# Hacia una Formalización de Sistemas Tolerantes a Fallas basada en Grafos Reactivos

Araceli Acosta[1] y Nazareno Aguirre[2]

[1] Facultad de Matemática, Astronomía y Física, Universidad Nacional de Córdoba, Haya de la Torre y Medina Allende, Córdoba (5000), Argentina,
aacosta@famaf.unc.edu.ar

[2] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Ruta 36 Km. 601, Río Cuarto (5800), Córdoba, Argentina,
naguirre@dc.exa.unrc.edu.ar

**Resumen.** En este artículo, presentamos una formalización de fallas y sistemas tolerantes a fallas basada en un nuevo formalismo denominado *sistemas reactivos de transiciones etiquetadas de estados*. La formalización difiere de propuestas anteriores en varios aspectos: *(i)* separa la descripción del sistema de aquella asociada con el comportamiento de la falla (la cual es *superimpuesta* sobre el comportamiento del sistema), *(ii)* provee un significado preciso de la noción de "sistema tolerante a fallas", la cual es dependiente de falla, y requiere una especificación de requisitos del sistema.

## 1 Introducción

La confiabilidad es uno de los atributos de calidad más importantes asociados al desarrollo de software. La corrección de software (es decir, en qué grado el software satisface sus requerimientos) es, tal vez, el factor más importante asociado a la confiabilidad. Sin embargo, en numerosos casos, existen eventos externos fuera del control del sistema que pueden provocar comportamiento no deseado. Ejemplo de este tipo de eventos pueden ser fallas en dispositivos de hardware, cortes de energía eléctrica, fallas de comunicación, etc. En particular, para sistemas críticos, es decir sistemas en los cuales el mal funcionamiendo del software o hardware pueden llevar a drásticas consecuencias (por ejemplo, sistemas de control de plantas nucleares, vehículos espaciales, etc.), razonar acerca de los efectos de estos eventos, y cómo nuestro sistema reacciona a ellos, es de suma importancia.

Esta importancia se refleja en el hecho que la tolerancia a fallas, cuyo objetivo de estudio se centra en cómo modelar, operar y razonar con sistemas tolerantes a fallas, se ha vuelto un área de investigación muy activa en los últimos años. La tolerancia a fallas ha estado asociada desde un principio a dispositivos de bajo nivel y protocolos en sistemas distribuidos; pero en los últimos años se han desarrollado nuevas áreas como las de sistemas *self-healing* y *self-adaptive* (sistemas que tienen la habilidad de reconfigurarse o "curarse" a sí mismos como respuesta a algún evento externo). Estas áreas encaran la tolerancia a fallas desde

una perspectiva más abstracta, también desde una etapa más temprana en el proceso de desarrollo de software.

Presentaremos aquí un nuevo formalismo para hablar de sistemas tolerantes a fallas que difiere de otras propuestas en varios aspectos. Primero, en el enfoque que presentamos se distingue explícitamente la descripción del sistema y la descripción del comportamiento asociado a una falla. El comportamiento asociado a la falla, considerado comportamiento de entorno, se *superimpone* sobre el comportamiento del sistema. Esto prohibe que en la descripción del sistema pueda hacerse uso de eventos que "comuniquen" la ocurrencia de una falla, lo cual sería poco realista. Segundo, esta formalización provee una noción más precisa de tolerancia a fallas, la cual es dependiente de cada falla, y requiere una especificación de los requerimientos del sistema. Esta formalización se basa en una variante de un formalismo propuesto recientemente por D. Gabbay llamado *Modelos reactivos de Kripke* [5]. Este formalismo nos permite expresar la superimposición del comportamiento asociado a la falla sobre el del sistema de una manera natural, y caracterizar una amplia variedad de tipos de fallas y sus consecuencias.

## 2 Conceptos Preliminares y Trabajos Relacionados

En la literatura sobre tolerancia a fallas, existe un consenso sobre cierta terminología. Tres conceptos suelen distinguirse a la hora de hablar de tolerancia a fallas; éstos son las nociones de *falla* (fault), error y *fallo* (failure):

- un fallo (failure) es un comportamiento indeseado del sistema (es decir, una violación a alguno de los requisitos del sistema),
- un error es un estado anormal del sistema, que puede llevar al mismo a un fallo,
- una falla es un evento que puede llevar al sistema a un error.

Se denomina a un sistema *tolerante a fallas* si, incluso en presencia de fallas, el sistema se comporta de la manera esperada (es decir, sin violar requisitos de sistema).

La tolerancia a fallas se asocia generalmente con los sistemas de hardware y redes de computadoras. De hecho, en el diseño de hardware y protocolos de redes, se han diseñado varios mecanismos para proveer cierto comportamiento esperado, a pesar de la ocurrencia ocasional de fallas. Estas fallas pueden deberse a una variedad de razones, tales como ruido electromagnético afectando un circuito integrado, o partículas energéticas afectando el estado de una componente electrónica. Los errores causados por este tipo de fallas se denominan generalmente *errores suaves* (soft errors), y se consideran una de las principales causas de la caída de sistemas [11].

Por supuesto, el interés en sistemas con altos niveles de disponibilidad y confiabilidad ha llevado a un gran desarrollo en el área de la computación tolerante a fallas. Más aún, este interés ha influenciado más recientemente áreas del diseño de software, como se puede observar con el origen de áreas como *self management*, *self healing* y *self adaptive systems* [4] [6] [7] .

Una característica sorprendente de la tolerancia a fallas es la falta de soluciones de propósito general, o ambientes para el análisis de sistemas tolerantes a fallas. La amplia mayoría de las soluciones a problemas de tolerancia a fallas son *ad hoc*, y atacan problemas específicos en contextos específicos. En el caso de técnicas para tolerancia a fallas en software, las soluciones más importantes tienen que ver con redundancia o diversidad. La diversidad puede ser diversidad de diseño, como en *N-version programming* y bloques de recuperación, o diversidad de datos, como en los bloques de reintento y la *N-copy programming* [10]. La redundancia puede ser redundancia temporal, con en el caso de la retransmisión de mensajes, o redundancia espacial, como es el caso de código de corrección de errores.

Por otro lado, los métodos formales atacan principalmente la especificación, el diseño y en general el desarrollo de sistemas de software, y su principal motivación es la de garantizar la corrección del software (es decir, que el mismo satisface sus requisitos). El mecanismo fundamental para lograr esto es la verificación formal. Como los métodos formales contribuyen a la construcción de sistemas libres de bugs, se podría decir que éstos contribuyen también a la construcción de sistemas tolerantes a fallas (los bugs son de hecho una de las causas fundamentales de fallas). Sin embargo, el enfoque tradicional de los métodos formales es insuficiente, dado que no atacan, en general, el problema de lidiar con fallas provenientes de eventos externos al software, no previstos.

Existen algunos enfoques formales a la tolerancia a fallas. Entre ellos podemos citar el trabajo de Arora y Gouda [1], quienes proponen el concepto de auto estabilización. En su trabajo, un sistema de denomina tolerante a fallas si, cuando una falla ocurre, y bajo la hipótesis de que la misma no vuelve a ocurrir, el sistema alcanza eventualmente un estado "seguro". Otro enfoque interesante es el de Magee y Maibaum [8], quienes proponen adoptar una semántica de sistemas basada en máquinas de estados, con diferentes tipos de estados para los sistemas, tales como estados "buenos" (normales) y "malos" (anormales), y un lenguaje para la especificación de requisitos que puede hacer uso de los distintos tipos de estados [9]. Otro trabajo fuertemente relacionado a este último, que utiliza lógica deóntica para la descripción de propiedades de sistemas tolerantes a fallas, es el descripto en [3].

## 3 Una Formalización de Sistemas Tolerantes a Fallas

La formalización que proponemos utiliza sistemas de transiciones de estados etiquetadas como semántica, y por el momento como vehículo para la especificación de sistemas tolerantes a fallas. Más precisamente, utilizamos sistemas de transiciones de estados etiquetadas para caracterizar la combinación de la funcionalidad normal del sistema, y los mecanismos que el desarrollador utiliza para lidiar con las fallas. La descripción del comportamiento de entorno asociado a la falla se "entrelaza" o mezcla con el del sistema, como veremos.

### 3.1 Especificación

La especificación es la descripción formal del comportamiento deseado del sistema. Siempre consiste en una descripción más abstracta a la implementación. Tal descripción es necesaria para poder decir si un sistema es en efecto tolerante a fallas: lo será sólo si ante la presencia de las mismas, el sistema continúa comportándose como es esperado. El comportamiento esperado del sistema lo describiremos mediante un sistema de transiciones etiquetadas de estados (LTS, por las siglas en inglés), es decir, una tupla de la forma $E = \langle S^e, Act^e, \delta^e, S_0^e \rangle$, donde $S^e$ es un conjunto de estados, $Act^e$ es un conjunto de acciones (o etiquetas), $\delta^e \subseteq S^e \times Act^e \times S^e$ es un conjunto de transiciones, y $S_0^e \subseteq S^e$ es un conjunto de estados iniciales. Cada transición $(s, a, s')$ se denota usualmente por $s \xrightarrow{a} s'$. Los LTS suelen describirse mediante grafos, como es usual en otros formalismos como los autómatas finitos.

Notemos que los requisitos del sistema pueden especificarse con una variedad de lenguajes, y es relativamente poco usual tener una descripción de naturaleza operacional, como la que proponemos. Si bien el lenguaje puede parecer poco adecuado, debe tenerse en cuenta que especificaciones formales de requisitos en formalismos de más alto nivel pueden llevarse a esta forma de manera directa. Utilizamos este formalismo para la representación de requisitos por razones de simplicidad en la presentación de las ideas.

*Ejemplo* Para hacer más clara la función de LTS como especificación de requisitos de sistema, consideremos como ejemplo el modelo ampliamente conocido de *Productor-Consumidor*, el cual consiste en dos procesos, un productor y un consumidor, en el cual el productor genera mensajes y los transmite por un canal, y estos mensajes son luego consumidos por el consumidor. Un requisito usual asociado con *Productor-Consumidor* es que todo mensaje producido por el productor es eventualmente consumido por el consumidor. Podemos especificar una versión más restrictiva de esta propiedad como se muestra gráficamente en la figura 1. Esta versión es más restrictiva porque exige que todo mensaje producido es consumido antes de que se produzca el siguiente mensaje (es decir, que la producción y consumo se alternan).

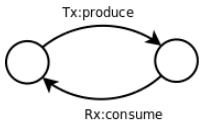


**Fig. 1.** Especificación simple mediante LTS de *Productor-Consumidor*.

### 3.2 Modelo del Sistema (Sin Fallas)

El comportamiento del sistema también será especificado usando un sistema de transición de estados etiquetados. Supondremos entonces que el comportamiento

del sistema está dado por una tupla $Sys = \langle S, Act, \delta, S_0 \rangle$, donde, al igual que para las especificaciones, $S$ es un conjunto de estados, $Act$ es un conjunto de acciones, $\delta \subseteq S \times Act \times S$ es un conjunto de transiciones con etiquetas, y $S_0 \subseteq S$ es el conjunto de estados iniciales. Por razones de simplicidad en la presentación, supondremos en este artículo que, en el modelo $Sys$ del comportamiento del sistema (sin fallas), sucede que $S = Reach_\delta(S_0)$, es decir, que todos los estados del sistema son alcanzables (mediante la aplicación de secuencias finitas de transiciones) desde el conjunto de estados iniciales. Es importante destacar que si bien el formalismo utilizado para especificar los requisitos de sistema y para el modelo (más concreto) del comportamiento del sistema es el mismo, éstos representan conceptos diferentes. Más precisamente, $E$ resultará ser un modelo abstracto de $Sys$ (bajo alguna definición apropiada de abstracción). Dado que $E$ y $Sys$ describen elementos comunes, es razonable pedir que $Act^e \subseteq Act$.

Varias nociones de *refinamiento* han sido definidas sobre este tipo de modelos formales. En este trabajo no nos preocuparemos por cuál de ellas es la más adecuada para nuestro propósito, sino que simplemente supondremos contar con una relación de refinamiento $\leq_r$, que nos permita establecer que $Sys$ satisface $E$ cuando $Sys \leq_r E$.

*Ejemplo* Continuando con el ejemplo anterior, consideremos una implementación de *Productor-Consumidor*, que se muestra gráficamente en la figura 2. En este modelo, la comunicación entre productor y consumidor se realiza mediante un canal confiable (es decir, sin pérdida de mensajes), y se ha añadido detalle respecto de la especificación, pues se tiene ahora eventos para representar la comunicación. El modelo que mostramos es, como sugiere la figura, el resultado de dos procesos concurrentes que se sincronizan con una acción de comunicación.

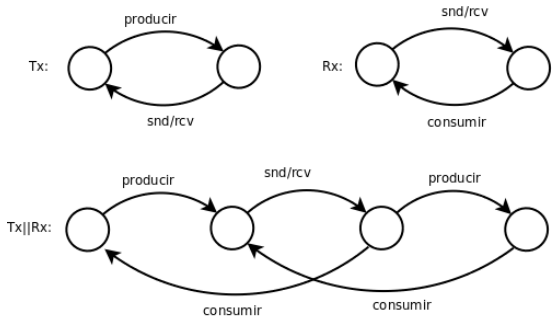


**Fig. 2.** Una implementación de *Productor-Consumidor*, representada mediante LTS.

### 3.3 LTS Reactivos

A diferencia de las descripciones del sistema y los requisitos asociados al mismo, el comportamiento asociado a las *fallas*, pensado como comportamiento de en-

torno del sistema, será descripto mediante una variante de *modelos reactivos de Kripke* [2] [5]. Los modelos reactivos de Kripke son un formalismo similar las estructuras de Kripke introducido recientemente por D. Gabbay. Un modelo reactivo de Kripke, a diferencia de una estructura de Kripke convencional, puede *reconfigurarse* cuando una transición es disparada, cambiando su topología. La variante que utilizaremos la denominamos LTS reactivo, y corresponde a una tupla $R = \langle S, Act, \delta, \delta^+, \delta^-, S_0, \delta_0 \rangle$, donde $S$ en un conjunto de estados, $Act$ es un conjunto de acciones, $\delta \subseteq S \times Act \times S$ es un conjunto de transiciones etiquetadas, $\delta^+ \subseteq S \times Act \times S$ y $\delta^- \subseteq S \times Act \times S$ son conjuntos de *meta-transiciones*, $S_0 \subseteq S$ es un conjunto de estados iniciales, y $\delta_0 \subseteq \delta$ es el conjunto de transiciones habilitadas inicialmente. Intuitivamente, las transiciones $\delta^+ \subseteq S \times Act \times S$ y $\delta^- \subseteq S \times Act \times S$ permiten habilitar e inhabilitar, respectivamente, las transiciones en $\delta$.

La idea principal es la siguiente: dado un sistema $Sys$ (representado como un LTS), la incorporación del comportamiento asociado a fallas sobre $Sys$ dará lugar a un LTS reactivo, el resultado de *superimponer* el comportamiento de entorno sobre $Sys$. Esto quedará más claro en la siguiente sección.

### 3.4 Introduciendo Fallas al Modelo del Sistema

Como ya adelantamos, utilizaremos LTS reactivos para representar sistemas con fallas. Dado un sistema $Sys = \langle S, Act, \delta, S_0 \rangle$, un sistema con una falla $f$ lo representaremos con un LTS reactivo $Sys^f = \langle S \uplus S^f, Act \uplus \{f\}, \delta \uplus \delta^f, \delta^+, \delta^-, S_0, \delta_0 \rangle$, donde $f$ es el evento de falla que se agrega al conjunto de acciones $Act$ de $Sys$, $S^f$ y $\delta^f$ son los estados y transiciones que, como consecuencia de considerar la ocurrencia de la falla $f$, son agregados al comportamiento del sistema.

Notemos que la incorporación del comportamiento de la falla *preserva* las transiciones y estados del modelo de comportamiento $Sys$ del sistema, complementando éste con nuevos estados y transiciones (algunas de las cuales pueden ser meta-transiciones); esto es lo que queremos decir cuando indicamos que la falla se *superimpone* al sistema. Es importante resaltar que $Sys$ debe contener tanto el comportamiento propio del sistema, como los mecanismos de detección y tratamiento de fallas que el desarrollador haya pensado: el comportamiento superimpuesto al sistema sólo tiene que ver con el efecto de las fallas, no su tratamiento.

*Ejemplo* Continuemos con el ejemplo del modelo de *Productor-Consumidor*. Una falla que podríamos incorporar al sistema es la potencial pérdida de mensajes por parte del canal de comunicación. Suponiendo que el evento asociado a esta falla es denotado por $f$, el efecto de la falla sobre el sistema *Productor-Consumidor* puede ser descripto por el LTS reactivo que se muestra en la figura 3.

En este caso, el sistema con la falla superimpuesta al mismo es un LTS convencional: no se agregan meta-transiciones. Notemos que, dado que $f$ es una acción nueva, respecto del sistema original, este evento no es observable en el sistema, aunque si lo es su efecto. De hecho, el sistema sin esta falla satisface la especificación de *Productor-Consumidor*, pero no lo hace agregando la falla al

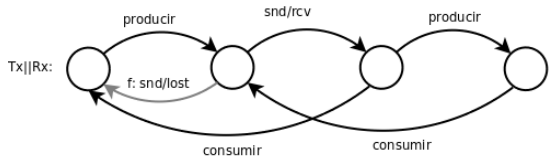


**Fig. 3.** Pérdida de mensajes, representado como falla superimpuesta al modelo simple de *Productor-Consumidor*.

modelo. En resumen, el sistema no es tolerante a la falla introducida, ya que su inclusión en el comportamiento del sistema lleva a una violación de los requisitos del mismo, es decir, a un *fallo* (failure).

*Ejemplo* Consideremos otro ejemplo de falla superimpuesta sobre el sistema de productor-consumidor. Otra falla posible asociada con este sistema es el caso en el cual se sufre una caída total del canal de comunicación. En este caso, el productor producirá y enviará los mensajes, pero el consumidor no los recibirá, es decir, el evento `snd/rcv` no provocará cambios en el consumidor. Podemos modelar esta situación mediante un LTS reactivo que extiende al sistema original, como se muestra en la figura 4. En este modelo, el evento asociado a la ocurrencia de la falla (no observable en el sistema original) es $f'$.

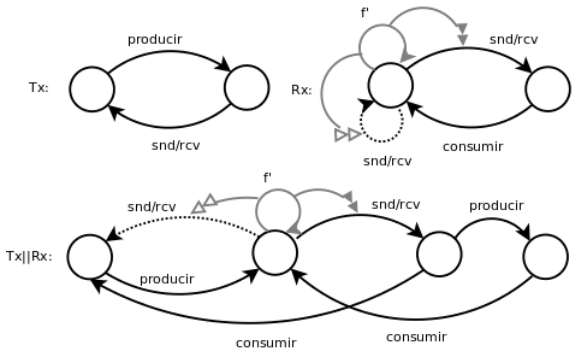


**Fig. 4.** Caída del canal de comunicación, representado como falla superimpuesta al modelo simple de *Productor-Consumidor*.

Hemos denotado con flechas con líneas de puntos a las transiciones que no están habilitadas inicialmente, y en gris las transiciones asociadas a las fallas. Las flechas con cabezas dobles son meta-transiciones, siendo las de cabezas rellenas las pertenecientes a $\delta^-$ y las de cabezas vacías las de $\delta^+$ (las que, al "dispararse", inhabilitan y habilitan otras transiciones, respectivamente). Al igual que en ejemplo anterior, el sistema *Productor-Consumidor* no es tolerante a la falla de caída

del canal de comunicación, ya que el sistema con esta falla superimpuesta viola la especificación del sistema.

## 4 Ejemplos de Definiciones y Propiedades

La formalización de sistemas tolerantes a fallas presentada nos permite realizar algunas tareas importantes. Una de ellas es categorizar de manera natural a las fallas, de acuerdo a la forma en que éstas alteran al sistema original. Daremos una breve descripción de esta categorización en esta sección.

Dado un sistema $Sys^f = \langle S \uplus S^f, Act \uplus Act^f, \delta \uplus \delta^f, \delta^+, \delta^-, S_0, \delta_0 \rangle$, con fallas superimpuestas en él, llamaremos *estados normales* al conjunto $S$, y *estados anormales* al conjunto $S^f$. Una *falla* es la ejecución de una transición $s \xrightarrow{a} s' \in \delta^f$ y $a \in Act^f$.

Una primera observación que podemos realizar sobre las fallas es que éstas pueden tener diferentes tipos de consecuencias sobre el sistema. En nuestro modelo, es evidente que una falla puede provocar alguno de los siguientes sucesos: *(i)* un cambio de estado a un estado normal, *(ii)* un cambio de estado a un estado anormal, *(iii)* habilita y/o inhabilitar transiciones, o *(iv)* una combinación de los anteriores.

Formalmente, diremos que un sistema $Sys$ es *tolerante a la falla* $f$, dada una especificación $E$ del mismo, si y sólo si $Sys^f$ refina $E$. Como mencionamos, no nos preocuparemos por el momento por la noción de refinamiento más adecuada para nuestros propósitos, pero existen muchas en la literatura. Por ejemplo, podríamos considerar *inclusión de trazas de ejecución* como noción de refinamiento; en este caso, $Sys$ será tolerante a la falla $f$ si y sólo si el conjunto de trazas de $Sys^f$, restringido al alfabeto $Act^e$ (el alfabeto de la especificación de $Sys$), está incluido en el conjunto de trazas de $E$.

Mediante la consideración de la noción de traza de ejecución, podemos formalizar la noción de *error*, en el sentido de la tolerancia a fallas. Un error es un estado $e = (s_0, \delta_0)$ (notemos que, en LTS reactivos, un estado en una traza dada está dado por el estado corriente en el LTS, y las transiciones habilitadas en el mismo momento) tal que existe una traza $\sigma$ de ejecución sobre $Sys^f$ y un par de valores enteros $i < j$, tal que $\sigma.i = (s_0, \delta_0))$ ($e$ es el $i$-ésimo estado de $\sigma$) y $\sigma^j$ (el prefijo de los primeros $j$ estados de $\sigma$ no satisface $E$.

En la figura 5 se muestra el problema de productor consumidor que es tolerante a la falla de pérdida de mensajes bajo esta noción de refinamiento. Esto es porque todas las trazas (bajo hipótesis de fairness) que puede generar este sistema, incluyendo la falla, restringido al alfabeto $Act^e$, está incluido en el conjunto de trazas de $E$.

Un ejemplo de un tipo de falla que podemos distinguir fácilmente es el de *fallas instantáneas*. Diremos que una falla es instantánea si y sólo si su ocurrencia no habilita ni inhabilita otras transiciones, es decir, que no tiene consecuencias más allá de la asociada a su momento de ocurrencia. La falla de pérdida de mensajes, por ejemplo, es instantánea. Una condición necesaria para que una falla instantánea sea *detectable* es que su ocurrencia lleve siempre a estados

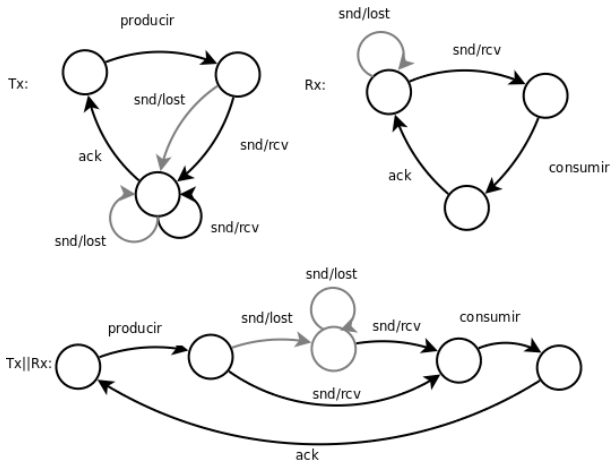


**Fig. 5.** Pérdida de mensaje, representado como falla superimpuesta al modelo con confirmación (ack) de *Productor-Consumidor*. Notar que la pérdida de la confirmación se modela análogamente.

anormales. Este es un ejemplo simple de un tipo de fallas cuya detectabilidad puede analizarse de manera muy simple, mediante alcanzabilidad en $Sys^f$.

## 5 Conclusiones y Trabajo Futuro

El trabajo que hemos presentado está centrado en especificar y diseñar sistemas tolerantes a fallas usando métodos formales. A diferencia de otros trabajos, algunos mencionados en la sección 2, en nuestra propuesta trabajamos con la hipótesis de que la fuente de fallas está fuera del control del sistema, aunque puede, por supuesto, afectar su comportamiento. Una consecuencia de esto es que los eventos asociados a las fallas no siempre son observables y/o controlables por el sistema: es necesario diagnosticar la ocurrencia de la falla a través de *consecuencias observables* y tomar las acciones adecuadas.

En nuestro enfoque, existe una distinción explícita entre la descripción del comportamiento del sistema y la del comportamiento asociado a las fallas, incluso con formalismos particulares para la descripción de cada uno. En el caso del sistema, el mismo se describe con un sistema de transición de estados etiquetados, un formalismo ampliamente utilizado para describir el comportamiento de sistemas concurrentes. Las fallas, y el comportamiento del sistema con fallas, en cambio, se describe mediante la utilización de *LTS's reactivos*.

La noción de tolerancia a fallas introducida requiere que los requisitos del sistema sean descriptos formalmente. En este trabajo, hemos optado por presentar éstos operacionalmente, mediante sistemas de transición de estados etiquetados. La descripción de los "requerimientos" de eventos de fallas no se mencionan aquí, ya que las fallas no son parte del comportamiento deseado, sino consecuencias su

entorno. Uno de los objetivos claves al modelar sistemas y su comportamiento frente a fallas es distinguir estas tareas. Nuestra formalización es un intento por mantener bien diferenciadas las tareas de diseño del sistema, comportamiento de tolerancia a fallas, y comportamiento de las fallas.

Actualmente, estamos trabajando en el análisis de modelos de sistemas y fallas, en particular análisis basado en model checking. Debido a que los LTS reactivos son traducibles a LTS convencionales [5], la aplicación de model checking a modelos con fallas es directa. Sin embargo, intentaremos definir alguna variante de algoritmos de model checking que aprovechen la representación compacta que ofrecen los LTS reactivos. Otro trabajo en curso es la definición de un lenguaje de especificaciones de más alto nivel para sistemas y fallas (cuya semántica, por supuesto, estará dada en términos de LTS y LTS reactivos).

## Referencias


1. Anish Arora and Mohamed G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *Software Engineering*, 19(11):1015–1027, 1993.
2. Walter Carnielli, F. Miguel Dionsio, Paulo Mateus eds, Ficha Tcnica, Compiladores Walter, A. Carnielli, F. Miguel Dionsio, Paulo Mateus, Impresso/acabamentos Grafitese, and Dov Gabbay. Reactive kripke semantics and arc accessibility, 2004.
3. Pablo F. Castro and T. S. E. Maibaum. Reasoning about system-degradation and fault-recovery with deontic logic. In Michael Butler, Cliff B. Jones, Alexander Romanovsky, and Elena Troubitsyna, editors, *Methods, Models and Tools for Fault Tolerance*, volume 5454 of *Lecture Notes in Computer Science*, pages 25–43. Springer, 2009.
4. Rogério de Lemos and José Luiz Fiadeiro. An architectural support for self-adaptive software for treating faults. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 39–42, New York, NY, USA, 2002. ACM.
5. Dov M. Gabbay. Reactive kripke models and contrary to duty obligations. In Ron van der Meyden and Leendert van der Torre, editors, *DEON*, volume 5076 of *Lecture Notes in Computer Science*, pages 155–173. Springer, 2008.
6. David Garlan and Bradley Schmerl. Model-based adaptation for self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 27–32, New York, NY, USA, 2002. ACM.
7. Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-organising software architectures for distributed systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 33–38, New York, NY, USA, 2002. ACM.
8. Jeff Magee and Tom Maibaum. Towards specification, modelling and analysis of fault tolerance in self managed systems. In *SEAMS '06: Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, pages 30–36, New York, NY, USA, 2006. ACM.
9. Tom Maibaum. Temporal reasoning over deontic specifications. pages 141–202, 1994.
10. Laura L. Pullum. *Software fault tolerance techniques and implementation*. Artech House, Inc., Norwood, MA, USA, 2001.
11. Goutam Kumar Saha. Software based fault tolerance: a survey. *Ubiquity*, 7(25):1–1, 2006.