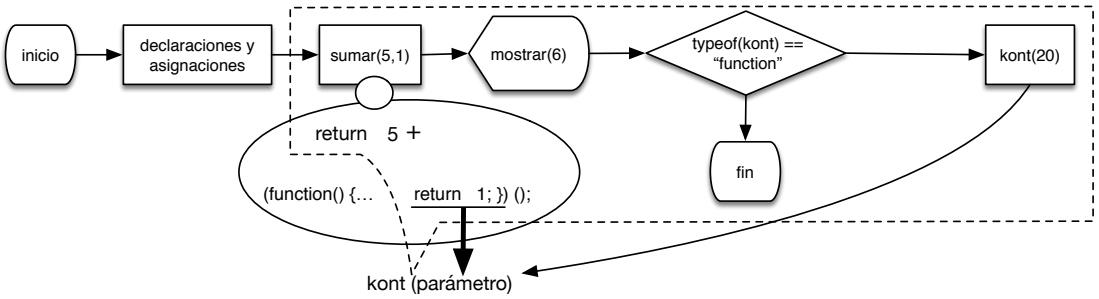

Figura 3. El diagrama de flujo de la pieza código ejemplifica continuaciones usando una continuación kont. La invocación de kont usa un parametro para reemplazar el retornado valor de la función anónima dentro de sumar (Figura elaborada desde [10]).

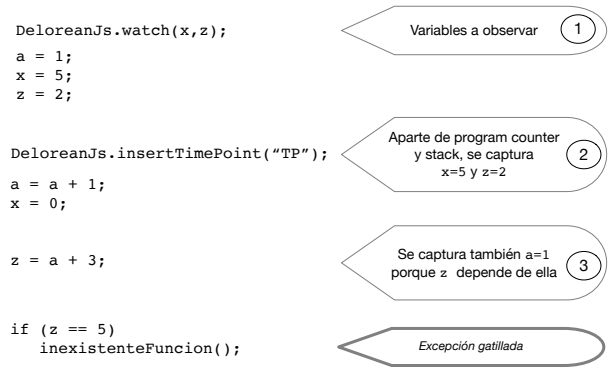

Figura 4. Las tres etapas para capturar y almacenar las variables observadas (ej. x y z) y las variables que pueden modificar a ellas (ej. la variable a porque modifica z).

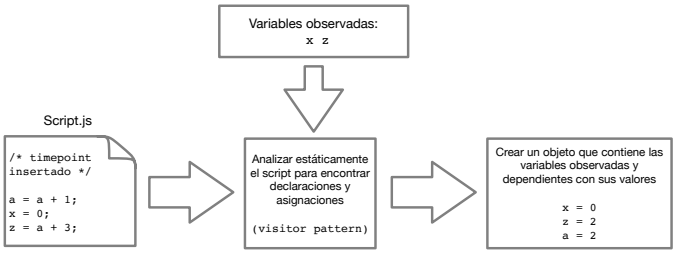

Figura 5. Proceso del análisis estático para registrar cada cambio que pudiesen recibir las variables observadas (Figura elaborada desde el código presentado en Figura 4).

Java [24] y xDSMLs [25]. Con respecto a JavaScript, podemos encontrar PECCit [4] y JARDIS [5], los cuales registran la traza de ejecución y ofrecen diferentes interfaces de usuario para navegar a través de esta traza. A diferencia de DeloreanJS, estos tipos de debuggers son *post-morter*, significando que no es posible volver a un punto en la historia de esta ejecución y continuarla con valores de variables potencialmente modificados.

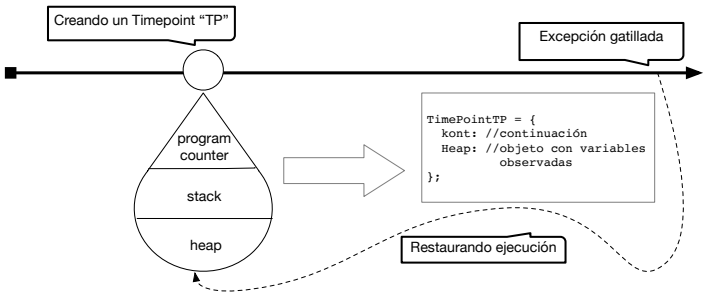

Figura 6. Insercción de un timepoint "TP" y posterior uso.

Debuggers remotos. JavaScript es generalmente[2] usado para construir aplicaciones Web que son usadas por usuarios con algún dispositivo que pertenece a un muy variado y amplio catalogo, pues lo único que se necesita es un navegador con acceso a internet. Por esta razón, cualquier configuración del dispositivo puede producir un potencial bug que los desarrolladores no podrían observar en un ambiente de desarrollo controlado. Para abordar esta dificultad, existen debuggers [28]–[30] que supervisan remotamente las ejecuciones de los usuarios. Similarmente a los debuggers omniscientes, estos registran y envían cada punto ejecución a un desarrollador por la red. Aunque estos debuggers permiten a los desarrolladores trazas de ejecuciones de diferentes usuarios en tiempo real, no ofrecen la posibilidad de retroceder en momentos de la ejecución a los usuarios.

Aunque las herramientas de *Runtime Verification* [31] no son necesariamente debuggers, ellas tienen un comportamiento similar a DeloreanJS porque permiten encontrar errores durante la ejecución de una aplicación. Entre las herramientas disponibles, podemos encontrar PQL [32], PTQL [33] y JavaMOP [34], [35]. Estas herramientas entregan la posibilidad a un programador expresar un patrón avanzado de ejecución, por ejemplo, detectar el acceso a un elemento no disponible en un arreglo porque otro hilo de ejecución lo eliminó. Lamentable-

[2]Aparte de la Web, JavaScript actualmente es usado en varios entornos de desarrollos, por ejemplo, es usado en el lado del servidor en una aplicación Web [26] y en administradores de ventanas de sistemas operativos basados en Linux [27].

mente, de la misma manera que los debuggers omniscientes y remotos, estas herramientas no permiten reanudar la ejecución de una aplicación luego de modificar valores con la intención de reparar el bug o, al menos, mejorar su comprensión.

V. Conclusiones

No es un misterio que la industria del software se esté orientando a construir aplicaciones Web cada vez más grandes y complejas, implicando que exista una mayor probabilidad que aparezcan bugs. Para construir estas aplicaciones, el lenguaje JavaScript es ampliamente usado y una amplia cantidad de debuggers se encuentra disponible en la Web [4], [5], [21]–[23], [28]–[30]. Sin embargo, a diferencia de DeloreanJS, ninguno de estos debuggers considera el enfoque de *volver en el tiempo* a un momento en la ejecución una aplicación escrita en JavaScript. Sin este enfoque, hay un conjunto de oportunidades que un programador pierde, por ejemplo, probar distintos valores de variables en un mismo contexto de ejecución con el objetivo de mejorar la comprensión del bug o explorar escenarios hipotéticos de la evolución de la ejecución dado un mismo contexto inicial. Para alcanzar una adopción por parte de la comunidad de desarrolladores en JavaScript, DeloreanJS tiene aún algunos desafíos:

Capturar completamente el heap. A diferencia del stack que se captura automáticamente con continuaciones, un programador debe ahora especificar qué variables del heap se guardarán en los timepoints. Resolver este desafío implica almacenar (varias) copias de las variables contenidas en el heap por cada timepoint.

Integrar a un existente debugger. Actualmente DeloreanJS ofrece el enfoque de retroceder en el tiempo de una ejecución. Integrar este enfoque con las características de los existentes debuggers podría mejorar significativamente la adopción de nuestra propuesta. Por ejemplo, debuggers omniscientes integrados con DeloreanJS permitirían analizar los registros de multiples trazas de ejecuciones partiendo desde un timepoint.

Como limitación podemos mencionar que DeloreanJS realiza la promesa de volver en el tiempo de una ejecución, la cual podría no cumplirse a totalidad si la aplicación depende de los resultados de eventos externos (ej. servicio Web que entrega la hora actual). Sin embargo, esta limitación se encuentra también presenta en los existentes debuggers.

Referencias

[1] "Usage of client-side programming languages," https://w3techs.com/technologies/history_overview/client_side_language/all, accessed: 2019-03-14.

[2] Smallpdf, "Word to pdf," 2019. [Online]. Available: https://smallpdf.com/word-to-pdf

[3] Oracle, "Oracle erp cloud," 2019. [Online]. Available: http://www.oracle.com/ERP

[4] Z. Azar, "Peccit: An omniscient debugger for web development," Master's thesis, University of Denver, Denver, United States, 2016.

[5] E. Barr, M. Marron, E. Maurer, D. Moseley, , and G. Seth, "Time-travel debugging for javascript/node.js," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Seattle, WA, USA, Nov. 2016, pp. 1003–1007.

[6] D. P. Friedman and M. Wand, "Reification: Reflection without metaphysics," in *Proceedings of the Annual ACM Symposium on Lisp and Functional Programming*, Aug. 1984, pp. 348–355.

[7] R. A. Kesley and J. A. Rees, "A tractable Scheme implementation," *Lisp and Symbolic Computation*, vol. 7, no. 4, pp. 315–335, 1995.

[8] F. Ruiz, G. Victorero, and P. Leger, "Deloreanjs website," Apr. 2019. [Online]. Available: http://pleger.cl/deloreanjs

[9] S. McKenzie, "Babel: A compiler for writing ES6 and ES7 generation JavaScript," 2019. [Online]. Available: https://babeljs.io/

[10] P. Leger and H. Fukuda, "Sync/cc: Continuations and aspects to tame callback dependencies on javascript handlers," in *Proceedings of the 32th Annual ACM Symposium on Applied Computing (SAC 2017)*. Marrakech, Morocco: ACM Press, Apr. 2017, pp. 1245–1250.

[11] J. Long, "Unwinder: A call/cc library," 2018. [Online]. Available: https://github.com/jlongster/unwinder

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, ser. Professional Computing Series. Addison-Wesley, October 1994.

[13] H. Vázquez, A. Bergel, S. Vidal, A. Díaz, and C. Marcos, "Slimming javascript applications: An approach for removing unused functions from javascript libraries," *Information and Software Technology*, vol. 107, pp. 18–29, 2019.

[14] P. Leger, É. Tanter, and R. Douence, "Modular and flexible causality control on the web," *Science of Computer Programming*, vol. 78, no. 9, pp. 1538–1558, Sep. 2013. [Online]. Available: http://pleiad.cl/weca

[15] P. Leger, E. Tanter, and H. Fukuda, "An expressive stateful aspect language," *Science of Computer Programming*, vol. 102, no. 0, pp. 108–141, May 2015.

[16] Y. Zheng, T. Bao, and X. Zhang, "Statically locating web application bugs caused by asynchronous calls," in *Proceedings of the 11th International World Wide Web Conference (WWW 2011)*. Hyderabad, India: ACM Press, Mar. 2011, pp. 805–814.

[17] N. ten Veen, D. Harkes, and E. Visser, "JSExplain: A double debugger for javascript," in *Proceedings of the 2018 Web Conference Companion (WWW 2018)*. Lyon, France: ACM Press, Apr. 2018, pp. 691–699.

[18] jQuery Foundation, "jQuery: A JavaScript library to manage event handling, animating, and Ajax interactions for the Web development. http://jquery.com/." [Online]. Available: http://jquery.com/

[19] "Angular: A framework to build web applications," 2019. [Online]. Available: https://angular.io/

[20] RxJS, "Reactive extensions for javascript," 2018. [Online]. Available: https://rxjs.dev

[21] J. Barton and J. Odvarko, "Dynamic and graphical web page breakpoints," in *Proceedings of the 11th International World Wide Web Conference (WWW 2011)*. Hyderabad, India: ACM Press, Mar. 2011, pp. 81–90.

[22] JsBin, "A open source framework to develop and debug javascript applications," 2019. [Online]. Available: https://jsbin.com/

[23] NodeJS-Inspector, "A open source framework to develop and debug javascript applications," 2019. [Online]. Available: https://jsbin.com/

[24] G. Pothier, É. Tanter, and J. Piquer, "Scalable omniscient debugging," in *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2007)*, Montreal, Canada, Oct. 2007, pp. 535–552.

[25] E. Bousse, J. Corley, B. Combemale, J. Gray, and B. Baudry, "Supporting efficient and advanced omniscient debugging for xDSMLs," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, Pittsburgh, PA, USA, Oct. 2015, pp. 137–148.

[26] NodeJS, "A javascript runtime built for the server side," 2018. [Online]. Available: https://nodejs.org

[27] GJS, "Javascript bindings for gnome," Mar. 2019. [Online]. Available: https://gitlab.gnome.org/GNOME/gjs/wikis/Home

[28] P. Vallet, "SessionStack: An online monitor of javascript applications." [Online]. Available: https://www.sessionstack.com/

[29] B. Wark, "Raygun: Real user monitoring remotely." [Online]. Available: https://raygun.com/

[30] TrackJS, "A JavaScript tracking error monitoring. https://trackjs.com/." [Online]. Available: https://trackjs.com/

[31] P. O. Meredith, "Efficient, expressive, and effective runtime verification," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Aug. 2012.

[32] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using PQL: a program query language," in *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2005)*, San Diego, California, USA, Oct. 2005, pp. 365–383.

[33] S. F. Goldsmith, R. O'Callahan, and A. Aiken, "Relational queries over program traces," in *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2005)*, San Diego, California, USA, Oct. 2005, pp. 385–402.

[34] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu, "An overview of the MOP runtime verification framework," *International Journal on Software Techniques for Technology Transfer*, vol. 14, no. 3, pp. 249–289, 2011.

[35] F. Chen and G. Roşu, "MOP: An efficient and generic runtime verification framework," in *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2007)*, Montreal, Canada, Oct. 2007, pp. 569–588.