Evaluación de la calidad en patrones de arquitecturas software para sistemas críticos ferroviarios: un enfoque basado en la mantenibilidad

Joaquín Acevedo Duprat[1], Emanuel Irrazábal[1],

[1] Facultad de Ciencias Exactas y Naturales y Agrimensura.
Universidad Nacional del Nordeste. Corrientes, Argentina.

{cruz95ad, emanuelirrazabal}@gmail.com

Resumen. El desarrollo de software crítico para sistemas ferroviarios necesita garantizar la seguridad a la vez que la mantenibilidad; un fallo puede ocasionar graves consecuencias para el entorno, especialmente pérdida de vidas humanas. En particular, en la normativa EN 50128 de software crítico ferroviario se especifican las buenas prácticas al construir la arquitectura del software. Pero los patrones de arquitectura que cumplen con las buenas prácticas antes mencionadas pueden ser complejos y, a su vez, disminuir aspectos relacionados con la calidad del producto software, especialmente la mantenibilidad. En este trabajo se seleccionaron un conjunto de patrones arquitecturales, se realizaron pruebas de concepto para cada arquitectura evaluándose los resultados en cuanto a complejidad y características del código fuente resultante. De esta manera se pudo obtener un resultado comparable entre los diferentes patrones: el nivel de seguridad asociado, la complejidad de su construcción y métricas relacionadas con la mantenibilidad. Los resultados muestran que existen patrones con baja complejidad y buena mantenibilidad que pueden ser utilizadas en la construcción de productos software según la norma EN 50128.

Palabras clave: EN 50128, mantenibilidad, arquitectura, sistemas críticos.

1 Introducción

Los sistemas críticos son aquellos compuestos por software y hardware dedicado en los cuales un fallo o mal funcionamiento puede ocasionar graves consecuencias para el entorno, especialmente pérdidas materiales, medioambientales o de vidas humanas [1]. Por esta razón dichos sistemas tienen una serie de propiedades para garantizar un comportamiento esperado con niveles de seguridad, fiabilidad, mantenibilidad y disponibilidad altos [2].

En particular, los sistemas ferroviarios son complejos, compuestos por distintos componentes software, hardware y humanos, que interactúan con su entorno de maneras muy variadas. Un fallo en uno de estos componentes o subsistemas puede llegar a tener asociados distintos niveles de peligros, pudiendo causar pérdidas financieras, daño al equipamiento, daños ambientales, lesiones a personas o en los

peores casos pérdidas de vidas humanas [3]. Por estos motivos dichos sistemas se encuentran regulados con normativas cuyo fin es preservar los recursos anteriormente mencionados.

Una normativa importante para el desarrollo de software crítico ferroviario es la norma EN 50128 [4]. En la misma se trata la calidad del software en términos del ciclo de vida del desarrollo, especificando los procedimientos y requisitos técnicos. Abarca los aspectos de seguridad en cinco niveles desde el 0 al 4, de acuerdo a la gravedad de las consecuencias del fallo, siendo el nivel 4 el que tiene asociado una mayor cantidad de técnicas y características a cumplir. A estos niveles de seguridad se los conoce con la sigla en inglés SSIL derivada de la expresión *software safety integrity level*.

En particular, en el apartado 7.4 de la norma EN 50128 se especifican las buenas prácticas al construir la arquitectura del software en términos de actividades, documentación, especificación integral de cada módulo de la arquitectura y uso de buenas prácticas de programación.

En este sentido, los patrones de arquitectura que cumplen con las buenas prácticas antes mencionadas pueden ser complejos [5] y, a su vez, disminuir aspectos relacionados con la calidad del producto software, especialmente la mantenibilidad [6].

En este artículo se ha llevado adelante una revisión de la bibliografía para seleccionar las arquitecturas software habituales al desarrollar sistemas críticos. Se diseñó una prueba de concepto para cada arquitectura y se evaluaron los resultados en cuanto a complejidad y características del código fuente resultante. De esta manera se pudo obtener un resultado comparable entre los diferentes patrones de arquitectura para software crítico, el nivel de seguridad asociado, la complejidad de su construcción y el grado de mantenibilidad resultante.

El trabajo está organizado de la siguiente manera: además de esta introducción la sección 2 se refiere a la selección de las herramientas y las arquitecturas software escogidas. En la sección 3 se detalla el diseño del experimento llevado adelante para evaluar la construcción de ejemplos con las arquitecturas seleccionadas. En la sección 4 se presentan los resultados de cada desarrollo y el análisis estático del código fuente. Finalmente, en la sección 5 se describen las conclusiones y los trabajos futuros.

2 Selección de las arquitecturas y las herramientas de análisis

A continuación se presentan otros trabajos relevantes relacionados con el desarrollo de arquitecturas software para sistemas críticos ferroviarios que han sido adecuadamente comprobados. En la tabla 1 se identifican los patrones tratados en cada publicación.

Las arquitecturas encontradas fueron las siguientes: programación de N versiones (PNV), aceptación por votos con redundancia de hardware (AV), bloque de recuperación (BR), maestro-esclavo (ME) y redundancia de hardware (RH).

Tabla 1. Selección de patrones de arquitectura software utilizados en sistemas críticos.

Publicación / patrones	PNV	AV	BR	ME	RH
[7]	X	X	X		
[8]		X			X
[9]		X		X	
[10]			X		
[11]			X	X	
[12]			X		
[13]	X		X		
[14]	X	X	X		X

.En la arquitectura programación de N versiones se trabaja sobre versiones equivalentes relacionadas a un mismo módulo funcional que estarán desarrolladas por distintos grupos. Cada versión envía la salida a un módulo votador que procesa las entradas y selecciona el valor final de salida (ver Fig. 1).

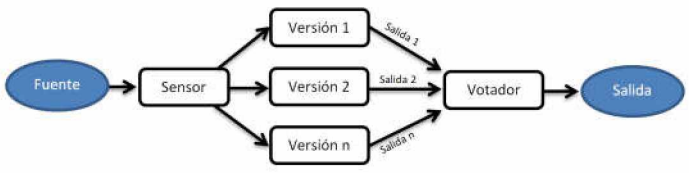

Fig. 1. Programación de N versiones.

En la arquitectura de aceptación por votos con redundancia de hardware se agrega un módulo de test en la salida de cada sensor para identificar fallos en las mediciones o enviar el valor aceptado al votador, dicho módulo es dinámico ya que puede tener diferente cantidad de entradas.

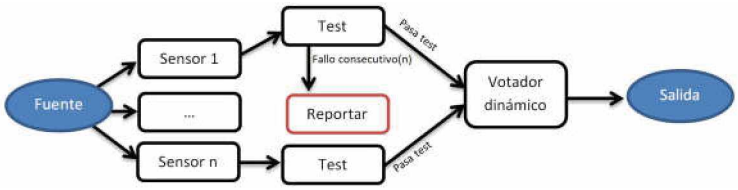

Fig. 2. Arquitectura de aceptación por votos con redundancia de hardware.

En la arquitectura de bloque de recuperación se establecen tres instancias. Una primera instancia define el test de aceptación, el mismo sirve de parámetro para

evaluar si una entrada es correcta o no. En una segunda instancia, se captura y almacena una medición, esta última cumple el rol de punto de verificación o *checkpoint*. Esta segunda medición capturada se procesa con la primera versión del programa y se evalúa con el test. En caso de que sea aceptada, la medición se envía a la salida. En caso de que falle el test, se envía el *checkpoint* a la siguiente versión y se repite la misma evaluación. Si todos los puntos de test fallan se considera una falla total y la medición *checkpoint* es errónea.

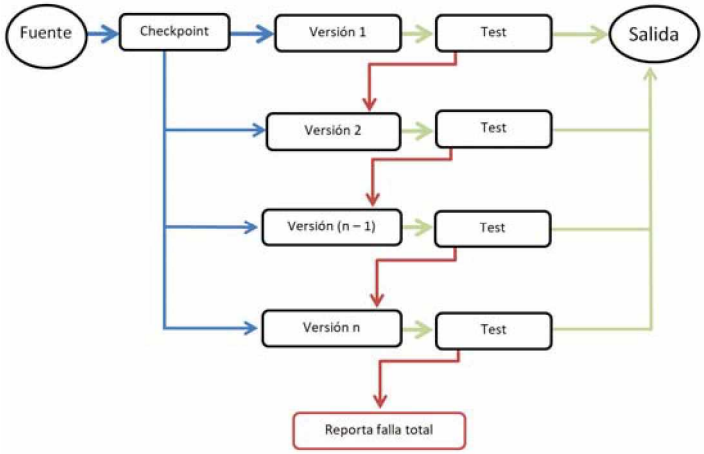

Fig. 3. Arquitectura de bloque de recuperación.

La arquitectura maestro-esclavo con *hot-swap* cuenta con módulos que cumplen dos roles diferenciados. El módulo esclavo toma la señal de la fuente y la procesa, cada esclavo además se da de baja luego de un número definido de fallos consecutivos. El módulo de maestro solicita esa información y trata cada una en un votador para enviar una salida correcta.

Fig. 4. Arquitectura de maestro esclavo.

La arquitectura de redundancia de hardware es de tipo convencional ejecutando una sola versión de software. Simplemente cuenta con la captura de la señal del sensor y un solo tratamiento software para cada entrada redundada.

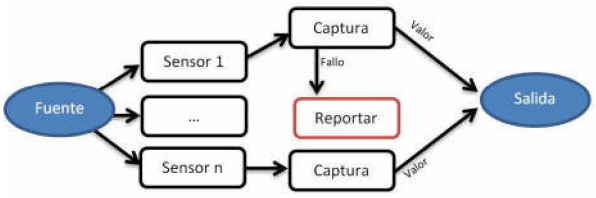

Fig. 5. Arquitectura de redundancia de hardware.

2.1 Selección de las herramientas de análisis estático

Como se indicó al inicio, la normativa EN 50128, que se encuentra bajo la normativa general de sistemas críticos ferroviarios EN 50126, busca garantizar la fiabilidad, disponibilidad, mantenibilidad y seguridad del sistema. En particular, según el estándar EN 50128 sección 9.2.4.4, la mantenibilidad debe diseñarse como un aspecto inherente del software, y en vinculación con normas de calidad del producto software, como, por ejemplo la ISO/IEC 9126.

La selección de un patrón de arquitectura tiene consecuencias en la calidad del código fuente, y, por lo tanto en la mantenibilidad del producto software resultante [15]. Es decir, la selección de una arquitectura solamente basada en la seguridad es un enfoque parcial. Y ante diferentes patrones que pueden cumplir un nivel SSIL alguno de ellos podrá generar código fuente más mantenible que otro.

En este sentido, existen diferentes características a tener en cuenta. De acuerdo con los anexos A.8, A.10 y A.19 de la normativa 50128 es necesario realizar un análisis estático del código fuente asistido mediante herramientas. Para la elección del lenguaje de programación se tuvieron en cuenta las restricciones de arquitectura hardware de la unidad de procesamiento del microcontrolador y el apartado D.54 de la la normativa EN 50128 en la cual se describen los requisitos que debe cumplir el lenguaje de programación. La elección del lenguaje C cumple con los requisitos de rigor mandatorio, esperado y deseable.

Teniendo en cuenta esto se han seleccionado las principales métricas relacionadas con la mantenibilidad a partir de las cuales poder comparar los patrones de arquitectura previamente identificados (ver tabla 2). Las métricas seleccionadas fueron las relacionadas con tamaño y complejidad, teniendo en cuenta desarrollos realizados anteriormente por el grupo de investigación [15]–[17]. Asimismo se tuvieron en cuenta los resultados de herramientas relacionadas con el uso de buenas prácticas de programación.

Junto con cada métrica se seleccionan las herramientas para el análisis estático del código fuente escrito en lenguaje C. Las herramientas se enumeran en la tercera columna.

Asimismo, se sumaron dos herramientas de análisis estático de código fuente orientadas a las normas de codificación. En especial, se incorporó el análisis de las buenas prácticas MISRA C [18], un estándar de facto para el desarrollo de software

crítico. Se eligió más de una herramienta por cada atributo para, de manera secundaria, contrastar las diferencias de medición.

Tabla 2. Selección de métricas y herramientas

Métrica	Descripción	Herramienta
LOC	Líneas de código fuente	CCCC[19] RSM [20] SONAR [21]
MVG	Complejidad ciclomática de McCabe. Es la cuenta de caminos linealmente independientes.	CCCC RSM SONAR
COM	Líneas de texto comentadas	CCCC RSM SONAR
RULES	Reglas de codificación orientadas al buen estilo, para aumentar la analizabilidad del código fuente. O reglas orientadas al uso correcto de las estrategias de programación	CPPCHECK[22] SONAR

3 Diseño del experimento

Habiendo seleccionado las arquitecturas y las herramientas de calidad se realizó el diseño del experimento con el cual se midieron las diferentes características de construcción de los patrones de arquitectura.

El experimento consistió en la construcción del código fuente que representa al patrón de arquitecturas en entornos controlados y homogéneos por parte de cinco grupos diferentes de trabajo. A partir de ello se pudieron medir tanto las características del proceso de desarrollo como las métricas del código fuente.

Resumiendo, el experimento tuvo los siguientes puntos:

- Contexto de trabajo: alumnos de la universidad homogeneizados habiendo realizado una prueba de habilidades en programación con lenguaje C.
- Capacitación previa: se realizó una capacitación previa a los diferentes equipos tanto en programación C como en aspectos de las arquitecturas.
- Desarrollo de los equipos: los equipos fueron elegidos al azar teniendo en cuenta las características de los alumnos. En base al estudio previo y la capacitación se dividieron a los alumnos en dos grandes grupos. Los equipos de trabajo fueron seleccionados con un integrante de cada grupo.
- Asignación de la arquitectura: se trabajó con cinco equipos de dos desarrolladores. cada uno asignado a una arquitectura determinada.

Los equipos tuvieron una semana de trabajo en los que se les proveyó del material necesario para la construcción de los prototipos hardware y el desarrollo del software de acuerdo con las características de cada patrón arquitectural. Tuvieron acceso al material de cada arquitectura para que puedan centrarse en el desarrollo propiamente de la solución.

A lo largo de la semana cada integrante incluía la cantidad de horas que estuvo trabajando en el desarrollo del software (esto incluía tiempo de codificación, tiempo de diseño o tiempo de análisis). Asimismo, al final de la semana cada integrante del equipo pudo evaluar el nivel de complejidad del desarrollo.

Una vez terminados los desarrollos y habiendo controlado el funcionamiento adecuado de cada solución se procedió a analizar el código fuente resultante con las herramientas indicadas en la tabla 2.

4 Resultado

A continuación se resumen los principales resultados y se establece una comparación en cuanto a nivel SSIL. Se ha seguido el apartado A.3 de la norma EN 50128 como criterio de selección, lo indicado en la publicación que presenta el uso de esta arquitectura y el resultado del experimento (ver tabla 3).

La segunda columna identifica que tan recomendable es este patrón para cada nivel de SSIL, siendo:

- DR. Débilmente recomendado para el nivel de seguridad.
- R. Recomendado para el nivel de integridad.
- AR. Altamente recomendado para el nivel de seguridad.

La tercera columna resume el tiempo de desarrollo estimado de acuerdo a las características del patrón. La cuarta columna promedia el grado de dificultad (Dif.) encontrado por parte de los miembros del equipo al construir el patrón (escala de 5 tipo lickert, 1 = muy sencillo, 5 = muy difícil). La quinta y sexta columna define la cantidad de horas invertidas por el equipo en análisis (TA) y desarrollo (TD).

Así, el patrón BR puede ser utilizado para niveles altos de seguridad y su complejidad es menor respecto de AV y PNV.

Tabla 3. Evaluación de los patrones de arquitectura.

Patrón	SSIL 1	2	3	4	Pasos de construcción	Dif.	TA	TD
PNV	DR	R	R	AR	Desarrollo de N versiones	4,2	25	30
AV	DR	R	R	AR	Desarrollo de N versiones más el desarrollo de los tests	4,5	30	35
BR	R	R	R	AR	Desarrollo de N versiones más desarrollo de los tests de forma secuencial	3,3	20	33
ME	DR	R	R	R	Desarrollo del maestro (control) y el esclavo (rutina para verificación del funcionamiento correcto)	3,5	20	22
RH	DR	R	R	R	Desarrollo de solo una versión	2,5	10	15

La arquitectura más sencilla de realizar fue la arquitectura de redundancia de hardware, en cuanto a desarrollo y número de componentes, la versión del programa es una sola y se sirve de la replicación de módulos de sensores. El único grado de dificultad se presenta en el control sobre estos, sin embargo se reduce a la gestión de conexión de los mismos.

La arquitectura más difícil de realizar fue la arquitectura AV. Se debió trabajar con implementaciones alternativas en cada versión para un mismo procesamiento, de manera que estas sean independientes. A su vez, el desarrollo de pruebas de aceptación aumentó la complejidad en cuanto al análisis. En cuanto a las métricas de código fuente, la arquitectura AV fue la de mayor cantidad de líneas de código y mayor complejidad ciclomática

La arquitectura que resultó con la peor relación entre grado de nivel integral de seguridad y dificultad requeridos, fue la arquitectura ME. Se trabajó sobre las funciones delegadas a cada módulo y la dificultad se presenta al trabajar sobre dos equipos físicos distintos sumado al problema de comunicar de forma eficiente a ambas partes.

A continuación se pueden ver los resultados del análisis estático del código fuente para cada arquitectura (ver tabla 4). En general todas las herramientas coincideron en cuanto a las mediciones. Respecto de la cantidad de líneas de código fuente AV y ME fueron los que presentaron mayor cantidad de líneas y mayor complejidad. Asimismo, AV y ME requirieron mayor cantidad de comentarios (la estrategia de desarrollo resultó más compleja). Respecto de las reglas de estilo y buenas prácticas de codificación, PNV y AV son las que concentraron mayor cantidad de violaciones.

Tabla 4. Evaluación de las métricas y atributos asociados a la mantenibilidad.

Métricas / Herramientas	**PNV**	**AV**	**BR**	**ME**	**RH**
RSM (LOC)	286	476	255	505	219
SONAR(LOC)	283	451	247	520	227
CCCC (COM)	62	317	245	299	136
RSM (COM)	69	306	284	298	129
SONAR(COM)	60	308	280	287	124
CCCC (MVG)	54	102	39	85	54
RSM (MVG)	57	106	39	87	52
SONAR(MVG)	55	105	38	84	53
CPPCHECK (RULES propias de estilo)	11	12	6	7	3
SONAR (RULES MISRA C de estilo)	0	1	0	4	2
CPPCHECK (RULES propias tipo “buena práctica”)	0	1	0	0	0
SONAR (RULES MISRA C tipo “buena práctica”)	3	1	0	3	1

5 Conclusiones y Trabajos Futuros

En el trabajo presentado se ha realizado una evaluación de diversas arquitecturas para software crítico ferroviario. Se han presentados los criterios de selección previos, empleando los estándares requeridos. Se realizó el diseño de un experimento para la construcción de las arquitecturas seleccionadas y su posterior evaluación en cuanto a complejidad. También se tuvieron en cuenta métricas de calidad de código fuente para cada implementación realizada, como está previsto en la sección 9.2.4.4 y el anexo A.19 de la norma EN 51280. De esta manera se pudo analizar la relación entre seguridad y mantenibilidad de las arquitecturas.

De las cinco arquitecturas analizadas se destaca la arquitectura PNV para niveles SSIL 3 y 4. Y la arquitectura RH para niveles SSIL 1 y 2. En ambos casos

Las arquitecturas más recomendables para los niveles altos de seguridad, como PNV y AV mantienen una mayor complejidad en cuanto al uso de buenas prácticas de codificación. Y por lo tanto, que pueda ser necesario un equipo de desarrollo más experimentado. En este sentido la arquitectura AV resultó ser mucho más compleja y menos mantenible.

Como trabajo futuro se pretende mejorar el diseño del experimento asegurando homogeneidad en el mismo. Asimismo se buscará aumentar la cantidad de arquitecturas y métricas.

Referencias

[1] D. J. Smith y K. G. Simpson, The Safety Critical Systems Handbook: A Straightforward Guide to Functional Safety: IEC 61508 (2010 Edition), IEC 61511 (2015 Edition) and Related Guidance. Butterworth-Heinemann, 2016.

[2] M. A. Lundteigen, M. Rausand, y I. B. Utne, «Integrating RAMS engineering and management with the safety life cycle of IEC 61508», Reliability Engineering & System Safety, vol. 94, n.º 12, pp. 1894–1903, 2009.

[3] J.-L. Boulanger, CENELEC 50128 and IEC 62279 standards. John Wiley & Sons, 2015.

[4] «Norma UNE-EN 50128:2012». [En línea]. Disponible en: https://www.une.org/encuentra-tu-norma/busca-tu-norma/norma/?c=N0049040. [Accedido: 18-jul-2019].

[5] W. Wu y T. Kelly, «Safety tactics for software architecture design», en Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004., 2004, pp. 368–375.

[6] L. Bass, P. Clements, y R. Kazman, Software architecture in practice. Addison-Wesley Professional, 2003.

[7] M. S. Durmuş, O. Eriş, U. Yildirim, y M. T. Söylemez, «A new voting strategy in Diverse programming for railway interlocking systems», en Proceedings 2011 International Conference on Transportation, Mechanical, and Electrical Engineering (TMEE), 2011, pp. 723–726.

[8] S. Mockel, F. Scherer, y P. F. Schuster, «Multi-sensor obstacle detection on railway tracks», en IEEE IV2003 Intelligent Vehicles Symposium. Proceedings (Cat. No. 03TH8683), 2003, pp. 42–46.
[9] M. Wiest, E. Kassa, W. Daves, J. C. Nielsen, y H. Ossberger, «Assessment of methods for calculating contact pressure in wheel-rail/switch contact», Wear, vol. 265, n.° 9-10, pp. 1439–1445, 2008.
[10] A. Watanabe y K. Sakamura, «MLDD (multi-layered design diversity) architecture for achieving high design fault tolerance capabilities», en European Dependable Computing Conference, 1994, pp. 336–349.
[11] J. P. J. Kelly, T. I. McVittie, y W. I. Yamamoto, «Implementing design diversity to achieve fault tolerance», IEEE Software, vol. 8, n.° 4, pp. 61–71, 1991.
[12] I. I. Courtright, V. William, y G. A. Gibson, «Backward error recovery in redundant disk arrays», CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1994.
[13] U. Voges, Software diversity in computerized control systems, vol. 2. Springer Science & Business Media, 2012.
[14] J.-C. Laprie, «Dependable computing and fault-tolerance», Digest of Papers FTCS-15, pp. 2–11, 1985.
[15] E. Irrazábal, J. Garzás, y E. Marcos, «Alignment of Open Source Tools with the New ISO 25010 Standard-Focus on Maintainability.», en ICSOFT (2), 2011, pp. 111–116.
[16] E. Irrazábal, «Mejora de la mantenibilidad con un modelo de medición de la calidad: resultados en una gran empresa», en XXI Congreso Argentino de Ciencias de la Computación (Junín, 2015), 2015.
[17] E. Irrazábal y J. Garzás, «Análisis de métricas básicas y herramientas de código libre para medir la mantenibilidad», REICIS. Revista Española de Innovación, Calidad e Ingeniería del Software, vol. 6, n.° 3, pp. 56–65, 2010.
[18] L. Hatton, «Safer language subsets: an overview and a case history, MISRA C», Information and Software Technology, vol. 46, n.° 7, pp. 465–472, 2004.
[19] «C and C++ Code Counter download | SourceForge.net». [En línea]. Disponible en: https://sourceforge.net/projects/cccc/. [Accedido: 18-jul-2019].
[20] «M Squared Technologies LLC - RSM for C/C++, Java and C#», msquaredtechnologies. [En línea]. Disponible en: http://msquaredtechnologies.com/index.html. [Accedido: 18-jul-2019].
[21] «Continuous Code Quality | SonarSource». [En línea]. Disponible en: https://www.sonarsource.com/. [Accedido: 18-jul-2019].
[22] «Cppcheck - A tool for static C/C++ code analysis». [En línea]. Disponible en: http://cppcheck.sourceforge.net/. [Accedido: 18-jul-2019].