UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Tesis Doctoral

Modelización de la concurrencia en sistemas distribuidos multiagente: Autómatas Cooperativos

autor:

Carlos Herrero Cucó

director:

Dr. Javier Oliver Villaroya

Noviembre de 2015

Dedicada a mis hijas que tanto esfuerzo, dedicación y cariño merecen y tanta alegría y orgullo dejan tras de sí.

Dedicada a Mariola que me aguanta, me ayuda, me acompaña, me sigue, me guía y me quiere. Gracias por estar a mi lado siempre.

Dedicada a mis padres que tanto me han enseñado con su ejemplo y que siempre han luchado con orgullo, honradez y paciencia por mi y mis hermanos.

Dedicada al resto de mi familia y amigos pues cada experiencia vivida juntos, cada alegría y cada pena me han hecho quien soy. Gracias a todos.

Resumen de la Tesis Doctoral

La Tesis Doctoral "Modelización de la concurrencia en sistemas distribuidos multiagente: autómatas cooperativos"tiene como objetivo final la definición y extensión de un modelo para la descripción de sistemas multiagente que aúne características deseables de las Redes de Petriz la "Teoría de Autómatas", así como la definición de un mecanismo automatizable de análisis sobre los sistemas descritos con este modelo que permite la simplificación de los mismos en sub-sistemas y la detección de partes críticas antes incluso de finalizar el modelado. También como objetivo se persigue dotar al sistema de un software de ayuda que automatice las partes de diseño, de análisis y de simulación de los sistemas descritos.

Como aportaciones de la Tesis Doctoral se muestra la potencia expresiva del modelo de los Autómatas Cooperativos (AC) a la hora de describir sistemas concurrentes, además partiendo del modelo de los AC se realiza una comparativa de los mismos con otros modelos presentes en la literatura, como son los ."Autómatas finitos Paralelos", los ."Autómatas Team", y las Redes de referencias", presentando en su caso mecanismos semiautomáticos de conversión entre los Automatas Cooperativos y ellos.

En la Tesis Doctoral también se define una extensión del modelo original cuyo objetivo es de mejorar su usabilidad y reducir el diagrama de estados de los autómatas sin renunciar a ninguna de las características deseables del modelo original, pero añadiendo, sin embargo, mayor claridad y legibilidad a los modelos resultantes y una manera más natural de enfocar situaciones habituales sin necesidad del uso del modelo completo de los Autómatas Cooperativos dado que en este caso no hay decibilidad sobre determinadas propiedades como la de acotamiento.

Los, así llamados, Autómatas Cooperativos Extendidos (ACE), cuya equivalencia expresiva con el modelo original es completa, permiten simplificar en gran medida y a tal efecto se muestran las simplificaciones resultantes en las conversiones entre sistemas desarrolladas anteriormente, así como un algoritmo de conversión entre el modelo AC original y el extendido.

Una vez realizada la extensión se plantea la necesidad de realizar un análisis de los sistemas modelados identificando partes críticas y subprocesos durante las etapas más temprana del diseño y que este análisis fuera, dentro de lo posible, automático. Para realizar dicho análisis se definen formalmente las relaciones existentes entre las distintas componentes del modelo. Más concretamente las relaciones de vinculación entre autómatas y reglas, las de concurrencia de autómatas en reglas y las de competencia de reglas por autómatas. Dichas relaciones se definen con cuatro niveles de certidumbre siendo la más débil la que puede obtenerse automáticamente en tiempo de diseño. Una vez explicadas las relaciones propone un algoritmo de cálculo que permite identificar tanto los subprocesos, las partes críticas del sistema e incluso los elementos supérfluos o unificables, independientemente del conocimiento heurístico que sobre el sistema a modelar se tenga. Un ejemplo de este cálculo presenta sobre un sistema tipo donde los nombres de reglas, autómatas, acciones y estados han sido codificados para ocultar dicha heurística.

Finalmente se presentan dos herramientas software, la primera es un sistema de ayuda para el modelado con los Autómatas Cooperativos Extendidos, que guia al diseñador en la descripción de las diferentes partes de las que consta el modelo verifica la sintaxis del mismo y parte de la lógica del mismo y a su vez permite realizar el

análisis a priori antes mencionado de forma automática en cualquier momento del diseño, presentando un informe del resultado del mismo. La segunda es un simulador a su vez extendido para simular tanto sistemas AC como ACE y que permite el uso de no determinismo en la elección tanto de reglas como de autómatas.

Resum de la Tesi Doctoral

La Tesi Doctoral "Modelització de la concurrència en sistemes distribuÃ¯ts multiagent: autòmats cooperatius"té com a objectiu final la definició i extensió d'un model per a la descripció de sistemes multiagent que uneix característiques desitjables de les "Xarxes de Petriï la "Teoria d'Autòmats", així com la definició d'un mecanisme automatitzable d'anàlisi sobre els sistemes descrits amb aquest model que permet la simplificació dels mateixos en sub-sistemes i la detecció de parts crítiques abans fins i tot de finalitzar el modelatge. També com a objectiu es persegueix dotar al sistema de programes d'ajuda que automatitzen les parts de disseny, d'anàlisi i de simulació dels sistemes descrits.

Com a aportacions de la Tesi Doctoral es mostra la potència expressiva del model dels Autòmats Cooperatius (AC) a l'hora de descriure sistemes concurrents, a més, partint del model dels AC, es realitza una comparativa dels mateixos amb altres models presents en la literatura, com són els .ªutòmats finits ParalÂ·lels", els .ªutòmats Team", i les "Xarxes de referències", presentant si escau mecanismes semiautomàtics de conversió entre els Autòmats Cooperatius i ells.

En la Tesi Doctoral també es defineix una extensió del model original que el seu objectiu és de millorar la seua usabilitat i reduir el diagrama d'estats dels autòmats sense renunciar a cap de les característiques desitjables del model original, però afegint, no obstant aÃ§ò, major claredat i llegibilitat als models resultants i una manera més natural d'enfocar situacions habituals sense necessitat de l'ús del model complet dels Autòmats Cooperatius atès que en aquest cas no hi ha decibilidad sobre determinades propietats com la d'acotament.

Els, així anomenats, Autòmats Cooperatius Estesos (*ACE), l'equivalència expressiva dels quals amb el model original és completa, permeten simplificar en gran mesura i a aquest efecte es mostren les simplificacions resultants en les conversions entre sistemes desenvolupades anteriorment, així com un algorisme de conversió entre el model AC original i l'estès.

Una vegada realitzada l'extensió es planteja la necessitat de realitzar una anàlisi dels sistemes modelats identificant parts crítiques i sub-processos durant les etapes més primerenca del disseny i que aquesta anàlisi fora, en la mesura del possible, automàtic. Per a realitzar aquesta anàlisi es defineixen formalment les relacions existents entre les diferents components del model. Més concretament les relacions de vinculació entre autòmats i regles, les de concurrència d'autòmats en regles i les de competència de regles per autòmats. Aquestes relacions es defineixen amb quatre nivells de certitud sent la més feble la que pot obtenir-se automàticament en temps de disseny. Una vegada explicades les relacions proposa un algorisme de càlcul que permet identificar tant els *sub-processos, les parts crítiques del sistema i fins i tot els elements innecesaris o unificables, independentment del coneixement heurístic que sobre el sistema a modelar es tinga. Un exemple d'aquest càlcul presenta sobre un sistema tipus on els noms de regles, autòmats, accions i estats han sigut codificats per a ocultar aquesta heurística.

Finalment es presenten dues eines software, la primera és un sistema d'ajuda per al modelatge amb els Autòmats Cooperatius Estesos, que ajuda al dissenyador en la descripció de les diferents parts de les quals consta el model verifica la sintaxi del mateix i part de la lògica del mateix i al seu torn permet realitzar l'anàlisi a priori

abans esmentat de forma automàtica en qualsevol moment del disseny, presentant un informe del resultat del mateix. La segona és un simulador al seu torn estès per a simular tant sistemes AC com ACE i que permet l'ús de no determinisme en l'elecció tant de regles com d'autòmats.

Summary of the Doctoral Thesis

The main objective of Doctoral Thesis "Modelling Concurrency in Distributed Multi-Agent Systems: Cooperating Automataïs the definition and extension of a model conceived to describe multi-agent systems that combines desirable features of the "Petri Nets.ªnd .ªutomata Theory". Another goal is the definition of an automated analysis mechanism over the model in order to simplify those systems in subsystems to detect critical parts even before the end of the modelling phase. The last objective of the Doctoral Thesis is to develop some software tools in order to help designers with the edition, analysis and provide simulation of the systems modelled.

The expressive power of the model of the Cooperating Automata (CA) when describing concurrent systems has been swoen as one of the contributions of the Thesis. The model has been compared with other similar proposals in the literature such as "Parallel Finite Automata", "Team Automata.ªnd Reference Nets". Some convertion algorithms between AC and these proposal has been presented also.

Another contribution of the author is an extension of the original model, the Extended Cooperating Automata (ECA), which aims to improve usability and reduce the state diagram of automatons without sacrificing any of the desirable features of the original model. This extension has been defined adding clarity and readability to the resulting models and a more natural approach to represent common situations without the use of the full model of the Cooperative Automata because, in this case, some properties such as boundedness are undecidable.

The ECA model expressive power is completely equivalent to the original model. However, systems modelled with ECA are quite simple compared to the same systems modelled with CA. The same translations between systems shown before are now simplified. And another conversion two-ways algorithm between CA and ECA has been also described.

The complexity of new systems, and consequently their respective models, has been increasing constantly. In this sense, simplifying but also detecting problems as bottlenecks is crucial at early stages of the design. Doing so over the implementation or implantation phases may duplicate the budget of the whole construction of a given system. Some relations between ECA model parts has been defined in order to perform this authomatic analysis. i.e. Linking relation between an automaton and a transaction rule, Concurrency relation between two automata, and Competition relation between two transaction rules.

These relations are defined with four levels of certainty from the weakest to the strongest. The weakest relations can be obtaied automatically at design time. An algorithm of calculus can be applied to these relations in order to identify different threads, critical parts of the system and even superfluous or unifiable elements. This analysis can be performed regardless of heuristic knowledge about the system. An example of this analysis (applied over a system where the names of rules, controllers, actions and statements have been coded to conceal this heuristic) has been included also.

Finally, two software tools are presented, the first is a support system for modeling with ECA, which guides the designer with the description of the different parts that comprise the model. It includes syntax check and also can perform the a priori analysis automatically during the design stage. This tool presents a report of the analysis

results. The second tool is an extended simulator capable to simulate both AC systems as ACE. The simulator provide non-determinism choice of rules and/or automata during execution.

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Los orígenes

La práctica moderna de la Ingeniería del Software implica un cierto número de fases alguna de las cuales se utiliza para validar el concepto investigado. Tal es la idea y la justificación de los diversos cálculos, modelos y propuestas para la modelización de sistemas. Entre todos los cálculos merecen especial mención aquéllos que permiten modelizar de forma clara y válida los sistemas concurrentes y paralelos y más concretamente los sistemas distribuidos. Dentro de todas las propuestas se sigue, como objetivo primordial, obtener un esquema del sistema que permita tener una idea clara del proyecto en una fase temprana, lo que es particularmente importante sobre todo en cuanto a los requerimientos a desarrollar [136]. Modelizar un sistema antes de construirlo es particularmente útil para sistemas complejos y comporta una serie de beneficios:

- comprender mejor la funcionalidad del sistema,
- identificar posibles riesgos,
- construir una base sobre la que apoyar el resto de fases de diseño y
- la posibilidad de generación automática de código para prototipado.

Los diseñadores necesitan disponer de técnicas simples pero potentes de modelado que les permitan experimentar con ideas en las etapas más tempranas del desarrollo del producto. En particular, en la ingeniería del software el conocimiento sobre aplicaciones distribuidas complejas es un ejercicio no trivial, incluso pese a la aparición de tecnologías "middleware" como OMG, CORBA, y el Jini de Sun [132]. Por ejemplo, los Sistemas de Trabajo Cooperativo Soportado por Computador (de sus siglas en inglés CSCW) [31] son habitualmente distribuidos y pensados para proporcionar herramientas que permitan a los usuarios, geográficamente separados, cooperar en táreas comunes de una forma concurrente. Dichos sistemas a menudo muestran un complejo comportamiento dinámico interactivo.

Dentro de la multitud de cálculos para la modelización de la concurrencia que se han ido desarrollando a lo largo de los años, desde los setenta, pueden distinguirse propuestas que difieren en el formalismo utilizado, (π-cálculo[141], extensiones

del λ-cálculo[98, 113], Øgeblik/Obliq [79], etc.). En concreto, como trabajo previo al desarrollo de esta tesis doctoral hemos presentado propuestas sucesivas [57, 58, 59] que culminaron en la definición de un modelo basado en el λ-cálculo, el λ-cálculo Multietiquetado Paralelo [60, 61]. Hasta el momento, todos ellos en distintos grados han demostrado expresividad suficiente para modelizar tanto sistemas concurrentes como orientados a objetos, distribuidos, etc. Sin embargo, la mayoría de estos modelos carecían de la componente gráfica que permite una representación visual del comportamiento de los sistemas modelizados, cosa que, si bien no resulta imprescindible, ha dificultado el entendimiento de los sistemas más complejos.

Las tecnologías orientadas al agente son extensiones naturales de aproximaciones basadas en componentes y han tenido el potencial de influir en gran medida en todos los trabajos posteriores orientados a la programación concurrente. Este área se ha convertido en una de las más importantes y activas en la ciencia de la computación. Algunos campos de aplicación donde las tecnologías orientadas al agente juegan y jugarán un rol crucial son: *Inteligencia Ambiental* [144, 145], que representa el reparto sin fisuras de la computación distribuida, las comunicaciones contínuas y los interfaces de usuario inteligentes tanto a nivel industrial como de consumo y que ha sido ampliamente tratada en la literatura como por ejemplo en [47]; *Computación en Red* como por ejemplo en [84], donde aproximaciones basadas en sistemas *multiagente* deben favorecer el uso eficiente de recursos en programación de alto rendimiento como infraestructura para aplicaciones o programas científicos, de ingeniería, medicina y aplicaciones comerciales; *Negocios Electrónicos*, donde los modelos basados en agentes muestran la automatización y semiautomatización de la reunión de información y transacciones por internet; *Web Semántica* [27] y Web 3.0 [37] donde los agentes necesitan al tiempo proveer servicios y hacer el mejor uso de los recursos disponibles; *Bioinformática y Biología Computacional* [108], donde los agentes inteligentes deben mostrar la explotación coherente de la revolución de datos ocurrida en biología; y otros como *Monitorización* y *Control* [83], *Gestión de Recursos* [41] y por ejemplo, aplicaciones *Militares* [112], *Espaciales* [76] y de *Manufactura* [95].

Otro reto fundamental a la hora de modelizar un sistema (sobre todo en fases tempranas del diseño) consiste en conseguir que el modelo sea lo suficientemente expresivo pero a la vez no demasiado extenso, para poder representar en su totalidad las funcionalidades descritas de una forma sencilla e inteligible. Si además se dispusiese de un mecanismo formal y, en lo posible, automático para poder simplificar el sistema completo en partes independientes e identificar aquellos componentes cruciales que puedan desembocar en cuellos de botella o situaciones críticas, se dispondría de una herramienta que permitiría un importante ahorro en costes de desarrollo al partir de un sistema simplificado desde el principio y no en un momento del desarrollo que hiciese que el coste se incrementase sustancialmente.

A continuacion se presenta el Capítulo 1, una introducción sobre los sistemas multiagente. En el Capítulo 2 se realiza un breve recordatorio sobre la Teoría de Autómatas [78] y las redes de Petri [115], bases teóricas sobre las cuales se sustenta el modelo de los Autómatas Cooperativos Extendidos [63, 64, 65], motivo de la presente tesis doctoral. Para ilustrar la aplicación de dichas bases teóricas en diferentes propuestas de modelos, en la Sección 2.4 del Capítulo 2 se presentan varios de dichos modelos para la representación de sistemas distribuidos y paralelos basados en autómatas como son los Autómatas Finitos Paralelos [131], los Autómatas Team [39]

y las Redes de Referencias [92, 103, 94] algunos de los cuales, más adelante, tras la definición en el Capítulo 3 del modelo de los Autómatas Cooperativos Extendidos, son comparados con este.

En el Capítulo 4 se realiza un estudio a priori sobre los sistemas modelizados con los Autómatas Cooperativos Extendidos identificando y categorizando las relaciones entre autómatas y reglas, autómatas y autómatas y reglas y reglas, así como implementando un algoritmo que permite, de forma automática simplificar el sistema e identificar posibles situaciones o agentes críticos en tempranas fases de diseño.

1.2. Agentes

1.2.1. Teoría de agentes

Un agente puede definirse como una entidad computacional autónoma y que resuelve un problema, y que además es capaz de realizar acciones flexibles en un entorno abierto, dinámico e impredecible. Los agentes a menudo se desarrollan en entornos en los que interactúan y cooperan con otros agentes (incluyendo como agentes a usuarios y a software) y posiblemente tengan intereses en conflicto.

Los agentes se distinguen de los objetos (en el sentido de la programación orientada a objetos) en que son entidades autónomas capaces de realizar elecciones en cuanto a sus acciones e interacciones. Los agentes no pueden, sin embargo, ser invocados directamente como objetos. No obstante, se pueden construir usando las tecnologías orientadas a objetos.

El impacto de las tecnologías de agentes en sus distintos campos de aplicación puede suceder de distintas maneras: primero, como una metáfora para el diseño de sistemas distribuidos complejos; segundo, como fuente de tecnologías para estos sistemas de computación y tercero como modelos complejos de sistemas del mundo real. Es basándonos principalmente en el primer aspecto y en menor medida en el tercero en el que se define el modelo de los *Autómatas Cooperativos*, teniendo en cuenta que el panorama de la computación se ha ido trasladando en el tiempo desde la visión de un sistema como un computador individual hacia la distribución de táreas y la cooperación entre distintas entidades a través de redes de computadores y se considera un sistema como un todo abierto, dinámico y, por supuesto, distribuido.

La evolución de la tecnología y de la cantidad de información muestra la necesidad de que los modelos y paradigmas tradicionales se adapten y potencien para que sean capaces de representar las características de los nuevos entornos abiertos y dinámicos en los que, por ejemplo, puedan interactuar sistemas heterogéneos traspasando fronteras organizacionales y operando de manera efectiva con circunstancias cambiantes y con grandes cantidades de información. En particular, es preciso que los modelos representen de alguna manera un cierto grado de autonomía para permitir a los componentes responder dinámicamente a los cambios en las circunstancias en las que se encuentre para continuar con el proceso que lleve a la consecución de los objetivos del sistema pese a dichos cambios. En desarrollos prácticos los servicios WEB, por ejemplo, ofrecen nuevos caminos para realizar negocios mediante herramientas estandarizadas y admiten una visión orientada a los servicios de distintas (e independientes) componentes software que interactúan entre ellas para dar una valiosa funcionalidad en conjunto. En el contexto de estos desarrollos las tecnologías de agentes se con-

vierten en las principales herramientas para solucionar los problemas emergentes y manipular la creciente complejidad de los sistemas a representar y desarrollar.

Agentes como diseño

El motivo por el cual este campo del diseño con agentes ha sido desarrollado inicialmente ha sido su potenial uso como una herramienta de abstracción para el diseño y la construcción de sistemas complejos. Los agentes dotan a los diseñadores, desarrolladores e ingenieros de un medio de estructurar aplicaciones en torno a elementos autónomos y capaces de comunicarse entre sí y les facultan para la construcción de herramientas software con las que hacer efectiva la metáfora de diseño [81]. En este sentido ofrecen un camino nuevo y más apropiado para el desarrollo de sistemas complejos, especialmente en entornos abiertos y dinámicos. Se deben, por tanto, desarrollar nuevas técnicas y metodologías así como nuevas herramientas para poder realizar este punto de vista en el desarrollo de sistemas. Por ejemplo, se requieren metodologías para análisis y diseño guiado, para el diseño de componentes individuales, para la infraestructura soporte, etc.

Por otro lado, otra de las ventajas adicionales es que los agentes ofrecen un medio adecuado para considerar sistemas complejos disgregándolos en el estudio del comportamiento de múltiples componentes distintas e independientes. Además, los agentes deben mantener las diferentes funcionalidades que los distingan (como eran la capacidad de coordinación, aprendizaje, etc.) y situarlos embebidos en el entorno adecuado para que interactúen. Estas nociones a la hora de diseñar un sistema dan como resultado la aparición de una serie de áreas de tecnologías que relacionan directamente estas abstracciones en el diseño y desarrollo de grandes sistemas, de agentes individuales, de medios de interacción entre agentes y en la consideración de funcionamiento de capas de alto nivel como organizaciones y su funcionamiento computacional. Los esfuerzos más recientes incluyen diferentes áreas como la ingeniería del software orientada a objetos, las arquitecturas de agentes, los sistemas de agentes móviles, las infraestructuras de agentes y las instituciones electrónicas.

Agentes como fuente de nuevas tecnologías

Las aproximaciones basadas en agentes tienen como fuente de tecnologías un gran número de áreas de interés, tanto aplicadas como teóricas. Incluidas en estas tenemos el "planning" distribuido y mecanismos de toma de decisión, mecanismos de auditoría automática, lenguajes de comunicación, mecanismos de coordinación, algoritmos y arquitectura de matching, ontologías y agentes de información, negociación y mecanismos de aprendizaje.

Teniendo en cuenta que las tecnologías de agentes incluyen las técnicas específicas y los algorítmos de interacción con otras de programación dinámica y entornos abiertos está claro que debemos traspasar las fronteras de uno y otro campo de estudio para unificarlos en un nuevo paradigma. Las tecnologías de agentes deben incluir conceptos como la reacción equilibrada y la deliberación de agentes individuales, el aprendizaje de y sobre otros agentes en el entorno y las preferencias del usuario, la búsqueda de caminos de interacción, negociación y cooperación entre agentes y el desarrollo de conceptos adecuados de formación y manipulación de coaliciones. Más

aún, la adopción de las aproximaciones basadas en agentes ha influido de manera incremental en otros campos. Por ejemplo, frente a las aproximaciones centralizadas los sistemas multiagentes proporcionan métodos más rápidos y efectivos de localización de recursos en entornos complejos como la manipulación de utilidades en redes. De manera similar, el uso de estas disciplinas en la simulación de sistemas del mundo real puede proporcionar respuestas a complejos problemas físicos o sociales que no serían obtenibles de otra forma, como pueda ser el caso de modelizar el impacto de los cambios climáticos sobre la vida de poblaciones biológicas o de el impacto de una política determinada en la economía o sociedad.

1.3. Estado del arte en los sistemas de agentes

1.3.1. Ingeniería del software orientada a agentes

El trabajo sobre metodologías e ingeniería del software para sistemas de agentes es fruto de la sinergia por la interacción de las comunidades existentes de investigación sobre este campo y su énfasis en la aplicación práctica de sus resultados en la industria. El objetivo fundamental es determinar cómo afectan las cualidades de los agentes a la ingeniería del software y qué herramientas adicionales y qué conceptos son necesarios para aplicar los sistemas de agentes a estos procesos de ingeniería y a sus estructuras.

Algunas áreas de interés bajo estudio son:

- Ingeniería de requerimientos para sistemas de agentes [110, 23];
- Técnicas de especificación de diseños(conceptuales) de sistemas de agentes [97];
- Técnicas de verificación [46];
- Diseño y análisis orientado a agentes [147];
- Ontologías específicas para requerimientos de agentes, modelos de agentes y modelos de organización [117];
- Librerías de modelos genéricos de agentes y componentes [102];
- Patrones de diseño de agentes [127];
- Técnicas de validación y prueba [80];
- Herramientas de apoyo para el proceso de desarrollo de sistemas de agentes (como plataformas de agentes) [90].

El uso de la ingeniería del software orientada al agente para el desarrollo de sistemas distribuidos complejos se opone a la aplicación tradicional de la ingeniería del software con agentes. Este sutil cambio de orientación, se basa en adoptar la metáfora de agente o de punto de vista del diseño en el que los agentes en si mismos tienen una manera elegante y natural de coordinarse para manejar situaciones y sistemas muy complejos. La complejidad de muchos sistemas viene de las interacciones entre los componentes del propio sistema y el paradigma de los agentes es una manera natural de expresar dichas interacciones. La abstracción de agentes puede aplicarse no ya para representar los componentes tecnológicos de los sistemas ya implementados, sino

también para modelar y diseñar sistemas complejos que pueden entonces ser implementados partiendo de otro punto de vista y con transacciones más claras y sencillas. Por supuesto, dada la inicial complejidad que motiva el uso de una aproximación basada en agentes, parece claro que el ciclo de vida completo del desarrollo de estos sistemas se verá afectado beneficiosamente (en aras de la claridad y sencillez) por el uso desde el principio de los sistemas de agentes en lugar de su incorporación en sólo alguna fase del desarrollo.

1.3.2. Arquitecturas de agentes

Las arquitecturas de agentes son las máquinas fundamentales que encontramos bajo las componentes autónomas y que soportan efectivamente el control del comportamiento en entornos dinámicos y abiertos del mundo real.

Los esfuerzos iniciales en el campo de la computación basada en agentes se focalizaron en el desarrollo de arquitecturas de agentes inteligentes y en los posteriores se han ido estableciendo varios estilos de arquitecturas. Estos estilos van desde los agentes puramente reactivos que operan de una manera simple estímulo-respuesta como aquellos basados en la arquitectura de subsumciones [22] y, en el otro extremo aquellos más "deliberativos" que razonan sus acciones, como por ejemplo la clase de los agentes creencia-deseo-intención (Belief-Desire-Intention agents BDI) [42, 121] cuya prevalencia fué incrementandose día a día, (incluyendo productos comerciales como JACK [146], de Agent Oriented Sowtware). Entre ambos extremos, podemos encontrar combinaciones híbridas o arquitecturas estratificadas que intentan incluir las reacciones y deliberaciones en un esfuerzo de adoptar lo mejor de cada aproximación. Además, ya hay en desarrollo aplicaciones más sofisticadas que los BDI, pero los beneficios de esta mayor sofisticación están muy limitados a áreas muy definidas de interés más que a soluciones generales.

1.3.3. Infraestructuras para agentes

A la hora de implementar sistemas basados en agentes hemos de hablar de infraestructuras, destinadas al uso operacional y de desarrollo de estos sistemas. La investigación en los sistemas de agentes Móviles [21] ha proporcionado importantes contribuciones en términos de eficiencia en el código de los mecanismos de movilidad y como fuente de descubrimientos.

El *Middleware* pueden ser descrito como la capa de software que está entre el servidor y el sistema operativo de red por un lado y la capa de aplicaciones por otro lado. Su propósito es proveer de un sistema común de interfaces de programación que los desarrolladores puedan usar para crear sistemas distribuidos.

Sin embargo, aunque los agentes inteligentes han sido estudiados durante años su implantación está todavía dando sus primeros pasos. Para conseguirla la comunidad científica ha desarrollado una gran variedad de plataformas de agentes en las últimas dos décadas, algunas de propósito general y otras orientadas a su uso en un dominio específico. Algunas de ellas fueron ya abandonadas, mientras que otras continuan ofreciendo nuevas versiones. Al mismo tiempo, la comunidad de investigación sobre sistemas orientados al agente sigue produciendo nuevas plataformas que difieren sobre todo al público al que van dirigidos así como al área de trabajo sobre el que se orientan.

Hay muchas disponibles tanto a nivel comercial como académico y a continuación se presenta una selección de las mismas.

- *Agent Factory Framework* [126]. Es una colección *open source* de herramientas, plataforma y leguanjes de programación para el desarrollo e implantación de sistemas multiagente. Se usa en multiples proyectos incluyendo robótica, *mobile computing* y otros, como por ejemplo en el *MiRA - Mixed Reality Agents project* [77].

- *AgentBuilder* [2]. Es una plataforma de agentes tradicional usada en multitud de casos de simulación. Permite a los desarrolladores sin experiencia en el desarrollo de sistemas inteligentes ni tecnologías basadas en agentes construir rápida y fácilmente aplicaciones inteligentes de agentes. Es una herramienta que utiliza el lenguaje *KQML* [44].

- *AgentScape* [111]. Diseñada para facilitar el diseño y desarrollo de sistemas distribuidos multiagente a gran escala, heterogeneos y seguros. *AgentScape* está enfocada a la autonomía y escalabilidad.

- *AGLOBE* [129]. Es una plataforma de diseño de agentes para testeo experimental. Sólo permite el testeo de sistemas cerrados sin comunicación con sistemas externos. Este sistema está financiado por la USAF (*U.S. Air Force*) para simulaciones de mundo real.

- *AnyLogic* [18]. Es una plataforma de simulación multimétodo que permite simulaciones de agentes de propósito general y de dinámica de sistemas.

- *Cormas* [96]. Es una plataforma de simulación basada en el desarrollo de aplicaciones dentro del lenguaje orientado a objetos Smalltalk. Es muy popular y tiene interesantes características para las aplicaciones de mundo real.

- *Cougaar* [54]. Esta propuesta sigue una arquitectura de agentes cognitivos y está realizada por DARPA (*Defense Advanced Research Projects Agency*). Se trata de una plataforma de código abierto que ofrece soporte a problemas logísticos.

- *CybelePro* [32]. CybelPro ofrece a sus usuarios una infraestructura robusta y de altas prestaciones para el desarrollo de sistemas a gran escala y con alto rendimiento.

- *EMERALD* [89]. Se trata de una implementación reciente para razonamiento interoperable entre agentes en la Web Semántica usando servicios de razonamiento seguro de terceras partes. Construido sobre JADE (comentado más adelante) cumple con el estándar de la FIPA (*Foundation for Intelligent Physical Agents*).

- *GAMA* [52]. Es una plataforma de simulación cuyo objetivo es proveer de un entorno de desarollo completo para construir simulaciones multiagentes con espacialidad explícita. Incluye visualización en 3D, manipulación de datos de sistemas de información Geográfica (GIS de sus siglas en inglés) y modelado multinivel.

- *INGENIAS Development Kit* [50]. Es una herramienta de desarrollo de sistemas multiagente que soporta la metodología INGENIAS. Incluye un metaeditor (INGENME) para producir editores visuales para lenguajes que se definen usando archivos XML.

- *JACK* [146]. Es una veterana plataforma comercial antes mencionada para la construcción, desarrollo, puesta en marcha y ejecución de sistemas multiagente. Está basado en la lógica BDI [42, 121]. Escrito completamente en Java [51] JACK es portable y es capaz de ejecutarse desde en smartphones hasta servidores con multiprocesadores. Sin embargo, aún no cumple con las especificaciones de la FIPA.

- *JADE* [14]. Se trata de una plataforma imlementada completamente en Java y dado que funciona con la versión 1.2 puede ser utilizado en dispositivos con recursos limitados como smartphones. Está orientado a la industria y es la plataforma más popular en la comunidad industrial y académica que cumpe el estándar de la FIPA.

- *Jadex BDI Agent System* [20]. Esta plataforma sigue el modelo BDI y facilita una construcción sencilla de agentes inteligentes basados en ingeniería del software. Permite la programación de software para agentes inteligentes en XML y Java. Está dirigido por el Grupo de Sistemas Distribuidos y Sistemas de Información de la Universidad de Hamburgo.

- *JAMES II (Java Framework for Modeling & Simulation)* [75]. JAMES II es un entorno de trabajo basado puramente en Java sin dependencias externas para modelado y simulación. Está basado en una arquitectura *plug-in* llamada coloquialmente *plug'n simulate*.

- *JAS, the Java Agent-Based Simulation Library* [130]. Si bien es cierto que JAS no es una plataforma de agentes pura, en la práctica actua como tal. JAS dispone de una serie de herramientas específicamente diseñadas para el modelado de simulaciones basadas en agentes. Se trata de un clon de la librería Swarm del instituto de Santa Fe.

- *Jason* [17]. Es un intérprete de una versión extendida de *AgentSpeak*, un lenguaje de programación orientado a agentes BDI implementado en Java.

- *JIAC, the Java-based Intelligent Agent Componentware* [74]. Es una arquitectura y plataforma basada en Java que facilita el desarrollo y la operación de aplicaciones y servicios distribuidos a gran escala.

- *MaDKit, the Multiagent Development Kit* [53]. Se trata de una plataforma de código abierto modular y escalable escrita en Java y construida bajo el modelo de organización AGR (*Agent/Group/Role*).

- *NetLogo* [135]. Es un entorno de modelado programable multiagente. Está diseñado con el mismo espíritu que el lenguaje de programación Logo [45]. NetLogo is muy popular en la comunidad educativa y de investigación.

- *MASON* [100]. Es un conjunto de herramientas rápidas, de eventos discretos y simulación multiagente en java que funciona como plataforma de simulación de agentes. MASON traza una relación entre el modelo y la visualización, lo que permite que los modelos puedan separarse dinámicamente de la visualización mediante puntos de control.
- *The Repast Suite* [106]. Esta es una familia de plataformas de simulación y modelado de código abierto que han sido desarrolladas continua y colectivamente durante más de 14 años. RePast fue desarrollado inicialmente en el *University of Chicago's Social Science Research Computing Lab.*
- *SeSAm (Shell for Simulated Agent Systems)* [88]. Es un entorno genérico para el desarrollo y simulación de modelos multiagentes. Su objetivo principal es permitir a los científicos construir modelos con programación visual, dado que el paradigma de agentes es muy intuitivo, especialente modelizando sociedades. Fue desarrollado inicialmente en Lisp pero desde el año 2000 fue rediseñado e implementado en Java.
- *Swarm* (Swarmfest 2012). Se trata de la primera herramienta software reutilizable creada para la modelización y simulación basadas en agentes, y fue desarrollado en el *Santa Fe Institute* en 1994. Fue diseñado específicamente para estudios de complejidad y aplicaciones de vida artificial.

Con el pasar de los años han surgido muchas tecnologías y arquitecturas orientadas ad hoc para la conexión de redes, lo que es a su vez el centro de soporte de los sistemas basados en agentes. Podemos incluir en ellas, por ejemplo, a Jini [132] inicialmente desarrollado por SUN Microsystems y en la actualidad de código abierto y denominado *Apache River*, UPnP (*Universal Plug and Play*)[137]. De manera similar tenemos lenguages como XML (Extensible Markup Language) [19] y RDF (Resource Description Format) [119] con ontologías estándarizadass y que proporcionan un significado a nivel semántico de la descripción de recursos y la manipulación de este tipo de datos.

La infraestructura de agentes también sirven para manipular estos sistemas a través de mecanismos como el *leasing* Jini, que permite el control de acceso a servicios de registro, soporte de comunicaciones para mecanismos subyacentes de transporte e intercambio de información robustos y soporte de seguridad, asegurando que los agentes están adecuadamente autentificados y autorizados a realizar las acciones que requieran.

Muchas de estas táreas se acumulan por el problema de influencias que existe trabajando en aplicaciones de computación basada en agentes. Uno de los más claros ejemplos (y más habituales) aparece en los servicios Web. Hay un gran número de entornos de desarrollo de agentes y herramientas (incluyendo estándares de la FIPA (Foundation for Intelligent Physical Agents). Los esfuerzos por estandarizar la gran oferta de sistemas para hacer del desarrollo de aplicaciones orientadas al agente una tárea escalable e interoperable entre los distintos modelos son las iniciativas y objetos de estudio más importantes para aportar madurez en este sentido. Podemos incluir entre estos estándares (aunque no los limitemos a ellos) la siguiente clasificación:

- Tecnologías Base como por ejemplo XML(eXtensible Markcup Language) y RDF(Resource Description Format)

- Comercio Electrónico como ebXML (Electronic Bussines using XML) [49] y el consorcio RosettaNet [143] (para desarrollo de estándares de comercio electronico abierto)

- Plug & Play universal como la tecnología de red Jini (Arquitectura de red para la construcción de sistemas distribuidos) y los sistemas Plug & Play

- Servicios Web (como la iniciativa industrial UDDI (*Universal Description, Discovery, and Integration*), SOAP(*Simple Object Access Protocol*) y WSDL/WSCL (*Web Services Description Language y Web Services Conversation Language*) todos ellos descritos en [142]

1.4. Aplicaciones y Desarrollo

Las aplicaciones potenciales de los sistemas basados en agentes pueden dividirse en tres categorías principales:

- Agentes Asistentes. Son agentes encargados de reunir información o realizar transacciones en nombre de sus clientes humanos en Internet. Un ejemplo de esto son las aplicaciones de agentes de búsqueda de hoteles por internet.

- Los sistemas de decisión de Multiagentes. Los agentes que participan en el sistema se coordinan y toman juntos algunas decisiones. Por ejemplo, un sistema de agentes que represente a varios componentes de una red de telecomunicaciones puede tratar de localizar conjuntamente recursos en la red y manipular las operaciones de la red. Los sistemas de decisión conjunta usados por los agentes involucrados deben ser mecanismos económicos, como los tipo subasta o aquellos basados en argumentación.

- Sistemas de simulación de Multiagentes. En los que un sistemas multi-agente se usa como modelo para simular algún otro en el dominio del mundo real. Típicamente estos modelos se usan para representar diferentes componentes interactuando de manera diversa y compleja y donde las propiedades del sistema por niveles no interfieren con las propiedades de los componentes. Ejemplos de este tipo de dominios pueden ser: economías humanas, sociedades humanas y animales, poblaciones biológicas, sistemas de tráfico por carretera, redes de computadores y juegos (como las criaturas basadas en agentes), etc.

La primera categoría se basa en agentes simples, mientras que las otras dos se basan en grupos de agentes; sin embargo, los agentes simples de la primera categoría deben interactuar con otros agentes presentes en la red para realizar sus táreas. Las decisiones en las otras dos táreas se realizan conjuntamente y no de manera individual. La principal distinción entre la segunda categoría y la tercera es el objetivo para el que se utilizan. Mientras que la segunda tiene como objetivo el propio sistema (incluyendo a los agentes) la tercera tiene como objetivo entender el sistema.

Esta última categoría es el marco en el que definimos en posteriores capítulos un nuevo modelo de sistemas multi-agente y por ello la veremos con más detalle.

1.4.1. Aplicaciones de Simulación

Los sistemas Multiagentes ofrecen modelos robustos para representar los entornos de mundo real con un grado adecuado de complejidad y dinamismo. Por ejemplo, la simulación de economías, sociedades, entornos medioambientales, etc. son áreas típicas susceptibles de simulación mediante estos sistemas.

La simulación basada en agentes se caracteriza por la intersección de tres campos científicos fundamentales, llamados computación basada en agentes, ciencias sociales y simulación por computación. Las ciencias sociales estudian interacciones entre entidades sociales e incluyen psicología social, dirección, política y biología. La simulación por computador concierne a técnicas de simulación de fenómenos por computador como eventos discretos, orientación a objetos, simulación por ecuaciones, etc.

Hay dos aproximaciones básicas para la simulación social con agentes. La primera está basada en la definición lógica de sistemas en los que subyace la interacción social (Modelo Fundacional) y la segunda es la observación y modelización de procesos sociales (Modelo Representacional) [99]. La primera está influida por las ciencias sociales y la segunda por la simulación social y ambas pueden ser usadas de manera conjunta.

Los científicos encuentran la simulación por computador útil cuando representa cambios que no son facilmente previsibles, pero sus causas son típicamente reconocibles de manera retrospectiva. Los simuladores de vuelo usados por los pilotos tienen una aproximación similar. El análisis del escenario para estrategias comerciales y políticas sociales tiene el mismo propósito. Un análisis basado en una simulación social por agentes puede ser más flexible y realista que otros métodos de modelado. Por ejemplo, una simulación y análisis del cambio climático (siguiendo el protocolo de Kyoto) puede capturar el desarrollo de las presiones sociales tanto como las elecciones individuales y la interacción social. La información de las reacciones humanas en circunstancias extremas pueden hacer más y más robustos los agentes.

Tenemos tres áreas de aplicación principales en simulación social:

- Estructuras sociales e Instituciones, donde la observación y la evidencia son usados para diseñar el modelo. Algunas veces estas simulaciones son útiles para desarrollar explicaciones plausibles a fenómenos observados. En otras ocasiones sirven para ayudar a diseñar las estructuras organizativas o para informar acerca de decisiones de mantenimiento o política interna. Por ejemplo un estudio de mercado para la selección de características de un producto por una compañía (enlazado con el diseño del propio producto) puede estar basado en un modelo de simulación por agentes del área de mercado en el que se va a vender estos productos, representando los agentes, por ejemplo, a los consumidores y eligiendo productos alternativos en base a precios, reputación de la cadena, garantías, información a través de otros agentes, etc.

- Sistemas Físicos. Ejemplos de estos sistemas pueden encontrarse en modelos de edificios inteligentes basados en agentes, sistemas de tráfico, poblaciones biológicas. Por ejemplo, se usan modelos de sistemas multiagentes para simular el impacto del cambio climático sobre poblaciones de fauna y flora en una zona concreta.

- Sistemas Software de todo tipo, habitualmente incluyendo comercio electrónico,

agencias de información, tráfico en redes de telecomunicaciones. Este tipo de sistemas son probados previamente a su implantación en modelos de sistemas multi-Agente para comprobar su comportamiento y su respuesta a priori.

Como resulta evidente, las distintas simulaciones pueden ser aplicadas a los más variopintos ámbitos y a un gran rango de diferentes contextos, desde procesos de manufactura hasta sistemas complejos de información pasando por ecosistemas, sistemas de seguridad, control, etc. e incluso simulando los más abstractos ítems, como ingeniería social, creencias, convicciones, derechos, deberes, etc.

1.5. Nuestro trabajo: los Autómatas Cooperativos

Vistos los aspectos fundamentales de la teoría de agentes y en concreto sobre el estado del arte en cuanto a simulación, pretendemos enfocar este trabajo en el diseño de un nuevo modelo de agentes. Las características del cual serán explicadas en sucesivos capítulos.

Nuestra idea es centrarnos en la modelización de la concurrencia de agentes y hacer un diseño "a alto nivel" entendiendo por esto un prototipo y no pretendiendo realizar una herramienta comercial de modelización, sino estudiar las relaciones entre las diferentes componentes del sistema en aras de analizar los mismos en las fases más tempranas de diseño.

Para aprovechar al máximo las ventajas y facilidades que impone la utilización de este tipo de modelos orientados a los agentes, hemos de proveer a nuestro modelo de una base teórica adecuada que permita asentar los fundamentos de corrección y de fiabilidad necesarios y de una facilidad de manejo para su utilización en la industria. Por ello, entre los formalismos utilizados, destacaríamos a aquellos que tienen una adecuada componente gráfica y además se basan en una teoría bien fundada y de reconocida validez. Si además de todo ello, cuentan con herramientas automáticas de verificación o son susceptibles de ser verificadas o si, en el caso de alcanzar excesiva expresividad, son susceptibles de ser simuladas por computador para mostrar la ejecución de un sistema como paso previo a su concreción en un modelo software real, tendremos una herramienta de verdad eficaz para servir como ayuda al diseño de software.

Ejemplo de este tipo de formalismos son los autómatas Team [39], una extensión de los autómatas Input/Output[101] diseñados para la modelización de sistemas *groupware* y CSCW(Computer Supported Cooperative Work) y basados en la idea de que los autómatas se agrupen en grupos y subgrupos y que el disparo de sus reglas de transición venga dado por la sincronización asociada a los grupos de autómatas ordenados según sus roles de maestro/esclavo. Más tarde se verá con en detalle.

Otro ejemplo de esta misma idea es el formalismo de los Autómatas Finitos Paralelos [131], que se basa en la teoría de autómatas [78] y en la de las redes de Petri[109], y que trata de expresar, mediante la construcción de una nueva clase de autómatas finitos con una nueva notación, y con la idea de la ejecución según el marcado de una red de Petri, pero con una normalización del número de tokens a la cantidad máxima de 1, que permite expresar la idea de concurrencia, paralelismo, sincronización y con ello permite modelizar sistemas distribuidos, concurrentes y paralelos.

Con la idea de aunar estas características deseables de la Teoría de Autómatas y de las redes de Petri, permitiendo además expresar la teoría de agentes asociada de manera que se representen y definan individualmente cada una de las componentes de un sistema complejo así como las interacciones entre ellas aparecen los Autómatas Cooperativos[5], una propuesta de 1999 de Badouel, Darondeau, Quichaud y Tokmakoff, cuya extensión y simulación es objeto de esta Tesis Doctoral. Esta propuesta se basa en la teoría de autómatas, dado que se representa un sistema como un compendio de agentes (que no son más que autómatas agente de comportamiento no determinista) junto con una porción de memoria privada que indica la pertenencia de un agente a una tárea. Cada uno de estos agentes regula el comportamiento individual de una componente independiente del sistema. Todos estos autómatas evolucionan en base a reglas de transición que, a la manera de las redes de Petri, muestran la sincronización de los distintos autómatas, la cooperación y la competencia por el uso o la comunicación con recursos. Esta sincronía se produce de manera similar a la evaluación de los distintos marcados de una red de Petri, evaluando en las reglas por un lado el estado de cada autómata involucrado y por otro las condiciones que los atributos de cada autómata cumplen, a modo de guardas de disparo y el disparo de la regla produce un nuevo marcado representado como una actualización de los valores de esos atributos.

Además, en la extensión que se propone en el Capítulo 3 se presenta como parte del estado un nuevo tipo de atributos, los atributos numéricos que difieren de los clásicos en varios términos. En principio, los atributos numéricos serán parte de la definición del estado del autómata. Como se trata de autómatas finitos, el número de estados debe ser a su vez finito. Este nuevo tipo de atributos tendrá, por tanto, un rango limitado y será de tipo Entero. En segundo lugar, los atributos clásicos son entendidos en el modelo de los Autómatas Cooperativos como identificadores de táreas, pudiendo los agentes incorporarse a estas táreas de forma dinámica. Como tales identificadores de táreas, el rango de estos atributos es virtualmente infinito y, por tanto, no forman parte del estado del autómata, sino que delimitan las posibles interacciones con otros autómatas.

Tan cercano al formalismo de las redes de Petri es el sistema resultante, que una extensión de las mismas es presentada junto a nuestra propuesta, las Redes de Referencias [92, 93, 94] cuya definición permite una traducción casi literal y automática de un sistema modelizado con Autómatas Cooperativos Extendidos trasladando los autómatas a redes de Petri y las reglas de transacción a redes de redes. Un ejemplo de dicha traducción está también mostrado en el Capítulo 3.

Una vez modelizado un sistema bajo la perspectiva de los Autómatas Cooperativos, el modelo está pensado para permitir realizar de manera automática una serie de análisis destinados a la detección de posibles elementos críticos como cuellos de botella y a la identificación de subprocesos independientes y cuya ejecución pueda separarse sin afectar al resultado de la ejecución del sistema en su conjunto logrando con ello su simplificación y la corrección previa de situaciones que, de detectarse en pleno proceso de implementación o, peor aún, de implantación, darían lugar a sobrecostes en tiempo y esfuerzo en el proyecto completo. Dicho análisis es presentado en el Capítulo 4 e integrado en la herramienta desarrollada y explicada en el Capítulo 7.

Finalmente, los modelos pueden simularse en una herramienta realizada al efecto que permite tanto la ejecución automática como paso a paso, mostrando la problemática de la sincronización y la selección determinista o no determinista de reglas

y autómatas según el estado del sistema. Ejemplos del funcionamiento de dicha herramienta se muestra en el Capítulo 6.

1.5.1. Resumen de objetivos y aportaciones de la Tesis Doctoral

Los objetivos principales de la Tesis doctoral son:

- La definición y extensión de un modelo para la descripción de sistemas multiagente que aúne características deseables de las redes de Petri y la Teoría de autómatas,
- La comparación de ambos modelos con otros similares presentes en la literatura.
- La definición de un mecanismo automatizable de análisis sobre los sistemas descritos bajo este modelo que permita la simplificación de los mismos en subsistemas y la detección de partes críticas antes incluso de finalizar el modelado.
- Dotar al diseñador, de un software de ayuda que automatice las partes de diseño, de análisis y de simulación de los sistemas descritos.

Como aportaciones de la Tesis Doctoral cabe destacar las siguientes:

- Se muestra la potencia expresiva del modelo de los autómatas cooperativos a la hora de describir sistemas concurrentes.
- Se realiza una comparativa del modelo AC con otros modelos presentes en la literatura como son: los Autómatas Finitos Paralelos, los Autómatas Team y las Redes de Referencias. Además se presentan, en su caso, mecanismos semiautomáticos de conversión entre modelos.
- Se realiza una extensión del modelo original en aras de mejorar su usabilidad y reducir el diagrama de estados de los autómatas sin renunciar a ninguna de sus características deseables pero añadiendo, sin embargo, mayor claridad y legibilidad a los modelos resultantes y dotándole de una manera más natural de enfocar situaciones habituales sin necesidad del uso del modelo completo de los Autómatas Cooperativos sobre el que no hay decibilidad sobre determinadas propiedades como la de acotamiento.
- Aprovechando que se ha realizado la extensión y que su equivalencia expresiva con el modelo original es completa se muestran las simplificaciones resultantes en las conversiones antes descritas, así como un algoritmo de conversión entre el modelo AC original y el extendido.
- Se plantea la necesidad de realizar un análisis en las etapas más tempranas del diseño y que este análisis fuera, dentro de lo posible, automático. Para realizar dicho análisis se definen formalmente las relaciones existentes entre las distintas componentes del modelo y, más concretamente, las relaciones de vinculación entre autómatas y reglas, las de concurrencia de autómatas en reglas y las de competencia de reglas por autómatas. Dichas relaciones se definen con

cuatro niveles de certidumbre siendo la más débil la que puede obtenerse automáticamente en tiempo de diseño. Una vez explicadas las relaciones propone un algoritmo de cálculo que permite identificar tanto los subprocesos, las partes críticas del sistema e incluso los elementos superfluos o unificables, independientemente del conocimiento heurístico que sobre el sistema a modelar se tenga. Un ejemplo de este cálculo se presenta sobre un sistema tipo donde los nombres de reglas, autómatas, acciones y estados han sido codificados para ocultar dicha heurística.

- Se presentan dos herramientas software. La primera es un sistema de ayuda para el modelado con los Autómatas Cooperativos Extendidos que guía al diseñador en la descripción de las diferentes partes de las que consta el modelo mientras verifica la sintaxis del parte de la lógica del mismo y a su vez permite realizar el análisis a priori antes mencionado de forma automática en cualquier momento del diseño, presentando al diseñador un informe del resultado del mismo. La segunda es un simulador que simula tanto el sistema AC como el extendido y que permite el uso de no determinismo en la elección tanto de reglas como de autómatas, la ejecución paso a paso y la ejecución secuencial de cierto número de reglas de transacción elegidas aleatoriamente sobre las habilitadas en cada momento.

Capítulo 2

Bases teóricas

2.1. Motivación

Como hemos planteado en el capítulo introductorio, para aprovechar al máximo las ventajas y facilidades que impone la utilización de estos modelos orientados a los agentes, hemos de proveer a nuestro modelo de una base teórica adecuada y de una facilidad de manejo para su utilización en la industria. Nuestra propuesta nace con la idea de aunar las características deseables de la Teoría de Autómatas y de las redes de Petri, permitiendo además expresar la teoría de agentes asociada de manera que se representen y definan individualmente cada una de las componentes de un sistema complejo así como las interacciones entre ellas.

Es por ello que conviene recordar brevemente las nociones básicas y las clasificaciones más frecuentes de la Teoría de Autómatas y sobre las redes de Petri, como presentamos en las siguientes secciones.

2.2. Teoría de Autómatas

El origen de los autómatas finitos probablemente se remonta a su uso implícito en máquinas electromecánicas, desde principios del siglo XX.

Ya en 1907, el matemático ruso Andréi Márkov formalizó un proceso llamado cadena de Markov [104], donde la ocurrencia de cada evento depende con una cierta probabilidad del evento anterior. Una cadena de Markov es un proceso estocástico con la propiedad de Markov, con sistemas que siguen una serie de eventos vinculados, donde lo que sucede después depende sólo del estado actual del sistema que describe.

Posteriormente, en 1943, surge una primera aproximación formal de los autómatas finitos con el modelo neuronal de McCulloch-Pitts, en [105]. Durante la década de 1950 prolifera su estudio, frecuentemente llamándoseles máquinas de secuencia; se establecen muchas de sus propiedades básicas, incluyendo su interpretación como lenguajes regulares y su equivalencia con las expresiones regulares. Al final de esta década, en 1959, surge el concepto de autómata finito no determinista en manos de los informáticos teóricos Michael O. Rabin y Dana Scott en [118].

La capacidad de las cadenas de Markov de "recordar" su estado es utilizada por los autómatas finitos que poseen una memoria primitiva similar, en que la activación

de un estado también depende del estado anterior, así como del símbolo o palabra presente en la función de transición.

Un autómata finito es un modelo matemático de un sistema, con entradas y salidas discretas. El sistema puede estar en cualquiera de un número finito de configuraciones o estados. El estado del sistema resume la información concerniente a entradas anteriores y que es necesaria para determinar el comportamiento del sistema para entradas posteriores. El sistema no recordará todas las entradas previas, sino tan sólo el estado actual del sistema, el conjunto de acciones válidas partiendo de dicho estado y las entradas aún no atendidas.

En la ciencia de la computación es posible encontrar muchos ejemplos de sistemas de estados finitos, y la teoría de autómatas finitos [78] es una útil herramienta para el diseño de tales sistemas. El ejemplo típico es un circuito lógico, con una serie de estados posibles.

Ciertos programas, utilizados con frecuencia, como los editores de texto y los analizadores léxicos que se encuentran en la mayoría de compiladores son diseñados como sistemas de estados finitos. Por ejemplo, un analizador léxico recorre los símbolos de un programa de computación para localizar las cadenas de caracteres correspondientes a identificadores, constantes numéricas, palabras reservadas, etcétera. En este proceso, el analizador léxico necesita recordar solamente una cantidad finita de información, como la longitud de un prefijo ya recorrido de una palabra reservada. La teoría de autómatas finitos se utiliza en el diseño de procesadores eficientes de cadenas de este tipo y de otros.

Un computador puede ser visto también como un sistema de estados finitos, aunque en este caso dado que el conjunto de estados posibles es muy amplio, el modelo no es tan útil como cabría esperar. Teóricamente el estado del procesador, de la memoria principal, de la secundaria, etc. en cualquier instante es uno cualquiera de un vasto conjunto de estados, suponiendo por supuesto un número fijo de dispositivos, una memoria limitada, etc.

2.2.1. Nociones básicas

Un autómata finito (AF) [78] consiste en un conjunto de estados finito y un conjunto de transiciones entre estados que se dan sobre símbolos de entrada tomados de un alfabeto Σ. Para cada símbolo de entrada existe exactamente una transición a partir de cada estado (que puede incluir el regreso a dicho estado). Un estado, por lo general denotado como q_0, es el estado inicial en el que el autómata comienza. Algunos estados están designados como final o de aceptación.

A un AF se le asocia un grafo dirigido etiquetado conocido como el diagrama de transiciones de la manera siguiente. Los vértices del grafo corresponden a los estados del AF. Si existe una transición del estado q al p sobre la entrada a entonces en el diagrama de transiciones existe un arco etiquetado con a que conduce del nodo q al nodo p. El AF acepta una cadena x si la secuencia de transiciones correspondientes a los símbolos de x conduce del estado inicial q_0 a un estado final.

Puede verse un ejemplo de diagrama de transición de estados en la Figura 2.1.

Definición 1 (Autómata Finito) *[78] Se denota formalmente a un AF por la siguiente construcción:* $(Q, \Sigma, \delta, q_0, F)$*, tales que:*

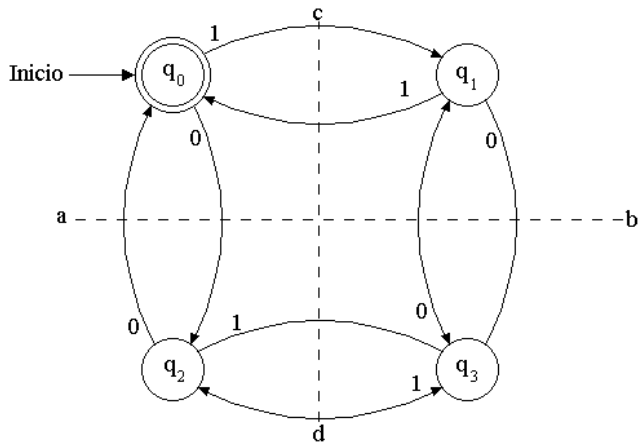


Figura 2.1: Diagrama de transiciones de un Autómata Finito

1. *Q es un conjunto finito de estados,*
2. *Σ es un alfabeto finito de entrada (las acciones),*
3. *δ es la función de transición que transforma $Q \times \Sigma$ en Q, esto es, $\delta(q, a)$ es un estado para cada estado q y símbolo a,*
4. *q_0 es el estado inicial y pertenece a Q y*
5. *F el conjunto de estados finales tal que $F \in Q$.*

Puede entenderse un AF como un control finito que se encuentra en algún estado Q y lee una secuencia de símbolos Σ escritos en una cinta según se muestra en la Figura 2.2. En un movimiento el AF que se encuentra en un estado q y está leyendo el simbolo a pasa a un estado $\delta(q, a)$ y mueve su cabeza lectora un símbolo a la derecha. Si el estado $\delta(q, a)$ es un estado final se considera entonces que el AF ha aceptado la cadena escrita (la secuencia de acciones) desde la posición de entrada hasta la anterior a la que apunta la cabeza lectora. Si la cabeza lectora se ha salido por el extremo derecho de la cinta, la secuencia de acciones ha sido aceptada completamente.

Para describir formalmente el comportamiento de un AF sobre una cadena de elementos del alfabeto Σ es necesario extender la función de transición δ para que se pueda aplicar sobre un estado y una cadena más que sobre un estado y un símbolo. Se define la función $\widehat{\delta}$ de $Q \times \Sigma^*$ a Q. La intención es que $\widehat{\delta}(q, w)$ represente el estado al que llegará un AF después de ejecutar la secuencia de acciones w, partiendo del estado q. Dicho de otra forma, $\widehat{\delta}(q, w)$ es el único estado p tal que existe una treyectoria en el diagrama de transiciones de q a p con la etiqueta w. De manera formal se define:

1. $\widehat{\delta}(q, \in) = q$
2. para todas las cadenas w y símbolos de entrada a, $\widehat{\delta}(q, wa) = \delta(\widehat{\delta}(q, w), a)$.

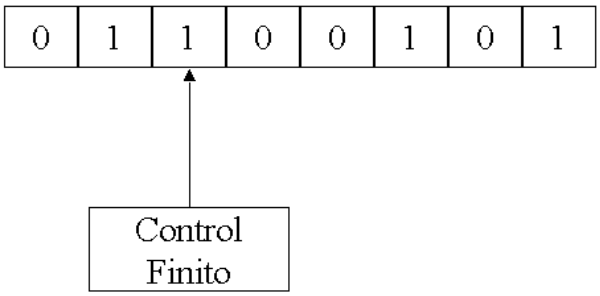


Figura 2.2: Un autómata finito

De esta forma, (1) establece que sin leer un símbolo de entrada el AF no puede cambiar de estado, y (2) nos dice cómo encontrar el estado después de leer una cadena de entrada no vacía wa. Esto es, hallar el estado $p = \widehat{\delta}(q, w)$ después de leer w. Y entonces, calcular el estado $\delta(p, a)$.

Puesto que $\widehat{\delta}(q, a) = \delta(\widehat{\delta}(q, \in), a) = \delta(q, a)$ [haciendo $w = \in$ en la regla (2)] no puede haber desacuerdo entre δ y $\widehat{\delta}$ sobre los argumentos para los cuales se definieron. Y a partir de este instante se usará δ en lugar de $\widehat{\delta}$ por conveniencia.

Se dice que una cadena x es aceptada por un autómata finito $M = (Q, \Sigma, \delta, q_0, F)$ si $\delta(q_0, x) = p$ para algún $p \in F$. El *lenguaje aceptado* por M, denominado $L(M)$, es el conjunto $\{x|\delta(q_0, x) \in F\}$. Un lenguaje es *regular* si es el lenguaje aceptado por algún autómata finito.

2.2.2. Autómatas finitos no deterministas

El concepto de no determinismo juega un papel clave en las teorías de lenguajes y computación. Considérese una modificación al modelo general de autómatas finitos para permitirle ninguna, una o varias transiciones de un estado sobre el mismo símbolo de entrada. Este nuevo modelo se conoce como *Autómata finito no determinista (AFND) [118]* un ejemplo del cual puede verse en la Figura 2.3. En dicha figura puede observarse que existen aristas, como las dos que parten del estado q_0 y están etiquetadas con 0. Una de ellas parte de dicho estado y llega al estado q_3 y la otra, regresa al mismo estado q_0.

Una secuencia de entrada $a_1, a_2, ...a_n$ es aceptada por un AFND si existe una secuencia de transiciones correspondiente a las secuencia de entrada que conduzca del estado inicial a algún estado final. Por ejemplo, 01001 es aceptado por el AFND de la Figura 2.3 debido a que existe una secuencia de transiciones particular que pasa por los estados $q_0, q_0, q_0, q_3, q_4, q_4$ etiquetadas por $0, 1, 0, 0, 1$. Este AFND en particular acepta todas las cadenas que tengan dos ceros consecutivos o dos unos. Si se representa el AFND con una cinta de entrada y con un control finito de estados se tendrá que, en un instante dado, el control finito podrá encontrarse en cierto número de estados. Para entender esto, se puede imaginar que se cuenta con un duplicado del autómata por cada posible estado en el que se pueda encontrar tras cualquier transición.

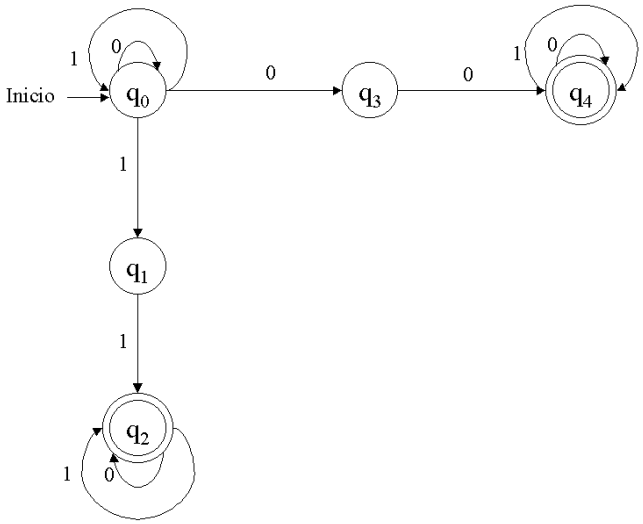


Figura 2.3: un autómata finito

De manera formal, se denota un AFND mediante la quíntupla $(Q, \Sigma, \delta, q_0, F)$ en donde Q, Σ, q_0 y F esto es, el conjunto de estados, el alfabeto de acciones, el estado inicial y el conjunto de estados finales tienen el mismo significado que en el AF y sin embargo, δ es una transformación de $Q \times \Sigma$ en 2^Q (esto es, el conjunto potencia de Q, es decir, el conjunto de partes de Q). La intención es que donde antes $\delta(q, a)$ era un estado al que se llegaba tras aplicar la transición a en el autómata cuando se encontraba en el estado q ahora se tendrá el conjunto de todos los estados p tales que exista una transición con etiqueta a desde el estado q a p. Para el autómata de la Figura 2.3 se tendrá la siguiente función de transformación δ:

Estado	Transición 0	Transición 1
q_0	$\{q_0, q_3\}$	$\{q_0, q_1\}$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$

La función δ puede extenderse a una función $\widehat{\delta}$ de $Q \times \Sigma^*$ a 2^Q y que refleje sucesiones de entradas de la manera siguiente:

1. $\widehat{\delta}(q, \in) = q$
2. $\widehat{\delta}(q, wa) = \{p|$ para algún estado para r de $\widehat{\delta}(q, w)$ p esté en (r,a)$\}$.

De forma que la condición (1) no permite un cambio de estado sin que se efectúe una transición y la condición (2) indica que al iniciar en el estado q y leer la cadena w seguida por el símbolo de entrada a se puede llegar al estado p si y sólo si uno de los

estados posibles en los que podamos estar después de efectuar todas las transiciones denotadas por w es r y de r podemos llegar a p efectuando a.

Al igual que ocurría antes, puesto que $\widehat{\delta}(q,a) = \delta(q,a)$ se puede utilizar de nuevo δ en lugar de $\widehat{\delta}$. Del mismo modo, es útil extender δ a argumentos en 2^Q de forma que:

3. $\delta(P,w) = \cup_{q\ \in P}\delta(q,w)$

para cada conjunto de estados de P $\subseteq Q.L(M)$, en el que M está en el AFND (Q,Σ,δ,q_0,F), es $\{w|\delta(q_0,w)$ contiene un estado en $F\}$

2.2.3. Equivalencia entre AFDs y AFNDs

Puesto que cada autómata finito determinista es también por definición un autómata finito no determinista, queda claro que la clase de lenguaje aceptado por los AFNDs incluye a los conjuntos regulares (aquellos aceptados por AFDs). Sin embargo, resulta que esos son los únicos conjuntos aceptados también por los AFNDs. La demostración consiste en mostrar que los AFDs pueden simular a los AFNDs, esto es, para cada AFND podemos construir uno determinista equivalente (uno que acepte el mismo lenguaje). Este concepto de equivalencia será usado más adelante cuando digamos que el modelo extendido propuesto en esta tesis doctoral es equivalente a aquel al que extiende.

La forma en la que un AFD simula a un AFND consiste en permitir que los estados del AFD correspondan a conjuntos de estados del no determinista. El autómata que se ha construido mantiene, en su control finito, el rastro de todos los estados en los que el autómata no determinista al que simula puede llegar tras leer la misma entrada leída en el determinista. La costrucción formal se encuadra en el Teorema 1.

Teorema 1 *Sea L un conjunto aceptado por un Autómata Finito No Determinista. Entonces existe un Autómata Finito Determinista que acepta L.*

Prueba 1 *Sea $M = (Q,\Sigma,\delta,q_0,F)$ un Autómata Finito no determinista que acepta a L. Defínase un Autómata Finito Determinista $M = (Q',\Sigma,\delta',q_0',F')$, de la manera siguiente. Los estados de M' son todos los subconjuntos del conjunto de estados de M. Es decir, $Q' = 2^Q$. M conservará, en su estado, el rastro de todos los estados en los que puede estar M en cualquier tiempo dado. F' es el conjunto de todos los estados de Q' que contienen un estado final de M. Un elemento de Q' será denotado por $[q_1,...,q_i]$ en donde $q_1,...,q_i$, se encuentran en Q. Observese que $[q_1,...,q_i]$ es un solo estado del AFD que corresponde a un conjunto de estados del AFD. Nótese también que $q_0' = [q_0]$.*

Se define $\delta'([q_1,...,q_i],a) = [p_1,...,p_i]$ si y sólo si $\delta(q_1,...,q_i,a) = \{p_1,...,p_j\}$ esto es, que δ' aplicada a un elemento $[q_1,...,q_i]$ de Q' se calcula aplicando δ a cada estado de Q que esté representado por $[q_1,...,q_i]$. Al aplicar δ a cada $q_1,...,q_i$ y tomar la unión, obtenemos algún conjunto nuevo de estados $p_1,...,p_j$. Este nuevo conjunto de estados tiene un representante, $[p_1,...,p_j]$ en Q' y este elemento es el valor de $\delta(q_1,...,q_i,a)$

Es fácil mostrar por inducción sobre la longitud de la cadena de entrada x que

$$\delta'(q'_0, x) = [q_1, ..., q_i] \text{ si y sólo si } \delta(q_0, x) = \{q_1, ..., q_i\}.$$

Base *El resultado es trivial para* $|x| = 0$ *ya que* $q'_o = [q_o]$ *y* x *debe ser* $\in$

Inducción *Supongamos que la hipótesis es verdadera para entradas de longitud* m *o menores. Sea* xa *una cadena de lngitud* $m + 1$ *con* a *en* Σ. *Entonces*

$$\delta'(q'_0, xa) = \delta'(\delta'(q'_0, x), a).$$

y tomando la hipótesis de inducción, sabemos que

$$\delta'(q'_0, x) = [p_1, ..., p_j] \text{ si y sólo si } \delta(q_0, x) = p_1, ..., p_j$$

Pero por la definición de δ' *sabemos que*

$$\delta'([p_1, ..., p_j], a) = [r_1, ..., r_k]. \text{ si y sólo si } \delta p_1, ..., p_j, a) = r_1, ..., r_k$$

y, por tanto,

$$\delta'(q'_0, xa) = [r_1, ..., r_k] \text{ si y sólo si } \delta(q_0, xa) = r_1, ..., r_k$$

que establece la hipótesis inductiva.

Para completar la prueba, sólo falta añadir que $\delta'(q'_0, x)$ *está en* F' *exactamente cuando* $\delta(q_0, x)$ *contiene un estado de* Q *que esté en* F. *Así pues,* $L(M) = L(M')$

Dado que ambos tipos de autómatas resultan ser equivalentes en el sentido de que aceptan los mismos conjuntos de entrada, o lo que es lo mismo, soportan las mismas transiciones, no se hará distinción alguna entre ellos y sólo nos referiremos a los Autómatas Finitos (AF).

En la práctica, muchos estados de un AFND no son accesibles partiendo del estado inicial q_0 y, por tanto, suele resultar adecuado comenzar únicamente con dicho estado inicial e ir añadiendo estados al AFD sólo cuando sean resultado de una transición partiendo de un estado previamente añadido.

2.3. Redes de Petri

Las redes de Petri, introducidas por Carl Adam Petri a principios de los años 60 [116], son una herramienta gráfica y matemática para el estudio de un gran número de sistemas: son uno de los formalismos más ampliamente aceptados para modelizar sistemas concurrentes y distribuidos. El éxito de las redes de Petri en los últimos 50 años puede medirse no sólo por las incontables situaciones en las que se pueden aplicar sino también por el desarrollo de aspectos teóricos, que van desde un análisis completo de distintos fenómenos en los modelos más simples a la definición de clases de redes con mayor expresividad (y más complejas). La principal aplicación de las redes de Petri es la *modelización* y el *análisis* de sistemas con componentes concurrentes que interactúan. Un modelo es una representación (en términos matemáticos) de las características más importantes del objeto o sistema de estudio. Manipulando esta representación, se pueden obtener nuevos conocimientos del sistema modelado sin ningún coste o peligro para el sistema real. Sin embargo, el modelado por sí solo sirve de poco. Es necesario analizar el sistema modelado. El sistema se modela primero como una red de Petri y después, este modelo se analiza. Este análisis nos lleva a una mejor comprensión del comportamiento del sistema modelado. Para realizar el análisis de las propiedades de una red de Petri se han desarrollado diferentes técnicas, automáticas o no, según los casos, que permiten la *verificación* de las propiedades que el sistema construido posea. De hecho, una línea de trabajo muy importante ha sido la construcción de herramientas automáticas de verificación para las diferentes extensiones que sobre el modelo de las redes de Petri se han propuesto.

A continuación, en la Sección 2.3.1 presentamos los conceptos básicos de las redes de Petri. Después inlcluimos la definición de las subclases de redes de Petri más habituales en 2.3.2. Y finalmente en la Sección 2.3.3 mostramos el formalismo de las redes de Petri Objetuales de Valk [139], estrechamente relacionado con el formalismo de los Autómatas Cooperativos.

2.3.1. Conceptos Básicos

Definición 2 (red de Petri [115, 109][120]) *Una* red de Petri *es una tupla* $N = (P, T, F)$ *donde* $P = \{p_1, p_2, \ldots, p_m\}$ *es un conjunto finito de lugares,* $T = \{t_1, t_2, \ldots, t_n\}$ *es un conjunto finito de transiciones, tales que* $P \cap T = \emptyset$ *y* $P \cup T \neq \emptyset$ *y* $F : (P \times T) \cup (T \times P) \to \{0, 1, 2, 3, \ldots\}$ *es un conjunto de arcos ponderados (o relación de flujo).*

Definición 3 (Marcado) *Un* marcado *es una aplicación* $M : P \to \{0, 1, 2, 3, \ldots\}$ *que asigna a cada lugar un entero no negativo (*token*), de forma que decimos que un lugar* p está marcado *con* k *tokens si el marcado asigna al lugar* p *un entero* k*. Un marcado* M *se representa mediante un vector con tantas componentes como lugares tenga la red. El número de tokens en el lugar* p *será* $M(p)$*.*

Definición 4 (red de Petri Marcada) *Una* red de Petri marcada *es un par* (N, M) *formado por una red de Petri* N *y un marcado* M*.*

A una *red de Petri* podemos asociarle un grafo dirigido con dos clases disjuntas de nodos, los *lugares* y las *transiciones*. Un círculo ◯ representa un lugar y una

barra ▯ representa una transición. Los arcos dirigidos (flechas) conectan lugares y transiciones. Algunos arcos van desde un lugar a una transición y otros desde una transición a un lugar. Un arco dirigido desde un lugar p a una transición t define p como un *lugar de entrada* para t. Un *lugar de salida* se indica con un arco desde la transición al lugar. Una transición tiene un determinado número de *lugares de entrada* (o *precondiciones*) y de *lugares de salida* (o *postcondiciones*). Los arcos se etiquetan con sus pesos (enteros positivos). Si una de esas etiquetas se omite, significa que el arco tiene peso uno. Los tokens se representan como puntos negros • en los lugares.

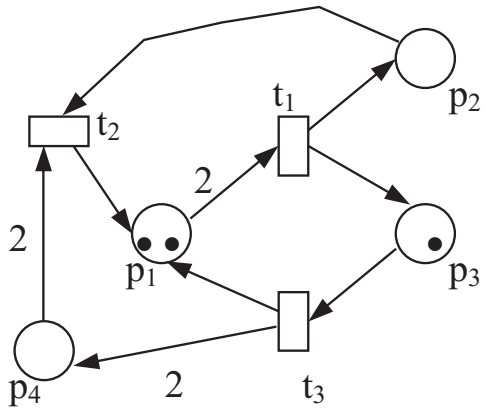


Figura 2.4: Red de Petri (a)

Definición 5 (Preset de una transición t) *El* preset *de una transición t es el conjunto de todos los lugares de entrada a la transición t,* $^\bullet t = \{\forall p \in P \mid F(p,t) \neq 0\}$.

Definición 6 (Postset de una transición t) *El* postset *de una transición t es el conjunto de todos los lugares de salida de la transición t,* $t^\bullet = \{\forall p \in P \mid F(t,p) \neq 0\}$.

Definición 7 (Preset de un lugar p) *El* preset *de un lugar p es el conjunto de todas las transiciones de entrada al lugar p,* $^\bullet p = \{\forall t \in T \mid F(t,p) \neq 0\}$.

Definición 8 (Postset de un lugar p) *El* postset *de un lugar p es el conjunto de todas las transiciones de salida del lugar p,* $p^\bullet = \{\forall t \in T \mid F(p,t) \neq 0\}$.

Ejemplo 1 *Para la red de Petri (N, M) de la Figura 2.4, los conjuntos preset y postset de transiciones y lugares son los siguientes:*

$^\bullet t_1 = \{p_1\}$	$t_1^\bullet = \{p_2, p_3\}$	$^\bullet p_1 = \{t_2, t_3\}$	$p_1^\bullet = \{t_1\}$
$^\bullet t_2 = \{p_2, p_4\}$	$t_2^\bullet = \{p_1\}$	$^\bullet p_2 = \{t_1\}$	$p_2^\bullet = \{t_2\}$
$^\bullet t_3 = \{p_3\}$	$t_3^\bullet = \{p_1, p_4\}$	$^\bullet p_3 = \{t_1\}$	$p_3^\bullet = \{t_3\}$
		$^\bullet p_4 = \{t_3\}$	$p_4^\bullet = \{t_2\}$

El cambio en el marcado de la red es el que simula el comportamiento dinámico de un sistema modelado mediante una red de Petri. Una red de Petri se ejecuta por

el *disparo* de transiciones. Una transición se *dispara* eliminando tokens de sus lugares de entrada y creando nuevos tokens distribuidos por sus lugares de salida.

Una transición puede dispararse si está *habilitada*. Una transición está habilitada si en cada uno de sus lugares de entrada hay, al menos, tantos tokens como el peso de los arcos que los conectan con la transición. Formalmente,

Definición 9 (Transición Habilitada) *Una transición t se dice* habilitada *si cada lugar de entrada p de t está marcado con al menos $F(p,t)$ tokens, es decir, con el entero que representa el peso del arco de p a t,*

$$\forall p \in {}^\bullet t \quad M(p) \geq F(p,t)$$

El *disparo* de una transición habilitada hace que se eliminen tantos tokens de sus lugares de entrada como indique el peso de los arcos que los conectan con la transición, y hace que en sus lugares de salida se depositen tantos tokens como indique el peso de los arcos que conectan dichos lugares con la transición.

Disparar una transición cambia el marcado M de una red de Petri a un nuevo marcado M'. Puesto que únicamente pueden dispararse las transiciones habilitadas, el número de tokens de cada lugar siempre será positivo cuando una transición se dispare. Si no hay tokens suficientes en alguno de los lugares de entrada de una transición, entonces la transición no estará habilitada y no se disparará.

Definición 10 (Disparo de una transición) *Una transición puede* dispararse *siempre que esté habilitada.*

El disparo *de una transición t habilitada en un marcado M elimina $F(p,t)$ tokens de cada lugar de entrada p de t, y añade $F(t,p)$ tokens a cada lugar de salida p de t, donde $F(t,p)$ es el peso del arco de t a p.*

El marcado M' resultado del disparo de una transición t habilitada en un marcado M, $M[t\rangle M'$, se define como sigue

$$\begin{array}{ll} M'(p) = M(p) - F(p,t) + F(p,t) & \forall p \in ({}^\bullet t \cap t^\bullet) \\ M'(p) = M(p) - F(p,t) & \forall p \in {}^\bullet t \backslash t^\bullet \\ M'(p) = M(p) + F(t,p) & \forall p \in t^\bullet \backslash {}^\bullet t \\ M'(p) = M(p) & \textit{en otro caso} \end{array}$$

Ejemplo 2 *Las transiciones habilitadas en la red de Petri de la Figura 2.4 son t_1 y t_3. Si se dispara la transición t_1, el marcado de la red $M = (2,0,1,0)$ cambia al marcado $M' = (0,1,2,0)$ como se muestra en la Figura 2.5.*

2.3.2. Subclases de redes de Petri más habituales

Definición 11 (red de Petri Pura) *Una red de Petri se llama* pura *si no tiene bucles, es decir, $\forall t \in T\ \{{}^\bullet t\} \cap \{t^\bullet\} = \emptyset$, los lugares de entrada de t no son a la vez lugares de salida de t.*

Definición 12 (red de Petri Ordinaria) *Una red de Petri se dice* ordinaria *si el peso de todos sus arcos es* 1.

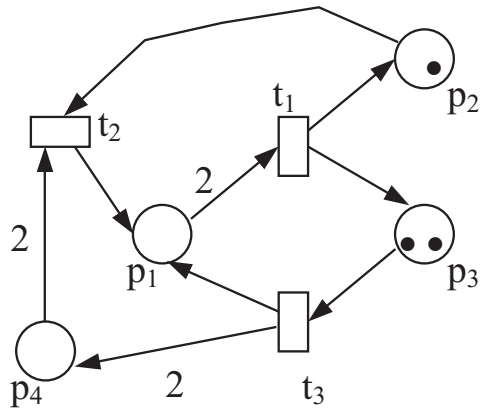


Figura 2.5: Marcado de la red de Petri de la Figura 2.4 después del disparo de t_1

Definición 13 (red de Petri Simple) *Una red de Petri* simple *es una red de Petri ordinaria tal que:* $p_1^\bullet \cap p_2^\bullet \neq \emptyset \Rightarrow (p_1^\bullet \subseteq p_2^\bullet) \vee (p_2^\bullet \subseteq p_1^\bullet)$.

Definición 14 (Grafo Marcado) *Un* grafo marcado *es una red de Petri ordinaria en la que cada lugar tiene exactamente una transición de entrada y una de salida, es decir,* $|^\bullet p| = |p^\bullet| = 1 \ \forall p \in P$.

Definición 15 (Máquina de Estados) *Una* máquina de estados *es una red de Petri ordinaria en la que cada transición tiene exactamente un lugar de entrada y uno de salida, es decir,* $|^\bullet t| = |t^\bullet| = 1 \ \forall t \in T$.

Definición 16 (Red de Libre Elección (Free-Choice Net)) *Una* red de libre elección *es una red de Petri ordinaria tal que un arco desde un lugar es el único arco saliente o es la única entrada a una transición, es decir,* $\forall t_j \in T$ *y* $p_i \in {}^\bullet t_j$, ${}^\bullet t_j = \{p_i\}$ *o* $p_i^\bullet = \{t_j\}$.

Definición 17 (red de Petri Lugar/Transición (Red L/T)) *Una* red de Petri Lugar/Transición *es una 6-tupla* $N = (P, T, F, K, M, W)$ *tal que*

- (P, T, F) *es una red finita siendo* P *y* T *un conjunto de lugares y transiciones, respectivamente;*
- $K : P \to \mathbb{N} \cup \{\omega\}$ *da una capacidad (posiblemente ilimitada) a cada lugar;*
- $W : F \to \mathbb{N} \backslash \{0\}$ *asigna un peso a cada arco de la red;*
- $M : P \to \mathbb{N} \cup \{\omega\}$ *es el marcado inicial, respetando las capacidades, es decir,* $M(p) \leq K(p)$ *para todo* $p \in P$.

En la siguiente definición, damos la regla de disparo para redes L/T.

Definición 18 (Regla de Disparo para redes L/T) *Sea N una red Lugar/Transición,*

Una aplicación $M : P \to \mathbb{N} \cup \{\omega\}$ se llama marcado *de N sii $M(p) \leq K(p)$ para todo $p \in P$.*

Una transición $t \in T$ está habilitada *en un marcado M sii*

$$\forall p \in {}^\bullet t : M(p) \geq W(p,t) \text{ y } \forall p \in t^\bullet : M(p) \leq K(p) - W(t,p)$$

Una transición $t \in T$ habilitada en M conduce a un marcado M', $M[t\rangle M'$, tal que para cada $p \in P$

$$M'(p) = \begin{cases} M(p) - W(p,t) & sii\ p \in {}^\bullet t \backslash t^\bullet \\ M(p) + W(t,p) & sii\ p \in t^\bullet \backslash {}^\bullet t \\ M(p) - W(p,t) + W(t,p) & sii\ p \in {}^\bullet t \cap t^\bullet \\ M(p) & en\ otro\ caso \end{cases}$$

2.3.3. Redes de Petri Objetuales.

Dentro de los distintos tipos de redes de Petri, tenemos un sistema denominado redes de Petri Objetuales de R. Valk [139]. Este tipo de redes de Petri es especialmente útil para la representación de sistemas complejos. El sistema básico se representa por una red de Petri, llamada Red Sistema. Los objetos del sistema, representados por tokens de esta Red de Sistema, tienen un comportamiento dinámico y son a su vez redes de Petri llamadas Redes Objeto. Entonces las redes de Petri Objetuales están compuestas por una Red Sistema y una o más Redes Objeto. En este contexto, Valk desarrolló primero modelos básicos denominados Sistemas de Objetos Elementales. En estas redes, las Redes Objeto están restringidas a ser redes elementales. En un sistema de objetos elemental unario, hay una única red objeto de la que pueden existir múltiples copias. Estudiando este modelo, Valk propuso diferentes nociones de marcado (bi-marcado, marcado de objetos y marcado de procesos) y las reglas de ocurrencia. El mostró que, en el caso general, la noción adecuada para una representación correcta de las estructuras *fork/join* (bifurcación/enlace) es el marcado de procesos. El bi-marcado y el marcado de objetos son nociones adecuadas para casos especiales.

En un sistema de objetos elementales simple, hay varias Redes Objeto, de cada una existe un único ejemplar. Los objetos pueden interactuar con la Red Sistema y con otras Redes Objeto. La interacción entre las Redes Objeto puede tener lugar sólo si las dos redes están en el mismo lugar en la Red Sistema.

La presentación de un sistema de objetos elemental simple que viene dada a continuación difiere ligeramente de la dada en [139].

Definición 19 ***(Sistema de Objetos elemental simple (EOS))****[140] Un sistema de objetos elemental simple es una tupla $EOS = (SN, \mathbf{ON}, I, \mathbf{M})$ donde:*

- *$SN = (P_1, P_2, T, W_1, W_2)$ es una red denominada Red Sistema de EOS, donde P_1 es un conjunto de lugares estándar, P_2 es un conjunto de lugares objeto, T es un conjunto de transiciones, $W_1 : P_1 \times T \cup T \times P_1 \to IN$ $W_2 : P_2 \times T \cup T \times P_2 \to 2^{\{1,\ldots,n\}}$ son las transiciones de flujo,*
- *$\mathbf{ON} = \{ON_1, ..., ON_n\} (n \geq 1)$ es un conjunto finito de sistemas de redes elementales disjuntos, llamados redes objeto de EOS, y denotado por $ON_i = (B_i, E_i F_i m_{o_i})$,*

- $I = (\rho, \sigma)$ *es la relación de interacción que consiste en una interacción entre sistema/objeto* $\rho \subseteq T \times E$ *donde* $E := \bigcup_{i=1}^{n} E_i$ *y una relación de interacción simétrica objeto/objeto* $\sigma \subseteq (E \times E) \backslash id_E$. *I está separada, es decir,* $e\sigma e' \Rightarrow \rho e = \emptyset = \rho e', e \in E, e' \in E$,
- $\mathbf{M}$ *es un marcado definido más adelante.*

Definición 20 ***(Marcado de Objetos)*** *[140] El conjunto* $Obj := (\{(ON_i, m_i) | 1 \leq i \leq n, m_i \in R(ON_i)\}$ *es el conjunto de objetos de EOS. Un marcado de Objetos es un par* (M_1, M_2) *con* $M_1 : P_1 \to IN$ *y* $M_2 : P_2 \to 2^{Obj}$.

La componente i $(1 \leq i \leq n)$ de EOS es la Red de sistema elemental $SN(i) = (P_2, T, W_2(i), M_{0i})$ definida por $W_2(i) = \{(x, y)/i \in W_2(x, y)\}$ y $M_{0i}(p) = 1$ si $(ON_i, m_i) \in M_2(p)$. Se asume que todos los componentes i $(1 \leq i \leq n)$ de SN son máquinas de estados. La componente 0 es la red de Petri $SN(O) = (P_1, T, W_1, M_1)$.

Las reglas de ocurrencia de un sistema de objetos simple elemental se define como sigue.

Definición 21 ***(Reglas de ocurrencia)****[140] Sea* $EOS = (SN, \mathbf{ON}, I, \mathbf{M})$ *una red de sistema simple elemental, con un marcado* $\mathbf{M} = (M_1, M_2)$.

1. *Una transición* $t \in T$ *se activa en* $\mathbf{M}$ *(denotado por* $\mathbf{M} \xrightarrow{t}$*) si* $t\rho = \emptyset$ *y*

   (a) *t se activa en la componente-0 de* $SN : M_1 \xrightarrow{t}$

   (b) *para cada* $q_1 \in P_2 \cap {}^\bullet t$ *hay un objeto* $(ON_i, m_i) \in M_2(q_1)$ *con* $i \in W_2(q_1, t)$.

   Si t está activada entonces t puede ocurrir $(\mathbf{M} \xrightarrow{t} \mathbf{M}')$ *con* $M' = (M_1', M_2')$ *definido por*

   - $M_1 \xrightarrow{t} M_1'$ *en la componente 0,*
   - *el objeto* (ON_i, m_i) *se borra de* q_1 *y se añade a* q_2 *al que* $q_2 \in P_2 \cap t^\bullet$ *con* $i \in W2(t, q_2)$ *(tal que* q_2 *es únicamente determinada debido al hecho que la componente i es una máquina de estados).*

2. *Un par* $[t, e] \in T \times E_i$ *con* $t\rho e$ *se activa en* $\mathbf{M}$ *(denotado por* $\mathbf{M} \xrightarrow{[t,e]}$*) si las condiciones del caso 1 se mantienen y la transición e se activa para* ON_i *en el marcado* m_i. *La marca siguiente* $\mathbf{M}' = (M_1', M_2')$ *se define por*

   - $M_1 \xrightarrow{t} M_1'$ *en la componente 0,*
   - *el objeto* (ON_i, m_i) *se borra de* q_1 *y el objeto* (ON_i, m_i'), *donde* $m_i \xrightarrow{t} m_i'$, *se añade a* q_2.

3. *Un par* $[e_i, e_j] \in E_i \times E_j$ *con* $e_i \sigma e_j$ *se activa en* $\mathbf{M}$ *(denotada* $\mathbf{M} \xrightarrow{[e_i, e_j]}$*) si algún lugar* $p \in P_2$ *contiene dos objetos* (ON_i, m_i) *y* (ON_j, m_j) *tal que* $m_i \xrightarrow{e_i}$ *y* $m_i \xrightarrow{e_j}$. *El marcado siguiente* $\mathbf{M}' = (M_1', M_2')$ *se define por*

   - $M_1 = M_1'$,

- *los objetos* (ON_i, m_i) *y* (ON_j, m_j) *en* p *se reemplazan por los objetos* (ON_i, m_i') *y* (ON_j, m_j') *donde* $m_i \xrightarrow{e_i} m_i'$, *y* $m_j \xrightarrow{e_j} m_j'$

4. *Una transición* $e_i \in E_i$ *con* $e_i\sigma = \rho e_i = \emptyset$ *está activada en* $\mathbf{M}$ *(denotada por* $\mathbf{M}_i \xrightarrow{e_i}$*) si algún lugar* $p \in P_2$ *contiene un objeto* (ON_i, m_i) *tal que* $m_i \xrightarrow{e_i}$. *El marcado siguiente* $\mathbf{M'} = (M_1', M_2')$ *se define por*

   - $M_1 = M_1'$,
   - *el objeto* (ON_i, m_i) *en* p *se reemplazan por el objeto* (ON_i, m_i') *donde* $m_i \xrightarrow{e_i} m_i'$.

2.4. Otros Modelos de Autómatas y redes de Petri

Como ya hemos visto en el capítulo introductorio, los sistemas multiagente son un paso adelante en el planteamiento de la modelización de sistemas. Partiendo de esta idea general podemos buscar múltiples representaciones para este tipo de sistemas, pero, dentro de las representaciones gráficas, las que nos interesan en este trabajo son aquellas basadas en la teoría de Autómatas.

En la literatura podemos encontrar varios modelos basados en autómatas para representar, concretamente, la problemática de los sistemas distribuidos y las aplicaciones *groupware*. Hemos escogido tres modelos relacionados en mayor o menor medida con el de los Autómatas Cooperativos. Se trata de un modelo basado en la teoría de Autómatas y redes de Petri, otro orientado directamente a aplicaciones groupware y por último, otro basado en la cooperación de autómatas entendida como la compartición de estados.

2.4.1. Los Autómatas Finitos Paralelos

Una propuesta con gran aceptación en el diseño de sistemas distribuidos, concurrentes y paralelos utilizando autómatas, es la de los Autómatas Finitos Paralelos (AFP) propuesto por Stotts y Pugh [131], cuya aportación se resume a continuación.

Este formalismo, propuesto por en 1996 [131] presentaba como mayor contribución la utilidad de su particular notación y la manera en la que encaja en los modelos de desarrollo de software.

los AFP's combinan las capacidades de modelado de las redes de Petri sin admitir la posibilidad de un conjunto de estados infinito. La clase de autómatas resultante es una versión de máquina finita de estados, pero con la capacidad de expresar directamente formas de paralelismo sin que se vean involucradas en el concepto de estado.

La notación usada para representar un AFP es similar a la usada habitualmente para autómatas finitos deterministas y no deterministas, pero ligeramente modificada para expresar la actividad en paralelo. La modificación esencialmente engarza los elementos en una estructura tipo red de Petri, pero se representa en una notación que hace los AFP's más reconocibles como autómatas finitos puros.

Los AFP's tienen múltiples aplicaciones en la industria mostrando sistemas de ejecución en paralelo, y pese a que en general no muestran sistemas en los que intervienen distintos autómatas como es la base de los AC, sí que resultan un claro ejemplo

de cómo la teoría de autómatas, junto con las redes de Petri pueden relacionarse entre sí y dar lugar a un modelo capaz de representar la mayoría de sistemas concurrentes, paralelos y distribuidos de manera gráfica y con una gran sencillez de interpretación.

Descripción Formal

Vamos a presentar la definición formal de los miembros de la clase AFP, y luego una descripción de su representación y comportamiento. La notación y el estilo es el comúnmente usado en la teoría de automatas.

Definición 22 (Autómata Finito Paralelo (AFP) [131]) *Un autómata finito paralelo M se define formalmente como una séptupla $M = (N, Q, \Sigma, \gamma, \delta, q_0, F)$ donde:*

N *es un conjunto finito de nodos,*
$Q \subseteq 2^N$ *es un conjunto finito de estados,*
Σ *es un alfabeto de entrada finito,*
$\gamma : 2^N \times (\Sigma \cup \{\lambda\}) \to 2^{2^N}$ *es la función de transición de nodos,*
$\delta : Q \times (\Sigma \cup \{\lambda\}) \to 2^Q$ *es la función de transición de estados,*
$q_0 \in Q$ *es el estado inicial,*
$F \subseteq N$ *es el conjunto de nodos finales.*

y donde γ y δ son funciones parciales sujetas a restricciones. (véase definición original en [131])

En un AFP no hay correspondencia uno a uno entre los estados y los nodos del diagrama de transición, como suele ser habitual en otras representaciones de autómatas finitos. Así, la función de transición entre nodos γ se usa para generalizar la noción de arco, que podemos encontrar en los grafos dirigidos. Específicamente, si $((\{A, B\}, a), \{C, D, E\})$ es un elemento de γ, decimos que una transición etiquetada con a existe con los nodos origen A y B y con nodos destino C, D y E.

Un estado en AFP es un conjunto de nodos. Durante la ejecución, los nodos que componen el estado actual son denominados "activos". La función de transición δ se define entonces de la siguiente manera: Inicialmente, el conjunto de nodos activos para M es exactamente q_0, el estado inicial. Durante la ejecución de M podemos ver el símbolo c en el estado q, el conjunto de nodos activos que constituyen el siguiente estado de M es cualquiera de los conjuntos en d(q,c). Más específicamente, cada transición entre estados se define en base a las transiciones entre nodos como sigue.

Definición 23 (Regla de transición de estados (Regla de disparo)) *Dado un estado $q \in Q$ y un símbolo de entrada $c \in (\Sigma \cup \{\lambda\})$ tenemos que:*

$\delta(q, c) = \{(q - m) \cup n | n \in \gamma(m, c)$ *para* $m \subseteq q\}$

y se extiende la operación de δ a cadenas (de Σ^* en lugar de símbolos de Σ) usando la función $\widehat{\delta}$, como sigue:,

$\widehat{\delta}(q, \lambda) = \delta(q, \lambda)$
$\widehat{\delta}(q, \alpha a) = \{\delta(p, \lambda) |$ *para algún* $r \in \widehat{\delta}(q, \alpha), p \in \delta(r, a)\}$

Sin embargo, el significado de estas funciones aplicado a un conjunto de estados es simplemente la unión del resultado de aplicarlas a los estados individuales del conjunto. En lo sucesivo se utilizará únicamente el símbolo δ y según el contexto se distinguirá si se trata de una u otra.

El AFP M acepta una cadena w si para cada estado q perteneciente a $\delta(q_0, w), q \subseteq F$. La interpretación intuitiva de esto es que una cadena será aceptada si después de ser consumida completamente por alguna ejecución de M, cada nodo activo es un nodo final. Corespondientemente, el lenguaje aceptado por un AFP se define como:

$L(M) = \{w|\delta(q_0, w)$ contiene un subconjunto de $F\}$

Ocurrirá un error de ejecución si no existe una transición de estados posible desde el estado actual con el símbolo de entrada actual.

Representación y ejecución informal

La mejor manera de representar un AFP es en la forma de un grafo dirigido con transiciones etiquetadas como el de la Figura 2.7. El conjunto de nodos N en el AFP se representa por los nodos del grafo. Cada transición γ se representa dibujando de manera colectiva una serie de arcos desde los nodos origen hasta los destino. Estos arcos se denominan "arcos enlazados". Para cada transicion con múltiples orígenes o destinos, se dibuja un círculo como punto de "colección" para enfatizar la ligadura de los arcos que compongan la transición.

Una transición que tenga más de un destino se denomina "transición paralela", si tiene más de una fuente se denomina "transición de sincronización". Uno o más nodos que constituyan colectivamente el estado inicial q_0 se denotan en las representaciones de un AFP con flechas gruesas. Los nodos finales se representan dentro de doble círculo.

Una transición entre nodos en γ es posible o "disponible" si todos sus nodos origen están activos y el símbolo input actual coincide con la etiqueta de la transición. En cada paso en la ejecución de un AFP, una transición disponible es elegida no deterministicamente y ejecutada, llegándose a un nuevo estado desde el estado actual. Nótese que esta elección no determinista posiblemente tenga dos acepciones: un nodo simple puede tener varias transiciones de salida con la misma etiqueta, o dos o más nodos activos pueden tener transiciones con la misma etiqueta. Cuando una transición se ejecuta, todos los nodos origen se vuelven inactivos, y entonces todos los nodos destino se vuelven activos. Esto consume el símbolo de entrada. No existe la noción de que un nodo esté doblemente activo aunque el destino de una transición sea un nodo que ya esté activo.

Si una transición se etiqueta con λ, indica que esa transición está disponible cuando todos sus nodos origen estén activos y no consume ningún símbolo de entrada cuando se ejecuta.

La manera más intuitiva de entender una ejecución AFP es en términos de una red de Petri. Un AFP tiene básicamente la misma estructura que una red de Petri donde el conjunto de nodos N representa los lugares de la red de Petri, los arcos ligados del AFP en γ representan las transiciones de la red de Petri, y el estado inicial q_0 es el marcado inicial de la red de Petri, con un token por lugar activo.

La ejecución de un AFP es como la ejecución de una red de Petri, excepto que en todos los lugares la cantidad de tokens se normaliza a un máximo de 1 después del disparo de cada transición. Por esto, la red jamás tendrá más de un token en cada lugar. La distribución de tokens es parte de la información del estado en una red de Petri normal y es posible acumular varios tokens en los lugares. Una red de Petri clásica tiene un conjunto de estados potencialmente infinito y, sin embargo, las redes

representadas por AFP's tienen limitado el número total de tokens a 1 en cada nodo del grafo por esto, los AFP's son autómatas de estados finitos.

Ejemplos de Autómata Finito Paralelo

Para ilustrar la representación y ejecución de un AFP se presenta la Figura 2.6. En el AFP los nodos 4 y 6 son nodos finales y el nodo 1 es el único nodo de inicio. El lenguaje L aceptado por este autómata es: $a^*b||(c^+d^+)^+$ donde el operador $||$ especifica el entrelazado de los lenguajes que están operando. El autómata finito determinista mostrado es equivalente al AFP en el sentido de que acepta el mismo lenguaje L y es minimal para L.

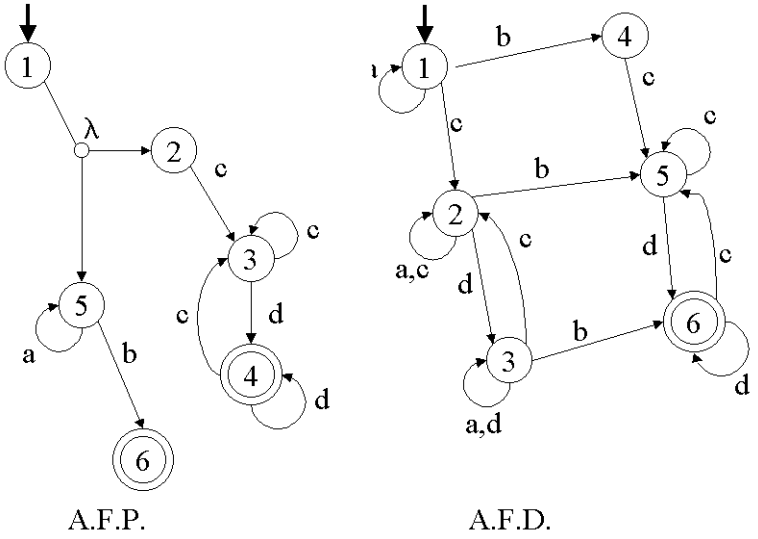


Figura 2.6: Ejemplo de AFP y su Autómata Finito Determinista Minimal equivalente

La secuencia de ejecución obtenida cuando se acepta la palabra "accbd$" (donde $ es el marcador de final) es como sigue:

Nodos Activos	Símbolo de entrada	Acción
{1}	λ	control de 1 a {2,5}
{2,5}	a	control de 5 a 5
{2,5}	c	control de 2 a 3
{3,5}	c	control de 3 a 3
{3,5}	b	control de 5 a 6
{3,6}	d	control de 3 a 4
{4,6}	$	aceptado

En contraste con esta ejecución acabada con éxito en el tratamiento de la cadena

de símbolos de entrada, la siguiente ejecución desemboca en un error al tratar la cadena “caba$”

Nodos Activos	Símbolo de entrada	Acción
{1}	λ	control de 1 a {2,5}
{2,5}	c	control de 2 a 3
{3,5}	a	control de 5 a 5
{3,5}	b	control de 5 a 6
{3,5}	a	error (no hay movimientos posibles

Para un mejor entendimiento del modelo, mostramos un segundo ejemplo en la Figura 2.7 que reconoce el lenguaje: $(a^*b||c^*)m(ac^*||(cd)^*||b^*a)$ usando dos nodos de entrada (el 1 y el 3) y tres nodos finales (5, 6 y 9).

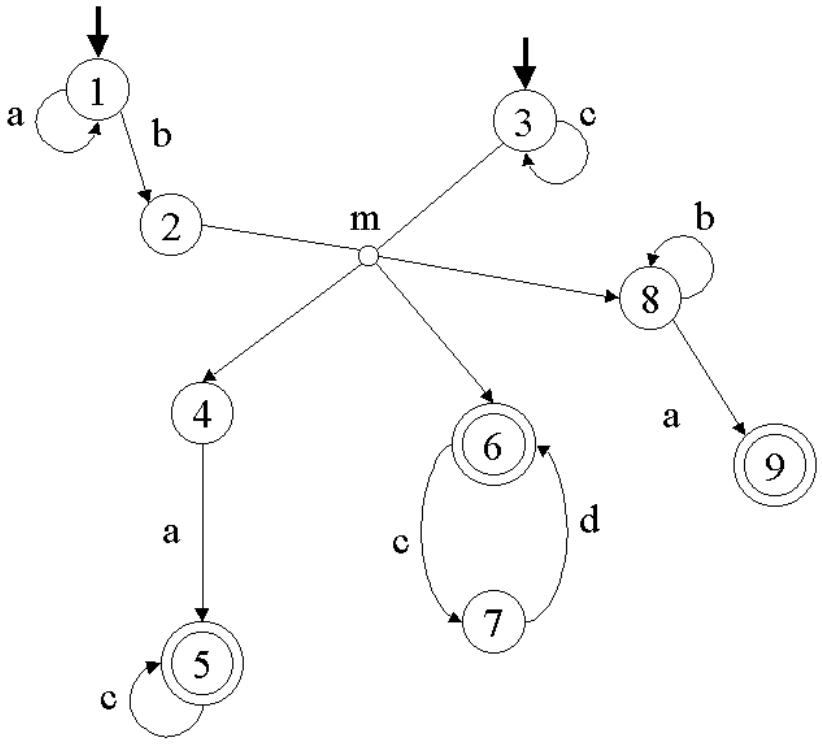


Figura 2.7: Otro ejemplo de Autómata Finito Paralelo

Una ejecución de este autómata finito paralelo aceptando la cadena ”acbmaba$” sería como sigue:

Nodos Activos	Símbolo de entrada	Acción
{1,3}	*a*	control de 1 a 1}
{1,3}	c	control de 3 a 3
{1,3}	b	control de 1 a 2
{2,3}	m	control de {2,3} a {4,6,8}
{4,6,8}	a	control de 4 a 5
{5,6,8}	b	control de 8 a 8
{5,6,8}	a	control de 8 a 9
{5,6,9}	$	aceptado

Nótese que cuando el control está en los nodos {4,6,8} y el simbolo de entrada es "a" el autómata puede elegir entre mover el control alcanzando a los nodos 4,6,9, pero dada la palabra del ejemplo se podría entonces haber alcanzado un estado de no aceptación. Por esto, el comportamiento de los AFP es de alguna manera similar a los autómatas no deterministas. Por el hecho de incorporar directamente el entrelazado, la notación de los AFP es más conveniente para expresar actividades paralelas cooperativas.

Nótese también, que el movimiento desde los nodos activos {2,3} a los nodos activos {4,6,8} con un símbolo de entrada "m" es un movimiento paralelo sincronizado. Como las transiciones que parten de los nodos 2 y 3 están enlazadas juntas el control debe ser en ambos nodos concurrentemente para poder sincronizarse y que el movimiento ocurra.

2.4.2. El modelo de los Autómatas Team

Los Autómatas Team [39, 12] son una propuesta de entorno formal para el desarrollo de sistemas *groupware*. Se definen en términos de autómatas componentes junto con un mecanismo de interconexión basado en las acciones compartidas, es decir, en las sincronizaciones. Los autómatas componentes pueden combinarse de una manera más o menos restrictiva, dependiendo de las acciones que comparten y el modo en el que las comparten.

Este formalismo puede ser usado como ayuda para representar *CSCW* [31] (*Computer Supported Cooperative Work*) que es el mecanismo para comprender mediante una ayuda por computador para el trabajo cooperativo entre usuarios. El *CSCW* es por su naturaleza obvia, una tecnología basada habitualmente en sistemas software multiusuario (los sistemas *groupware*).

Con el incremento de la complejidad de los sistemas *groupware* las abstracciones tienden a hacerse especialmente útiles, es por ello que es necesaria una terminologa precisa y consistente para definir este tipo de sistemas y he aquí, donde el uso de un formalismo adecuado para *CSCW* se vuelve crucial. Además, también es preciso un marco riguroso para describir, comparar y contrastar sistemas *groupware*.

La propuesta inicial de los Autómatas Team presentada en [39] y en [12] fue desarrollada de manera explcíta para el análisis del fenómeno *CSCW* y de los sistemas *groupware*. La claridad del modelo permite capturar de forma explícita las nociones relativas a la colaboración y la coordinación en sistemas distribuidos. Los Autómatas Team constan de una especificación abstracta de los componentes de un sistema

y permiten la descripción de diferentes mecanismos de interconexión basados en el concepto de "acción compartida".

Gracias a la definición formal de los autómatas, los resultados y metodologías de la teoría de autómatas son aplicables en este modelo. Evidentemente este modelo no es útil para representar situaciones no regladas de los modelos groupware, como puedan ser las relaciones sociales u otros aspectos no automatizables. Sin embargo, sí ha mostrado ser adecuado para representarlos en muchas áreas. Por ejemplo, un amplio espectro desde componentes hardware hasta protocolos de interacción entre grupos de agentes pueden ser modelizados con los Autómatas Team.

Los autómatas Team son una extensión de los autómatas Input/Output [101]. Están estrechamente relacionados con los Sistemas Concurrentes controlados por Vectores [87], las redes de Petri y otros modelos de sistemas concurrentes y colaborativos.

Definición de Autómatas Team

Para describir el modelo de los Autómatas Team necesitamos algunas definiciones previas. Sea $\mathcal{I}$ un conjunto de enteros positivos usado para indexar a los autómatas componente dados por $\mathcal{I} = \{i_1, i_2, ...\}$ con $i_j < i_l$ si $1 \leq j \leq l$. Con el conjunto V_i, $i \in \mathcal{I}$, denotamos con $\prod_{i\in\mathcal{I}} V_i$ el producto cartesiano $\{(v_{i_1}, v_{i_2}, ...)|v_{i_j} \in V_{i_j}$, para todo $j \geq 1\}$ si $v_i \in V_i$, para todo $i \in \mathcal{I}$, entonces $\prod_{i\in\mathcal{I}} v_i$ denota al elemento $(v_{i_1}, v_{i_2}, ...)$ de $\prod V_i$. Si $\mathcal{I} = \emptyset$, entonces $\prod_{i\in\mathcal{I}} V_i = \emptyset$. Además de la notación prefija $\prod_{i\in\mathcal{I}} V_i$ o $\prod_{i\in\mathcal{I}} v_i$ para un producto cartesiano se usa también la notación infija $V_{i_1} \times V_{i_2} \times ...$ or $v_{i_1} \times v_{i_2} \times ...$, respectivamente. Sea$A \subseteq \prod_{i\in\mathcal{I}} V_i$ entonces para $j \in \mathcal{I}$, $proj_j : A \to V_j$ se define como $proj_j(a_{i_1}, a_{i_2}, ...) = a_j$, para $J \in \mathcal{I}$, $proj_{Jj} : A \to \prod_{i\in\mathcal{J}} V_i$ se define como $proj_J(a) = \prod_{i\in\mathcal{J}} proj_j(a)$. se usa $proj_i^{[2]}$ $proj_J^{[2]}$ como notación más breve para proyecciones dobles de este modo $proj_i^{[2]}(a, b) = (proj_i(a), proj_i(b))$ y $proj_J^{[2]}(a, b) = (proj_J(a), proj_J(b))$.

Definición 24 (Sistema de transición etiquetado e inicializad (Ilts) [12]) *Un Ilts es una construcción $A = (Q, \Sigma, \delta, I)$, donde Q es su conjunto de estados, posiblemente infinito, Σ su alfabeto de acciones, tal que $Q \cap \Sigma = \emptyset$, $\delta \subseteq Q \times \Sigma \times Q$ su conjunto de transiciones etiquetadas e $I \subseteq Q$ su conjunto inicial de estados.*

Definición 25 (Autómata componente [12]) *Un autómata componente es una constucción $C = (Q, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, I)$ tal que $(Q, \Sigma_{inp} \cup \Sigma_{out} \cup \Sigma_{int}, \delta, I)$ es un Ilts, llamado el Ilts subyacente de C y $\Sigma_{inp}, \Sigma_{out}$ y Σ_{int} son alfabetos disjuntos mutuamente llamados alfabetos* input, output *e* internal *de C, respectivamente.*

Si tenemos una colección de autómatas componente $S = (C_i|_i \in \mathcal{I})$ donde $\mathcal{I}$ es un conjunto de enteros positivos usados para indexar los autómatas componente involucrados, entonces si todas las acciones internas están asociadas únicamente a una componente entonces decimos que S es un sistema componible.

Definición 26 (Espacio completo de transición [12]) *Sea $a \in \cup_{i\in\mathcal{I}}\Sigma_i$. El espacio completo de transicion de a en S está denotado por $\Delta_a(S)$ y se define como $\Delta_a(S) = \{(q, q') \in \prod_{i\in\mathcal{I}} Q_i \times \prod_{i\in\mathcal{I}} Q_i|\exists j \in \mathcal{I} : proj_j^{[2]}(q, q') \in \delta_{j,a}$ y $(\forall i \in \mathcal{I} : proj_j^{[2]}(q, q') \in \delta_{i,a}]$ o $proj_j(q) = proj_i(q')$.*

En otras palabras, el espacio completo de transición de a en S consiste en todas las combinaciones posibles de a-transiciones de componentes de S, con las componentes no participantes permaneciendo invariantes.

Definición 27 (Autómata Team [12]) *Un autómata Team sobre S es una construcción $T = (\prod_{i\in\mathcal{I}} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, \prod_{i\in\mathcal{I}} I_i)$ tal que $\Sigma_{int} = \cup_{i\in\mathcal{I}}\Sigma_{i,int}$, $\Sigma_{out} = \cup_{i\in\mathcal{I}}\Sigma_{i,out}$ y $\Sigma_{imp} = (\cup_{i\in\mathcal{I}}\Sigma_{i,imp})\backslash\Sigma_{out}$ y $\delta \subseteq \prod_{i\in\mathcal{I}} Q_i \times\Sigma\times\prod_{i\in\mathcal{I}} Q_i$ donde $\Sigma = \Sigma_{inp}\cup\Sigma_{out}\cup\Sigma_{int}$ es tal que para todo $a\in\Sigma$ $\delta_a \subseteq \Delta_a(S)$, y además $\delta_a = \Delta_a(S)$ si $a\in\Sigma_{int}$.*

El control espacial en Autómatas Team

Una componente vital de cualquier sistema groupware es la seguridad y el control de acceso a la información. Por ejemplo, en los sistemas de archivos electrónicos típicos los derechos de acceso, como el permiso de escritura o el de lectura, están asignados a usuarios con normas básicas como "necesita saber", las reglas de pertenencia o las listas ad-hoc de accesibilidades.

En los sistemas groupware es usual que se necesite un método de permisos más refinado. Un ejemplo de ello es el acceso simultáneo de varios usuarios a un documento que estén editando, redactando o actualizando conjuntamente de manera asíncrona en tiempo real.

Los mecanismos de control espacial presentados en la metáfora del control espacial de acceso [24, 26] que está basada en la metáfora de realidad virtual de lugares y espacios [25] se pueden representar de manera eficaz con los autómatas Team [13]. Cada *espacio* se representa como un autómata componente y los cambios dinámicos en el acceso se representan añadiendo acciones externas cuando los accesos a recursos en un *espacio* se pueden representar como acciones internas.

El siguiente ejemplo muestra cómo los Autómatas Team pueden ser usados para describir formalmente algunas claves del control de acceso.

En la literatura acerca de la seguridad la autentificación está relacionada con la verificación y es la comprobación de que el usuario sea realmente la persona que representa mientras que la autorización se relaciona con la validación y consiste en comprobar que el usuario tenga acceso al recurso dado. Este ejemplo trata de autorización, así que se asume que cuando el usuario está en el sistema ha habido una autentificación de su personalidad. Entonces, cuando el usuario trate de leer o escribir en un fichero sólo se realizará la prueba de autorización y, por tanto, si se permite o no dicho acceso al usuario.

Considérese un fichero F, que contiene un archivo o documento almacenado de manera electrónica en un sistema de archivos típico. Considérese también a un usuario cuyos permisos de acceso a ese archivo pueden ser revocados o garantizados. El sistema de archivos mantiene información de qué usuarios tienen qué derechos de acceso a F. Partimos de tres posibles tipos de accesos: acceso nulo, (que implica que el usuario no tiene permiso de lectura ni escritura sobre F), acceso de lectura (que implica que el usuario no puede modificar ni escribir en el archivo) y acceso completo (que implica que el usuario tiene la capacidad de escribir, leer, modificar, etc. al fichero F).

Además, se asume la existencia de un asistente administrador del sistema, que puede cambiar los derechos de los usuarios. Por lo tanto, este asistente tiene el derecho de conceder y revocar los permisos de acceso del usuario a F. Las acciones $m(r)$, $\underline{m}(r)$,

$m(w)$ y $\underline{m}(w)$ modelan las operaciones de "conceder permiso de lectura", "revocar permiso de lectura", "conceder permiso de escritura" y "revocar permiso de escritura", respectivamente. Los derechos de concesión y revocación de permisos son también permisos como los anteriores, pero no están aplicados directamente sobre F. Estos derechos se consideran de hecho meta-operaciones.

Finalmente, se asume que el derecho de concesión y revocación de permisos es también susceptible de concesión o revocación. Por lo tanto, si hay un administrador de sistema que puede permitir (o no) al asistente del administrador de sistema que conceda o revoque permisos, entonces este administrador tiene meta-meta-permisos. El administrador tendrá por tanto el meta-meta-permiso de conceder o revocar los meta-permisos del asistente sobre conceder o revocar permisos de acceso a F. Una acción típica del administrador podría ser $\underline{m}^2(w)$, que en este caso revocaría el permiso del asistente a conceder o eliminar permisos de escritura sobre F. Estas situaciones tan complicadas (recursivas) pueden aparecer de esta manera dependiendo de la política de control de accesos elegida. El ejemplo propuesto muestra cómo estas situaciones pueden ser presentadas concisamente y sin ambigüedades usando el modelo de los autómatas Team.

La Figura 2.8 muestra al Autómata Componente M^0 que modela los tres niveles A, B y C, que corresponden a los permisos nulo, de lectura y de escritura, respectivamente. El estatus del usuario determina directamente el nivel sobre el que puede operar y las acciones de revocación o concesión de derechos se identifican con los cambios de estado (cambios de nivel).

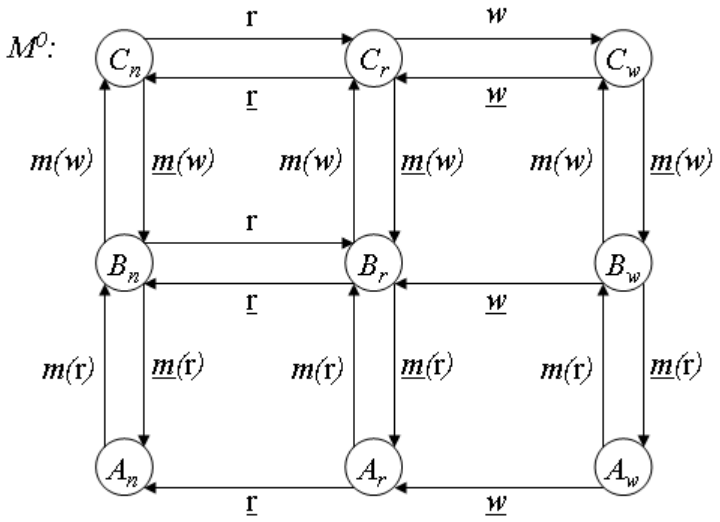


Figura 2.8: Autómata Componente M_0

En M^0 el usuario se mueve en dos dimensiones. Verticalmente entre los niveles A, B, y C, representando los cambios dinámicos en sus permisos de acceso sobre F, y horizontalmente entre los estados n, r y w (***n**ull*, ***r**eading* y ***w**riting*) que indica sus actividades actuales sobre el fichero F. Nótese que, por ejemplo, el estado B_w indica

que el usuario está escribiendo cuando en realidad tiene permiso de lectura pero no de escritura. Este estado sólo es accesible desde el estado C_w con una acción $\underline{m}(w)$ o desde A_w con una acción $m(r)$. Por lo tanto, este estado B_w sólo es accesible si los permisos del usuario cambiasen cuando está escribiendo.

Como este cambio de estado está impuesto al usuario por el asistente del administrador del sistema las acciones de concesión o denegación de derechos de acceso son claramente acciones pasivas desde el punto de vista del usuario. Por esta razón se ha elegido que este tipo de acciones se consideren acciones *input* mientras que las acciones de lectura y escritura son acciones *output*. Nótese que M^0 no es *input-enabled*, lo que implica inmediatamente que no es un autómata I/O.

Incluyendo acciones del tipo comentado como $(C_w, \underline{m}(w), B_w)$, se está implícitamente asumiendo un modelo de *revocación pospuesta*. Este tipo de revocación (opuesto a la *revocación inmediata*) implica que el usuario puede continuar su actividad aunque sus derechos hayan sido revocados. Podrá hacerlo hasta que intente reiniciar su actividad, en ese momento la comprobación de su autorización decidirá si puede o no reiniciar esa actividad.

Si en lugar de esa acción se permitiesen acciones del tipo $(C_w, \underline{m}(w), B_r)$ entonces se estaría asumiendo el uso de *revocación inmediata*. Esto significaría que si el usuario estuviese leyendo cuando su permiso de lectura o escritura es revocado, cesaría inmediatamente de leer o escribir y pasaría al estado siguiente. Esta práctica puede, en algunas aplicaciones, ser especialmente desaconsejable o causar problemas.

La noción de meta se extiende claramente a capas arbitrarias. Ahora aparece, como efecto de las revocaciones, una cuestión sumamente interesante: ¿Puede la revocación de un meta-permiso revocar a su vez los permisos que se han pasado a los otros? Esto introduce los conceptos de *revocaciones superficiales* y *revocaciones profundas*.

Las *revocaciones superficiales* implican que una acción de revocar un meta-permiso no implica la revocación de los diferentes permisos pasados a otros, por contra las *revocaciones profundas* implican la revocación conjunta de todos los derechos previamente pasados. Ahora, recuérdese la acción $\underline{m}^2(w)$, del administrador del sistema. Esta acción implicaría revocar el derecho del asistente a conceder o revocar permisos de escritura y, por tanto, implicaría que o bien sería él solo quien perdiera derechos (*revocacion superficial*) o bien todos los usuarios perderían a su vez el derecho de lectura (*revocacion profunda*). El modelo de los Autómatas Team puede mostrar tanto las *revocaciones superficiales*, como las *revocaciones profundas* o un sistema de *revocaciones híbridas*. Es evidente que el modelo más sencillo es el de *revocaciones superficiales* mientras que el modelo de *revocaciones profundas* es un complicado reto para su modelado e implementación [35]. A continuación se muestra cómo modelar el planteamiento de las *revocaciones profundas* usando los Autómatas Team.

La Figura 2.9 muestra un autómata componente capturando una capa (k) de un sistema de especificación de meta-accesos multicapa del ejemplo propuesto de acceso de lectura y escritura al fichero F. Ya se ha visto la capa 0 a través del autómata componente M^0 en la Figura 2.8. Para cada valor de $k \geq 1$ existe el correspondiente autómata componente que está relacionado directamente con dicha capa k. Para cada autómata componente M^k las acciones horizontales $m^k(r)$, $\underline{m}^k(r)$, $m^k(w)$ y $\underline{m}^k(w)$, son aciones *output* y las acciones verticales $m^{k+1}(r)$, $\underline{m}^{k+1}(r)$, $m^{k+1}(w)$ y $\underline{m}^{k+1}(w)$ son acciones *input*. Nótese que esto significa que para cada $k > 0$ el autómata componente M^k no es un autómata I/O. Es inmediato, partiendo de la definición, que

$\{M^0, M^1, ..., M^n \mid n \geq 1\}$ es un sistema componible. Puesto que cada acción output está asociada únicamente a un autómata componente este es siempre compatible.

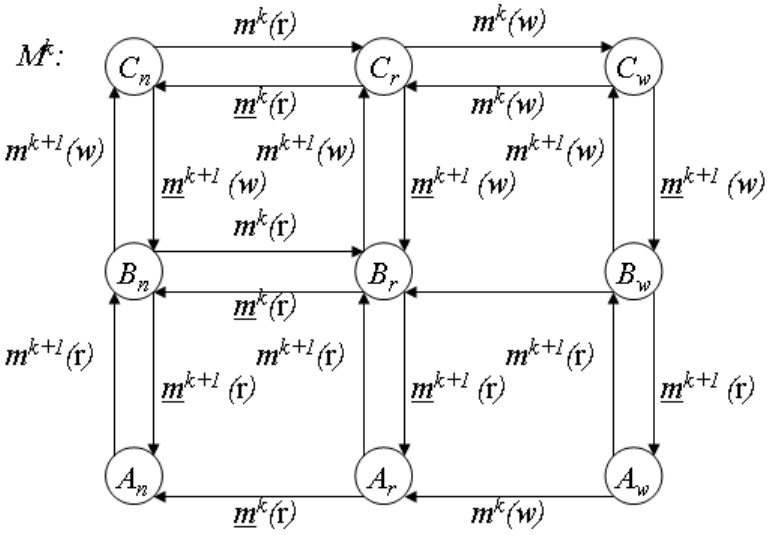


Figura 2.9: Autómata Componente M^k, meta-acceso en la capa k

Finalmente, se identifica para $k = 0$ a r con $m^0(r)$, $\underline{r}$ con $\underline{m}^0(r)$, w con $m^0(w)$, $\underline{w}$ con $\underline{m}^0(w)$ y de manera similar: $m(r) = m^1(r)$, $\underline{m}(r) = \underline{m}^1(r)$, $m(w) = m^1(w)$, $\underline{m}(w) = \underline{m}^1(w)$.

Con esto, se puede definir una estructura multicapa componiendo un autómata Team sobre el sistema componible $M^0, M^1, ..., M^n \mid n \geq 1$. Por la elección de las acciones está claro que sólo autómatas componentes contiguos M^{k-1} y M^k pueden sincronizarse en acciones comunes. Por lo tanto, debido a los resultados de las construcciones iterativas de los autómatas mostrados en [39] se puede especificar el Autómata Team describiendo la interacción de dos autómatas, M^{k-1} y M^k y requiriendo que el team autómata resultante sea un subTeam[39] del autómata final que modela la estructura multicapa deseada.

En la Figura 2.10 se muestra al Autómata Team T^k_{k-1} sobre M^{k-1} y M^k, representando la capa $k-1$ y la capa k de esta estructura multicapa (sólo la parte alcanzable). La relación de transición de este Team T^k_{k-1} se ha elegido modelizando la *revocacion profunda*. En la figura se han añadido superíndices para indicar que el estado pertenece al autómata M^{k-1} o M^k, distinguiento por tanto un estado B_r perteneciente a M^{k-1} de un estado B_r perteneciente a M^k

Un autómata M^k puede revocar derechos de lectura a M^{k-1} en cualquier momento realizando una acción *output* $\underline{m}^k(r)$, y forzando a M^{k-1} a realizar esta acción (esta vez, como accion *input*). Se aplican dos reglas de actividad en este Autómata Team sobre $\{M^0, M^1, ..., M^n \mid n \geq 1\}$:

- Primero, cuando un autómata componente M^k con $1 \leq k \leq n$ realiza una transicion derecha (conceder) o izquierda (revocar) entonces el autómata M^{k-1} debe realizar una transición ascendente (obteniendo derechos) o descendente

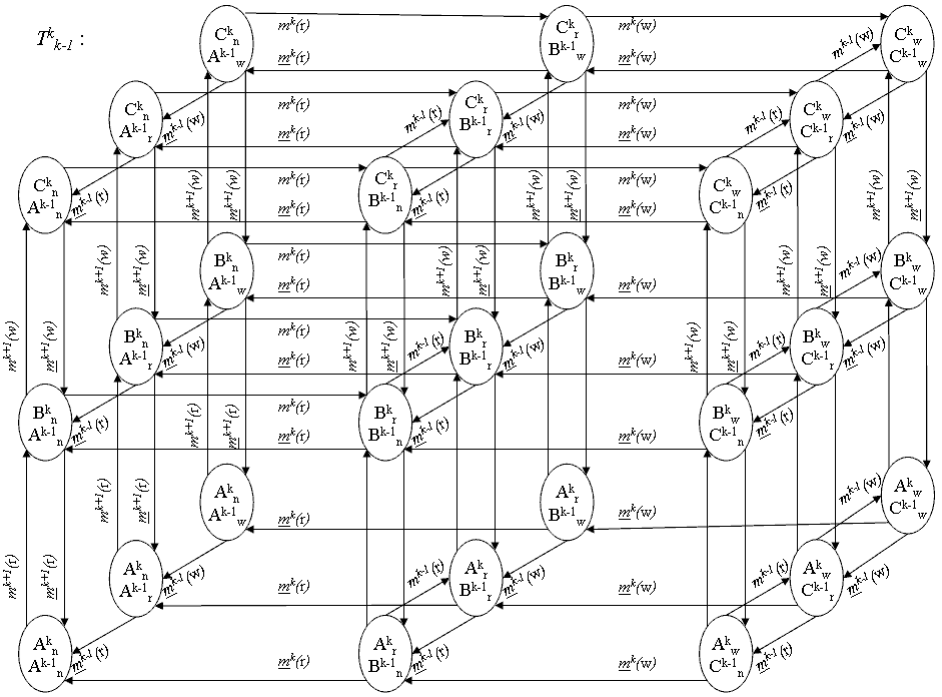


Figura 2.10: Autómata Team T^k_{k-1} sobre M^{k-1} y M^k

(perdiendo derechos). Esto se realiza de la manera siguiente. Cada una de las acciones *output* $m^k(r)$, $\underline{m}^k(r)$,$m^k(w)$, $\underline{m}^k(w)$ de M^k son acciones *input* de M^{k-1}. Se elige la relación de transición del autómata Team T^k_{k-1} sobre M^{k-1} y M^k tal que esas acciones *output* son sincronizaciones *master-slave* en T^k_{k-1}. Notese que el resto de acciones de T^k_{k-1} son acciones libres.

- Segundo, el autómata componente M^{k-1} se ve forzado a realizar una transición descendente cuando, eventualmente efectue una transición izquierda por la ejecución de una de sus acciones *output*. Todas estas acciones *output* de M^{k-1} están involucradas como acciones *master* en una sincronización *master-slave* con las acciones input del mismo nombre del autómata componente M^{k-2} (como *slaves*). Esta transición izquierda forzará de nuevo una transición descendente de M^{k-2} y sucesivamente hasta llegar a M^0 o la capa 0.

2.4.3. Redes de Petri de Alto Nivel: Redes de Referencias

Una red de Petri de Alto Nivel (RPAN) [72] es una red de Petri donde se añaden tipos a los que pertenecen los tokens, lugares y transiciones, de manera que los disparos de las transiciones se producen siempre que coincidan en número y tipo. Además para las redes de referencias, se añaden instancias de redes, y se usan canales síncronos para permitir interactuar las diferentes redes instancia y dos tipos de tokens, las tuplas de valores y los indicadores de instancias.

Definición 28 (Red de Petri de alto nivel (RPAN)) *Una red de Petri de Alto Nivel es una estructura* $RPAN = (P, T, D, Type, Pre, Post, M_0)$ *donde:*

- P *es un conjunto finito de elementos llamados "lugares".*
- T *es un conjunto finito de elementos llamados "transiciones" tal que* $T \cap P = \emptyset$
- D *es un conjunto finito no vacío de dominios no vacíos llamados "tipos"*
- $Type : P \cup T \rightarrow D$ *es una funcion usada para asignar tipos a lugares y determinan los modos de transiciones.*
- $Pre, Post: TRANS \rightarrow \mu PLACE$ *son los marcados antes y después donde:*
  - $TRANS = \{(t, m) \mid t \in T, m \in Type(t)\}$
  - $PLACE = \{(p, g) \mid p \in P, g \in Type(p)\}$
- $M_0 \in \mu PLACE$ *es un multiconjunto llamado el marcado inicial de la red.*

y donde además, $\mu PLACE$ *es el conjunto de multiconjuntos sobre el conjunto* $PLACE$*, y es llamado también el conjunto de marcados de la red, es decir:* $M \in \mu PLACE$ *es un marcado.*

Definición 29 (Modo de transición habilitado) *Se dice que un modo de transición* $tr \in TRANS$ *está habilitado en un marcado* M *sii* $Pre(tr) \leq M$

Definición 30 (multiconjunto finito de modos de transición habilitado) *Se dice que un multiconjunto finito de modos de transición* $T_\mu \in \mu TRANS$ *está habilitado sii:* $Pre(T_\mu) \leq M$ *donde la extensión lineal de de Pre viene dada por* $Pre(T_\mu) = \sum_{tr \in TRANS} T_\mu(tr) Pre(tr)$

Definición 31 *Todos los modos de transición en* T_μ *estan* habilitados concurrentemente *si* T_μ *está habilitado, es decir, si hay suficientes tokens en los lugares input para satisfacer la combinación lineal de los marcados "pre" para cada modo de transición en* T_μ

Dado un multiconjunto finito de modos de transición T_μ habilitado en un marcado M entonces se puede producir un paso resultando un nuevo marcado M' dado por: $M' = M - Pre(T_\mu) + Post(T_\mu)$ donde se usa la extensión lineal de $Post$ y se denota por $M\ [\ T_\mu > M'$ o por $M \stackrel{T_\mu}{\rightarrow} M'$

Notese que un paso puede producirse por la ocurrencia tanto de un modo de transición sólo o por un cierto número de modos de transición habilitados concurrentemente.

El formalismo de las redes de referencias [103, 92] y su simulador *Renew* [94] (que en lo sucesivo se usarán indistintamente) define una clase especial de redes RPAN, que usa Java como lenguaje de inscripciones y extiende las redes de Petri con redes-instancia, referencias de redes y sincronización de transiciones dinámicas a través de canales síncronos. Una Red de Referencias consiste en *lugares* (gráficamente representado por elipses), transiciones (cajas) y arcos (lineas con flechas). Existen en esencia

tres tipos de arcos. Primero, arcos input y output ordinarios, que vienen señalados con una flecha simple. Segundo, hay arcos reservados que son simplemente una notación abreviada para un arco input y uno output. En tercer lugar, hay arcos de prueba, que no tienen flechas. Un token sencillo puede ser accedido (comprobado) por varios arcos de test a la vez.

Cada lugar o transición puede tener asignado un nombre, mostrado en el simulador con letras en negrita. Cada elemento de la red puede cargar con inscripciones semánticas. Los lugares tienen un numero arbitrario de expresiones de inicialización. Las expresiones de inicialización son evaluadas y los valores resultantes sirven como marcado inicial de los lugares. Por defecto, un lugar está inicialmente sin marcar. Los arcos pueden tener una inscripción de arco. Cuando una transición se dispara sus inscripciones de arcos son evaluadas y los tokens se mueven de acuerdo al resultado. Las transiciones pueden ser equipadas con una variedad de inscripciones. Las *inscripciones de expresión* son expresiones ordinarias que son evaluadas en cuanto el simulador busca una ligadura en la transición. Las inscripciones en Java son expresiones. El resultado de esta evaluación se descarta pero en tales expresiones se puede usar el operador de igualdad (=) para influenciar la ligadura de variables que se usen en algún sitio. Las *inscripciones de guarda* son expresiones que están prefijadas con la palabra reservada *guard*. Una transición se podrá disparar únicamente si todas sus inscripciones de guarda se evalúan a cierto. Con estas añadiduras se cubre el formalismo básico de las redes coloreadas [82].

Hay dos clases de tokens: tokens *valor* y tokens *referencia*. Por defecto, un arco puede transportar un *black token* denotado por []. Pero si se añade una inscripción de arco a un arco, la inscripción se evaluará y el resultado determinará qué tipo de token se moverá a través de él.

Las variables están acotadas a un único valor durante el disparo de una transición, sin embargo, durante el siguiente disparo de la misma transición las variables pueden estar acotadas a valores diferentes. Esto es similar al modo en que se usan las variables en programación lógica (por ejemplo en Prolog).

El lenguaje de las inscripciones de las redes de referencia se ha extendido para incluir *tuplas*. Una tupla se denota por una lista de expresiones separadas por comas encerradas entre corchetes. Las tuplas son muy útiles para almacenar un grupo completo de valores relacionados dentro de un único token y, por tanto, en un único lugar.

Adicionalmente hay inscripciones de creación que se relacionan con la creación de instancias de red y con los *canales síncronos*. Una red se especifica como una estructura estática. Sin embargo, una instancia de una red tiene un marcado que puede cambiar con el tiempo. Cuando una simulación comienza se crea una nueva instancia de red. Las nuevas instancias son creadas por transiciones que llevan *inscripciones de creación* que consisten en un nombre de variable, dos puntos (:) la palabra reservada **new** y el nombre de la red a instanciar.

Cuando una instancia deja de ser referenciada en el simulador y ninguna de sus transiciones está habilitada es liberada por el mecanismo de "garbage collection" de Java.

Las instancias necesitan algún tipo de comunicación, para ello se escogieron los *canales síncronos* propuestos para las redes de Petri coloreadas en [30] que sincronizan dos transiciones y las disparan al mismo tiempo. Ambas transiciones deben aceptar los

nombres de los canales y el conjunto de parámetros antes de poder sincronizarse. En las Redes de Referencias se generaliza este mecanismo permitiendo que se sincronicen transiciones de diferentes instancias con uno o varios canales síncronos. Para esto en la red que inicia la sincronización se utiliza un formato como sigue: primero una variable que indica la instancia subordinada con la que sincronizará, luego dos puntos (:) seguidos por el nombre del canal y por último los parámetros (opcionales). En la red instanciada se recibe el canal pero sin el nombre de variable previo a los dos puntos.

Capítulo 3

Los Autómatas Cooperativos Extendidos

En este capítulo se presenta un formalismo similar a las redes de Petri [115, 109] para la modelización de la coordinación de agentes en un sistema distribuido denominado *Autómatas Cooperativos* [5] así como su extensión y simplificación y que conforma la propuesta de esta tesis doctoral.

3.1. Autómatas Cooperativos

Un sistema puede verse como una red de Petri de alto nivel en la que los tokens son elementos activos que consisten en un autómata junto con una porción de memoria privada. Las transiciones de la red son principalmente vectores de sincronización que dan todas las posibles interacciones entre objetos dinámicos. Se asume, básicamente, que los objetos pueden sincronizarse de acuerdo con su comportamiento (el servicio que prestan o por el contrario el servicio que solicitan) indiferentemente de su estado interno. Por ejemplo, si un proceso quiere imprimir un archivo, no tiene que preocuparse de qué impresora puede realizar el trabajo, cualquier objeto puede ser reemplazado por otro que ofrezca el mismo servicio de forma segura, incluso si ambos son muy diferentes respecto a otros servicios. Una sincronización puede requerir la creación de nuevos objetos (tokens) y puede también requerir la destrucción de otros.

El estudio de las diversas propiedades y caractersticas de los *Autómatas Cooperativos* así como la automatización, en la medida de lo posible, del proceso de creación, definición y simulación de un sistema distribuido será de vital ayuda para plantear este formalismo como una herramienta eficaz y con expresividad suficiente como para detectar a priori los posibles problemas planteados por una solución a un entorno dado y para, en el mejor de los casos, la obtención de un sistema de prototipos. Estos deben permitir, a su vez, la obtención de un código fuente que sirva de base para la implementación de un sistema real con todas las garantías que ofrece una verificación formal del sistema.

Para ilustrar la representación de un sistema bajo la perspectiva de los autómatas cooperativos vamos a ver un sencillo ejemplo del modelo “productor-consumidor”. El

sistema se compondrá de dos autómatas como los de la Figura 3.1 y tres reglas básicas (además de las de creación y destrucción en su caso)

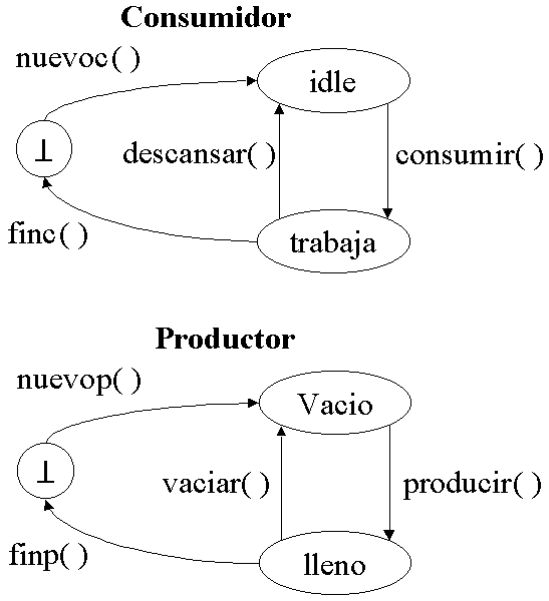


Figura 3.1: Autómatas Productor y Consumidor

Las reglas serán las siguientes:

```
name: sincronización Prod-Cons
 synch:   prod.vaciar() + cons.consumir
 guard:   prod.tareap == cons.tareac
 update:  null

name: Producción
 synch:   prod.producir
 guard:   null
 update:  null

name: Descanso
 synch:   cons.descansar
 guard:   null
 update:  null
```

En este ejemplo hemos añadido un atributo “tareac” y “tareap” a los objetos de tipo consumidor y productor, respectivamente, que deben ser iguales para poder sincronizar siempre el mismo productor con el mismo consumidor.

Por ejemplo, para tener dos consumidores y un único productor que sirva a ambos, tendríamos la siguiente regla de inicialización:

```
name: inicializacion_2c_1p
 synch:   prod.newp() + cons1.newp + cons2.newp
 guard:   null
 update:  prod.tareap = new
          cons1.tareap = prod.tareap
          cons2.tareap = prod.tareap
```

De esta manera los dos consumidores competirían para poder usar el recurso productor, y si volviéramos a invocar a la regla tendríamos dos nuevos consumidores y su productor asociado.

3.1.1. El modelo básico

El modelo básico de los *Autómatas Cooperativos* consiste en coordinación pura. Los tokens pueden ser creados o destruidos a través de sincronizaciones de sus eventos, pero no pueden intercambiar información, puesto que no tienen memoria privada. Como instancias típicas de este modelo tenemos la sincronización de autómatas de Arnold y Nivat [4], las redes de transición de estados y el formalismo Gamma [8, 9, 10].

Este modelo básico puede ser extendido sin embargo para tratar *Workflow Management Systems* [38] que describen el problema de instancias en un sistema distribuido. Por ejemplo, una instancia puede estar involucrada con la participación de un investigador en una conferencia; entonces se crean varias tareas, por ejemplo: la inscripción, la reserva del vuelo, del hotel, etc. Estas tareas requieren la coordinación de varios sistemas de computadores que pertenecen respectivamente a la institución del investigador, la agencia de viajes, la organizacion de la conferencia, etc. A cada instancia se le debe asignar un único indentificador que se transmite a cada una de las tareas que se generan para que el conjunto del sistema distribuido pueda reconocer las tareas involucradas con el tratamiento de una instancia dada. Se modelan las tareas con tokens que, por lo tanto, contienen un identificador en su memoria privada. En semejante sistema, sin embargo, todavía no hay flujo de información entre tokens diferentes, debido al hecho de especificar un vector de sincronización que nos permite afirmar tan solo que ciertos componentes pueden tener el mismo identificador.

Avanzando un paso más, podemos considerar aplicaciones *groupware* [33] que involucran sistemas distribuidos de agentes que cooperan para resolver una tarea global y cuya estructura puede cambiar dinámicamente. En este caso, un token puede contener también referencias a otros tokens en su memoria privada. Dado que en estas aplicaciones los tokens incorporan tanto estructuras de datos como especificaciones de comportamiento, pueden ser considerados como objetos en el sentido de la terminología de los lenguajes orientados al objeto [124].

Si la memoria local se presenta como una lista de atributos-valores parece haber una delimitación clara de la expresividad en términos del flujo de información que puede ser medida por el número de atributos. En este capítulo, mostramos que la expresividad completa se alcanza tan pronto se admitan dos atributos. La jerarquía consiste entonces en tres clases: La clase completa (con dos o más atributos) que se adapta adecuadamente al diseño de aplicaciones *groupware*, la clase intermedia (con un atributo) que se adapta a la necesidad de aplicaciones *workflow* y el modelo básico (sin atributos) que permite la coordinación pura de eventos sin flujo de información.

La clase completa de autómatas cooperativos es suficientemente potente como para

permitir la simulación de máquinas de Turing. No hay, por lo tanto, esperanza alguna de obtener herramientas generales automáticas para verificar propiedades de estos sistemas, sin embargo, se piensa que este modelo general es útil como herramienta de modelado.

Por el contrario, se puede decidir si un sistema tiene un número finito de estados si lo restringimos a la así llamada subclase de autómatas cooperativos *workflow* para la que tan solo se permite un atributo. Esta propiedad, también llamada *boundedness (acotamiento),* está demostrada fundamental para la verificación de sistemas *workflow* [1].

3.1.2. Una aproximación al modelado

Se propone un modelo para describir los sistemas basados en agentes dinámicos utilizando *Autómatas Cooperativos* [5]. Este modelo utiliza agentes para proveer servicios y artefactos para mantener las relaciones entre agentes. Ambos están representados por un autómata de estados finitos y un conjunto finito de atributos que define la pertenencia a un grupo.

El modelado de sistemas utilizando los *Autómatas Cooperativos* generalmente requiere los siguientes pasos:

1. Identificar al autómata en el sistema. Deben ser agentes (como procesos u objetos), o artefactos que sirvan para mantener las relaciones entre autómatas.

2. Identificar los estados de cada autómata y las acciones que conllevan. No es necesario identificar todas las acciones de los autómatas en este punto.

3. Identificar las interacciones entre autómatas. En nuestro modelo, dichas interacciones están representadas como vectores de sincronización. Cuando en una interacción, todos los estados asociados a dicha interacción cambian, en sus respectivos autómatas, lo hacen sincrónicamente. Las interacciones pueden provocar la creación o destrucción de autómatas.

4. Iterar estos pasos si es necesario.

5. Finalmente, es importante validar el modelo cuando está completo. ¿Refleja adecuadamente el modelo al sistema bajo desarrollo? Si es así, entonces ¿Es el sistema lo que se deseaba representar? El modelo puede servir para asegurarse de que las descripciones son correctas.

A continuación se perfila la aproximación al diseño e implementación de sistemas cooperativos, mediante modelado con *Autómatas Cooperativos.*

Cuando se emprende el diseño de un sistema cooperativo, la primera etapa consiste en identificar los recursos a utilizar, las tareas a realizar usando dichos recursos y los agentes del mundo exterior responsables de las tareas. Estos últimos en contraste con los recursos y tareas que habitualmente habitan en el sistema, deben actuar a través de artefactos que proporcionen conexiones para establecer relaciones entre ellos y para controlar las tareas que inician. Si tenemos en mente el diseño y la evaluación temprana de un modelo de un sistema cooperativo, y no la realización completa de los agentes de control del mundo exterior, esta diferencia desaparece dado que tanto

los recursos como las tareas se representan como artefactos en el modelo. Sin embargo, quedan unas leves diferencias entre recursos, tareas y agentes. Los recursos son estáticos (no tienen creación dinámica ni se dividen) y son intercambiables (todos los elementos de una fuente de recursos capaces de realizar el mismo servicio en un momento dado son equivalentes). En contraste con los recursos, las tareas son objeto de creación dinámica y son habitualmente divididas en subtareas que son reensambladas más tarde si es necesario (esto es un aspecto típico de los sistemas de manufactura y de flujos de trabajo). Además una tarea tiene un tiempo de vida y aparece como una colección dinámica de subtareas que pueden involucrar diferentes recursos. Esto es de vital importancia en un sistema cooperativo para anexar información permanente a subtareas, permitiéndoles reconocer su tarea principal. El caso de agentes externos, que pueden también participar y marcharse del sistema en ciertos momentos, es diferente: El problema no es tanto mantener la traza de clusters de agentes, como mantener datos operativos que representen sus relaciones conocidas en el sistema, basadas en que el sistema puede establecer reglas de interacción. Estos datos deben ser proporcionados mediante grafos o hipergrafos (con agentes como nodos), modificados de acuerdo a las reglas específicas de transformación cuando el sistema cambia las relaciones entre agentes (como resultado de alguna interacción).

Autómatas

El papel de un sistema cooperativo soportado por computador(CSCW) es fijar las reglas de interacción y evolución de una colección dinámica de agentes, tareas y recursos, representados por artefactos que los representen en un entorno real. El papel de un modelo es proporcionar un marco para el diseño y alguna ayuda para la validación temprana de dichos sistemas. Para ser útil, un modelo debe proporcionar una representación uniforme de los items (recursos, tareas y agentes) y de las interaciones. Los Grafos o Hipergrafos y sus reescrituras son posibles candidatos, pero es de suponer que los ingenieros preferirán un formalismo más cercano a su práctica habitual. Esta es la razón por la que se propone un modelo basado en autómatas y relaciones. Un autómata finito no es universal desde el punto de vista de la potencia de la computación, pero proporciona un buen punto de partida para el diseño de sistemas cooperativos. Como primera etapa en el diseño, se propone modelar todos los items del sistema (agentes, subtareas y recursos) como autómatas finitos cada uno de los cuales defina los posibles estados y las acciones de un componente particular. Los recursos se representan por autómatas cuyas acciones representan los servicios que ofrecen en cada uno de sus posibles estados. Las tareas o subtareas (habitualmente descritas como diagramas de flujo) se representan por autómatas cuyas acciones están en una correspondencia uno a uno con las cajas de los diagramas de flujo. Los agentes, definidos por roles y escenarios que pueden ser interpretados como palabras sobre algún alfabeto, están representados por autómatas que reconocen estas palabras. Situando un autómata junto con un nuevo estado (muerto o nonato) con transiciones adicionales a, y desde este nuevo estado, se representa la creación o destrucción dinámica de todos los tipos de items. En esta etapa, el sistema debe ser como un almacén de instancias de autómatas autónomos que puede crecer o encogerse dibujando instancias desde el estado nonato o empujando instancias ya existentes al estado muerto. Esto permite fijar la política de coordinacion que es la principal cuestión en el diseño de sistemas

cooperativos soportados por ordenador.

Coordinación

Como preliminar a esta segunda etapa, se propone identificar las acciones coordinadas que deben tener lugar en el sistema sin tener en cuenta las variaciones en el tiempo de las relaciones entre agentes. En el mismo sentido, como los autómatas proporcionan los átomos de la descomposición de estados, se necesitan átomos para la descomposición de comportamientos. El tomar todas las acciones autónomas de un autómata en el almacén como posibles átomos puede no ayudar al entendimiento de las relaciones que se crean y se mantienen entre agentes si no queda claro, desde el principio, que no serán autónomas en el sistema final. Por ejemplo, un recurso no podrá realizar un servicio hasta que sea pedido por una tarea, y una tarea no podrá ser creada hasta que no sea pedida por algún agente. Por lo tanto, se propone enumerar la lista de interacciones atómicas que pueden ser consideradas para un análisis posterior del sistema bajo diseño. Hasta que las relaciones que varíen en el tiempo entre agentes no sean tenidas en cuenta, esta enumeración sólo puede ser basada en los autómatas componentes. Se sugiere abstraer de la información del estado y por lo tanto tomar como átomos los vectores de acciones sincronizadas. Una acción coordinada descrita por un vector de acciones sincronizadas puede tener lugar si existen en el almacén suficientes autómatas con los estados actuales autorizándoles a realizar las acciones respectivas del vector. En este sentido se obtiene una aproximación preliminar al sistema cooperativo, con la ventaja principal de que puede ser experimentado utilizando un simulador. La aproximación será muy inicial mientras se imponga control estático de las interacciones y no se tenga en cuenta las relaciones que varíen en el tiempo entre agentes.

La etapa principal en el diseño es la posterior restricción de las interacciones entre autómatas para hacer cumplir la política de cooperación que es el propósito de los sistemas cooperativos soportados por computador. Esto abarca desde las reglas simples, por ejemplo, asegurarse de que una tarea lanzada por un agente sincroniza con la terminación de ese agente y no de otro, hasta las reglas arbitrariamente complejas que dependan de las relaciones actuales entre agentes, y posiblemente modifique estas relaciones. Idealmente, uno podría pensar que el almacén es un grafo o un hipergrafo cuyos nodos son autómatas componentes, y cuyos arcos o hiperarcos describen sus relaciones. El disparo de los vectores de sincronización debe ser controlado por reglas que impongan restricciones relacionales sobre los autómatas involucrados en una interacción dada, y eso también define las posibles modificaciones del grafo resultante de la interacción. El diseño de la estructura relacional y sus transformaciones es extremadamente complejo. No es realista tratar de resolver todos los problemas al mismo tiempo introduciendo directamente alguna gramática de grafos adecuada. Se propone, por lo tanto, un método interactivo en el que el modelo inicial con vectores de sincronización incontrolados se refine paso a paso incorporando condiciones de la estructura relacional y modificaciones locales. La disponibilidad de un simulador para ver qué es lo que funciona mal en cada paso en los escenarios de prueba es crucial. El simulador debera ser proporcionado con una herramienta de visualización que mostrara los estados actuales de los autómatas componentes y la estructura relacional entre ellos.

Hay serios inconvenientes en la visión ideal presentada hasta ahora. Tratar con grafos e hipergrafos y sus reescrituras no es una práctica usual en ingeniera del software, y es todavía más dificil cuando uno tiene que atender al mismo tiempo el comportamiento local de los nodos (los autómatas componentes). Tratar con la estructura relacional como un dato global no ayuda a producir implementaciones distribuidas de sistemas cooperativos soportados por computador, donde deben ser cortados en trozos. Para aliviar estas dificultades, se sugiere codificar las relaciones con un sistema de atributos enteros anexados al autómata componente. Utilizando estos atributos, una relación ternaria $R(x, y, z)$ entre los autómatas componentes X, Y y Z vistos como nodos de un hipergrafo que puede ser representado añadiendo los atributos $R1$, $R2$ y $R3$ a estos autómatas respectivos e inicializándolos a un valor común. Los valores son significativos dependiendo de una permutación de enteros, por lo tanto, los atributos sólo serán testeados mediante la igualdad. Añadir guardas a los vectores de sincronización, requiriendo valores iguales para atributos especificados en los autómatas respectivos que participen en una acción coordinada, es entonces equivalente a chequear la ocurrencia de un subgrafo especfico conectándolos en la visión ideal. Añadiendo las posibles modificaciones de los valores de los atributos como efectos laterales de los vectores de sincronización permite describir las modificaciones de la estructura relacional.

No hay nada nuevo para los ingenieros, y no hay pérdidas si el simulador se proporciona con una ayuda visual mostrando el grafo implcito reconstruido desde los valores actuales de los atributos de los autómatas componentes. El beneficio principal es permitir ver configuraciones del sistema como un conjunto no estructurado de tokens, donde cada uno representa un autómata componente como una tupla, mostrando el estado actual y el valor actual de sus atributos. Esta visión es de ayuda para la simulación dado que permite dirigirla preseleccionando los tokens de acuerdo a los estados o a su pertenencia a un grupo, donde un grupo es un conjunto de autómatas con un atributo común. Esto es también de ayuda para la implementacion distribuida de sistemas cooperativos.

En relación a esto, es importante que los modelos obtenidos puedan ser compilados para funcionar en plataformas existentes. Este parece ser el caso de los autómatas cooperativos, que pueden probablemente ser adaptados para funcionar en Jini, dado que los vectores de sincronización coinciden con las transacciones Jini [132]. La fase final es obtener un prototipo del sistema, dando lugar a los trozos de un programa interpretando las acciones del autómata, más los protocolos de intercambio de datos entre dichos trozos de programa implementando las acciones respectivas de cada vector de sincronización (transacción Jini). Los valores de atributos de los autómatas involucrados proporcionarán datos significativos para identificar flujos de información a tener en cuenta.

3.1.3. Desde la Coordinación a la Cooperación

Identificamos tres niveles de autómatas cooperativos, dependiendo del grado de dinamismo que muestren en el modelo.

Nivel de Recursos: En este nivel sólo la coordinación básica está permitida entre autómatas. El modelo se compone de autómatas y vectores de sincronización, y especifica las combinaciones de acciones que ocurren concurrentemente.

Cuando hay un número fijo de autómatas, este modelo está ya tratado en la literatura [**4**], y puede ser completamente analizado por model-checking con el μ-calculo. Al modelo se le añade la posibilidad de la creación y destrucción de agentes en la sincronización.
El modelado de sistemas debe ser totalmente anónimo hasta que sea imposible para una instancia de un agente mantener conocimiento de interacciones. Esto limita el uso del modelo hasta que sea imposible mantener relaciones entre agentes, y sólo especifica el comportamiento individual (en los autómatas de estados finitos) y la coordinación inter-agentes (en los vectores de sincronización)

Nivel de Tareas: Utilizando autómatas del nivel de recursos no es posible modelizar tareas y subtareas. Estas son esenciales para describer sistemas de flujo de tareas donde es común que una tarea se divida y sea procesada en paralelo. Más tarde, cuando el procesado está completo, la tarea debe re-ensamblarse. En estos sistemas, hay concurrencia y competición entre tareas, pero no hay otras relaciones entre ellas. As pues, un atributo simple es suficiente para representar la pertenencia de agentes a una tarea. Asumimos la existencia de un servidor de nombres que produce nuevos para los atributos.
Llevando un paso más allá a nuestro modelo, podemos distinguir los autómatas del nivel de recursos (sin atributos) de los autómatas del nivel de tareas (que tienen un único atributo). El valor de este atributo debe ser obtenido en la creación (desde el servidor de nombres) o pasado por un autómata que ya exista en una interacción de sincronización. En este nivel es posible decidir sobre la propiedad de acotamiento

Nivel de Relación: Los autómatas del nivel de tareas son útiles para describer los sistemas de flujo de tareas, pero no son suficientemente expresivos para modelar sistemas en los que los agentes mantengan múltiples relaciones dinámicas, tpicas de los sistemas cooperativos soportados por computador. En estos sistemas los usuarios (agentes) se unen en grupos y los dejan cuando lo consideran necesario. Una adaptación lógica de nuestro modelo es extender el número de atributos de un autómata a un número finito de atributos. Estos atributos están tratados como variables que pueden ser testeadas por el operador de igualdad, pasadas a otros autómatas o reasignadas con interacciones sincronizadas.
Un grupo es un conjunto de autómatas cuyos atributos comparten un valor común, es decir diferentes relaciones pueden existir entre miembros de un grupo. No es posible cuando los autómatas tienen un único atributo. Desafortunadamente, tener múltiples atributos dota al modelo de la potencia del modelo de Turing, y es por lo tanto imposible decidir sobre diversas propiedades.

En la Sección 3.1.3 definimos la coordinación de agentes en un sistema distribuido mediante la especificación del conjunto de interacciones posibles entre ellos. Una interacción es un multiconjunto de acciones que algunos agentes pueden realizar a pesar de su estado interno, por ejemplo, el multiconjunto **recibir_archivo_para_impresión + enviar_archivo_a_impresión** puede describir una posible sincronización entre un usuario que tiene un archivo para imprimir y una impresora lista para realizar la impresión. Esto es muy similar al modelo de sincronización de autómatas de Arnold y Nivat [4] en donde los vectores de sincronización de longitud fija han sido reemplazados por multiconjuntos. Esta extensión permite la creación y borrado de agentes en

tiempo de ejecución, y también permite a este formalismo abarcar el modelo de las redes de Petri (ambos modelos pueden encontrarse en la Sección 2.3).

Permitir cooperar a los agentes en un sistema distribuido, requiere además que esos agentes tengan la capacidad de intercambiar información durante el proceso de sincronización. Esto se logra dotando a los agentes de una memoria local cuyo contenido condicione la capacidad de sincronización y que podrá ser modificado durante el disparo de dicha sincronización. Por este motivo se considera el formalismo de los programas paralelos basados en Objetos (POP) presentado por Engelfriet, Leit, y Rozenberg en [40] desde el que este modelo deriva.

Coordinación de Autómatas Bajo nuestro punto de vista, coordinar las actividades de agentes en sistemas distribuidos se reduce a especificar todas las posibles interacciones entre dichas actividades. Juntando todas las especificaciones de los agentes podemos asumir, sin pérdida de generalidad, que todos los agentes obedecen a la misma dinámica dada por el autómata agente.

Definición 32 ***(Autómata Agente)*** *Un Autómata Agente es una estructura* $A = (\Sigma, S, T)$ *cuyos componenestes son:*

Σ *es un conjunto finito de nombres de acciones.*

S *es un conjunto finito de estados, sea* $S_\perp = S \cup \{\perp\}$ *el conjunto obtenido mediante la adición de un valor no definido* $\perp \notin S$ *al conjunto* S*.*

$T \subseteq S_\perp \times \Sigma \times S_\perp$ *es un sistema de transición para el que se asume que el conjunto de nombres de acciones puede ser particionado en tres subconjuntos disjuntos* $\Sigma = \Sigma_1 + \Sigma_2 + \Sigma_3$ *tal que* $T = T_1 + T_2 + T_3$ *con* $T_1 \subseteq \{\perp\} \times \Sigma_1 \times S$*,* $T_2 \subseteq S \times \Sigma_2 \times S$*,* $T_3 \subseteq S \times \Sigma_3 \times \{\perp\}$*. Los elementos de* Σ_1*,* Σ_2 *y* Σ_3 *son respectivamente denominados* creaciones*,* acciones regulares *y* destrucciones*.*

Utilizaremos la notación $x \xrightarrow{a} x'$ como una abreviatura de $(x.a.x') \in T$. Cada agente actualmente activo en el sistema viene dado por su estado $s \in S$, por lo que el estado global del sistema viene dado por un multiconjunto finito de estados. El conjunto de la actividad del sistema se obtiene realizando la clausura del multiconjunto del autómata agente.

Definición 33 ***(Clausura del Multiconjunto)*** *Sea* $\langle X \rangle$ *el monoide abeliano libre generado por un conjunto finito de generadores* X*. Sea el conjunto de multiconjuntos finitos de elementos del conjunto* X*. Un elemento de* $\langle X \rangle$ *se representa como una suma formal* $u = \sum_{x \in X} u(x) \cdot x$ *donde* $u(x)$ *es el numero de ocurrencias de* x *en* u*, y sea* $0 = \sum_{x \in X} 0 \cdot x$ *el elemento nulo. La clausura del multiconjunto de un autómata agente* $A = (\Sigma, S, T)$ *es el menor subconjunto* $\langle T \rangle$ *de* $\langle S \rangle \times \langle \Sigma \rangle \times \langle S \rangle$ *tal que:*

1. $\langle T \rangle$ *contiene* T*, desde la identificación de un estado* $s \in S$ *hasta el multiconjunto* $1 \cdot s \in \langle S \rangle$ *reducido a ese estado simple, y desde la identificación de* $\perp$ *con el elemento nulo* $0 \in \langle S \rangle$*,*

2. $u \xrightarrow{0} u$ *para cada* $u \in \langle S \rangle$

3. $u \xrightarrow{a} u'$ *y* $v \xrightarrow{b} v'$ *implica que* $u + v \xrightarrow{a+b} u' + v'$

Definición 34 (Autómata coordinado) *Un sistema de autómatas coordinados viene dado por un par* $(A, Synch)$ *donde* $A = (\Sigma, S, T)$ *es un autómata agente y* $Synch \subseteq \langle\Sigma\rangle$ *es un conjunto de los llamados multiconjuntos de sincronización. Un marcado es un multinconjunto finito de estados,* $M \in \langle S\rangle$*. La relación de transición entre marcados, denominada la relación de disparo, viene dada por:* $M[e > M' \iff (e \in Synch) \wedge M \xrightarrow{e} M' in\ \langle T\rangle$*. Un sistema de autómatas coordinados, se dice que está marcado si se presenta con un marcado inicial* M_0*. Habitualmente se especifica el marcado inicial va un conjunto de sincronización particular denominado* ***inicialización****, que siempre está habilitado pero está destinado a ser disparado sólo inicialmente. Hasta que este multiconjunto tenga un estado particular no será considerado un elemento de Synch.*

Si el autómata agente es determinista, como suele ser el caso, entonces el sistema de autómatas coordinados es una instancia de un programa Gamma [8, 9, 10]. Los ejemplos clásicos de sistemas de autómatas coordinados son la sincronización de autómatas de Arnold y Nivat y las Redes de Petri.

Sincronización de autómatas de Arnold y Nivat.

La sincronización de multiconjuntos restringe el comportamiento de la clausura del multiconjunto de los autómatas agente, para obtener el comportamiento buscado. Esto es una reminiscencia del modelo clásico del producto sincronizado de los sistemas de transición, donde se consideran vectores de acciones (de una longitud fija) en lugar de multiconjuntos.

Definición 35 (Producto sincronizado de los sistemas de transición) *El producto sincronizado de los sistemas de transición* $T_i \subseteq S_i \times A_i \times S_i$ *dado como un conjunto de vectores de sincronización* $Synch \subseteq \prod_{i=1}^{n} A_i$ *es la restricción del producto del sistema de transición* $\prod_{i=1}^{n} T_i \subseteq S \times A \times S$ *donde:*
$S = \prod_{i=1}^{n} S_i\ ,\quad A = \prod_{i=1}^{n} A_i\ ,\quad y\quad s \xrightarrow{a} s' \Leftrightarrow \forall i \in \{1, ..., n\}\quad s(i) \xrightarrow{a(i)} s'(i)$
a aquellas transiciones cuya etiqueta sea un vector de transición:
$s[a > s' \Leftrightarrow a \in Synch\quad \wedge\quad \forall i \in \{1, .., n\} s(i) \xrightarrow{a(i)} s'(i)$
si cada sistema de transición viene con un estado inicial $s_0(i)$*, entonces el vector* $s_0 = (s_0(i), 1 \leq i \leq n)$ *es el estado inicial del producto sincronizado.*

Se obtiene un sistema aquivalente de autómata coordinado siendo $\sigma = A_i + ... + A_n$ una copia disjunta de cada conjunto de nombres de acción A_i tal que cada vector de sincronización $a = (a(1), ..., a(n)) \in \prod_{i=1}^{n} A_i$ corresponde sin ambigüedad al multiconjunto homónimo $a = a(1) + ... + a(n) \in \langle\Sigma\rangle$. El sistema de transición $T \subseteq S_\perp \times A \times S_\perp$ se obtiene tomando copias disjuntas de los sistemas de transición T_i en los que un nodo $\perp$ y las transiciones $\perp \xrightarrow{init_i} s_0(i)$ han sido añadidos. Las sincronizaciones son los multiconjuntos $a = a(1) + ... + a(n)$ asociados con los vectores $a = (a(1), ..., a(n)) \in \prod_{i=1}^{n} A_i$ y el multiconjunto de inicialización es $\mathbf{initialisation} = \mathbf{init}_1 + ... + \mathbf{init}_n$. Se puede observar que el multiconjunto de sincronización no puede crear ni borrar agentes y por lo tanto la estructura del sistema no puede ser modificada una vez que haya sido creada por el multiconjunto de sincronización de inicialización. Una situación perfectamente simétrica se puede encontrar en las Redes de Petri, cuyos agentes y recursos son creados o consumidos pero no tienen comportamiento.

Teniendo una red de Petri como puede verse en la Sección 2.3 un sistema equivalente de autómatas coordinados puede ser definido como sigue:

1. $S = P$
2. $\Sigma_1 = \{\mathbf{in}_p | p \in P\}, \Sigma_2 = \emptyset, \Sigma_3 = \{\mathbf{out}_p | p \in P\}$
3. $T_1 = \{\perp \stackrel{\mathbf{in}_p}{\rightarrow} s | p \in P\}, T_2 = \emptyset, T_3 = \{\perp \stackrel{\mathbf{out}_p}{\rightarrow} s | p \in P\}$
4. $Synch = \{\sum_{p \in P} F(p,t) \cdot \mathbf{out}_p + \sum_{p \in P} F(t,p) \cdot \mathbf{in}_p | p \in P\}$
5. $\mathbf{initialisation} = \sum_{p \in P} M_0(p) \cdot \mathbf{in}_p$

Puede ser interesante observar que los sistemas de autómatas coordinados pueden ser traducidos en redes de Petri Etiquetadas. Una red de Petri Etiquetada es una red de Petri con una función de transición etiquetada $\ell : T \rightarrow L$. La relación de disparo de una Red de Petri Etiquetada se obtiene mediante el etiquetado de la relación de disparo de la red de Petri subyacente:

$$M[a > M' \Leftrightarrow M[t > M' \wedge \ell(t) = a$$

Esta relación de transición es por lo tanto generalmente no determinista. La traducción es como sigue. Se fija una enumeración para cada multiconjunto de sincronización $a \in Synch$ de un sistema de autómatas coordinados $S = (\Sigma, S, T, Synch)$, por ejemplo, se elige un vector $\overline{a} = (a_1, ..., a_n)$ tal que $a = a_1 + ... + a_n$. Se llama realización de a a cualquier vector de transiciones $t = (t_1, ..., t_n)$ con t_i etiquetado con a_i. Por ejemplo, sea t_i una transición de la forma $\perp \stackrel{a_i}{\rightarrow} s_i'$, $s_i \stackrel{a_i}{\rightarrow} s_i'$, o $s_i \stackrel{a_i}{\rightarrow} \perp$ si a_i pertenece respectivamente a Σ_1, Σ_2, o Σ_3. La etiqueta resultante viene dada por:

1. $P = S$
2. T es el conjunto de todas las realizaciones t de cada sincronización a con una t etiquetada a:$\ell(t) = a$,
3. $F(s,t) = \sharp\{i | t_i = s \stackrel{a_i}{\rightarrow} x \text{ para cada } x \in S_\perp\}, F(t,s) = \sharp\{i | t_i = x \stackrel{a_i}{\rightarrow} s \text{ para cada } x \in S_\perp\}$, y
4. M_0 es el marcado inicial de S

A su vez, si hablamos de las redes de Petri Objetuales de Valk, ya vistas en la Sección 2.3.3, un sistema de objetos elemental, puede ser traducido a autómatas coordinados como sigue:

El autómata agente (Σ, S, T_A) se define por:

1. **Conjunto de estados S**:

   Sea $\overline{B}_i = \{\overline{b} | b \in B_i\}$ y $f_i : B_i \rightarrow \overline{B}_i$ la biyección dada por $f_i(b) = \overline{b}$ $(1 \leq i \leq n)$, $S = P_1 \bigcup (\cup_{i=1}^{n} S_i)$ con $S_i = P_2 \times (B_i \cup \overline{B}_i)$

2. **Conjunto de nombres de acción Σ**:

$\Sigma_1 = \{in_p | p \in P_1\} \bigcup \{init_{p,b} | p \in P_2, b \in \cup_{i=1}^{n}(B_i \cup \overline{B}_i)\}$

$\Sigma_2 = \{in_{p,b}, out_{p,b} | p \in P_2, b \in \cup_{i=1}^{n} B_i\} \bigcup \{tr_{p,q,b} | p, q \in P_2, b \in \cup_{i=1}^{n}(B_i \cup \overline{B}_i)\}$

$\bigcup \{in\_tr_{p,q,b}, out\_tr_{p,q,b} | p, q \in P_2, b \in \cup_{i=1}^{n} B_i\}$

$\Sigma_3 = \{out_p | p \in P_1\}$

3. **Sistema de transición T**:

$T_1 = \{\perp \overset{in_p}{\rightarrow} p | p \in P_1\} \bigcup \{\perp \overset{init_{p,b}}{\rightarrow} (p,b) | p \in P_2, b \in \cup_{i=1}^{n}(B_i \cup \overline{B}_i)\}$

$T_2 = \{(p,\overline{b}) \overset{in_{p,b}}{\rightarrow} (p,b), (p,b) \overset{out_{p,b}}{\rightarrow} (p,\overline{b}) | p \in P_2, b \in \cup_{i=1}^{n} B_i\}$

$\bigcup \{(p,b) \overset{tr_{p,q,b}}{\rightarrow} (q,b) | p, q \in P_2, b \in \cup_{i=1}^{n}(B_i \cup \overline{B}_i)\}$

$\bigcup \{(p,b) \overset{in\_tr_{p,q,b}}{\rightarrow} (q,b), (p,b) \overset{out\_tr_{p,q,b}}{\rightarrow} (q,\overline{b}) | p, q \in P_2, b \in \cup_{i=1}^{n} B_i\}$

$T_3 = \{p \overset{out_p}{\rightarrow} \perp | p \in P_1\}$

El conjunto Synch de multiconjuntos de sincronización se define como sigue.

1. Para cada sistema de transición $t \in T$ con $t\rho = \emptyset$, hay un multiconjunto de sincronización asociado con cada tipo de objeto posible y cada posible estado de dichos objetos.

   Sea $V_t = \sum_{p \in P_1 \cap {}^\bullet t} W_1(p,t).out_p + \sum_{p \in P_1 \cap t^\bullet} W_1(t,p).in_p$, V_t corresponde con la activación de t en la parte de red de Petri de SN. El conjunto $Synch_t$ de multiconjuntos de sincronización corresponde con cada posible activación de t, que contiene los multiconjuntos de la forma $Synch_t = V_t + \sum_{q_1 \in P_2 \cap t^\bullet} \sum_{b \in C_{q_1}} tr_{q_1,q_2,b}$ tal que existe algún $i \in W_2(q_1,t) \cap W_2(t,q_2)$ con $q_2 \in P_2 \cap t^\bullet$ y $C = c \cup f_i(B_i - c)$

2. Para cada transición objeto $e \in E$ tal que $\sigma e = \rho e = \emptyset$, hay un multiconjunto de sincronización asociado a cada lugar del sistema que puede contener este objeto. $Synch_e$ denota el conjunto de multiconjuntos de sincronización asociado con e

$$Synch_e = \left\{ \sum_{b \in {}^\bullet e} out_{p,b} + \sum_{b \in e^\bullet} in_{p,b} | p \in P_2 \right\}$$

3. El multiconjunto de sincronización $Synch_{t,e}$ asociado con cada transición $t \in T$ del sistema y cada objeto transición e tal que $t\rho e$,

$$Synch_t = \left\{ V_t + \sum_{b \in {}^\bullet e} out - tr_{q_1,q_2,b} + \sum_{b \in e^\bullet} in - tr_{q_1,q_2,b} \right\}$$

   donde $P_2 \cap {}^\bullet t = \{q_1\}$ y $P_2 \cap t^\bullet = \{q_2\}$.

4. Para cada par de objetos transición (e, e') tal que $e\sigma e'$, hay un multiconjunto de sincronización asociado con cada objeto lugar de la red del sistema SN,

que debe contener los dos objetos. El conjunto $Synch_{e,e'}$ de multiconjuntos de sincronización asociados con (e, e') se define por:

$$Synch_{e,e'} = \left\{ \sum_{b \in {}^\bullet e} out_{p,b} + \sum_{b \in e^\bullet} in_{p,b} + \sum_{b' \in {}^\bullet e'} out_{p,b'} + \sum_{b' \in e'^\bullet} in_{p,b'} | p \in P_2 \right\}$$

5. Finalmente la marca inicial viene dada por:

$$inicialización = \sum_{p \in P_1} M_1(p).in_p + \sum_{i=1}^{n} \left[\sum_{b \in m_{0_i}} init_{p_i,b} + \sum_{b \notin m_{0_i}} init_{p,\bar{b}} \right]$$

donde $p_i \in P_2$ tal que $(ON_i, m_{0_i}) \in M_2(p_i)$

3.1.4. Autómatas Cooperativos

Se asumen como ciertas las siguientes afirmaciones.

1: El comportamiento autónomo de un agente puede ser descrito por un autómata finito.

2: Hay un número finito de tipos de agentes y, por lo tanto, de autómatas.

Si se añade a todos los autómatas un estado ficticio $\perp$ con transiciones entrantes y salientes modelizando la creación y destrucción de agentes, se obtiene que el comportamiento de los agentes puede ser descrito por un autómata agente.

3: El estado de un sistema es un multiconjunto de tokens, que representan a los agentes "vivos" con sus respectivos estados de S, más un token en el estado ficticio $\perp$, como una fuente de agentes.

4: Las relaciones entre agentes están codificadas mediante registros anexados a los tokens agente. Todos los registros especifican los valores de una lista fija de atributos. El valor asignado a los atributos es arbitrario, pero la comparación de valores de atributos es significativa.

5: Los cambios de estado resultan de las transacciones síncronas. Parte del control de transacciones consiste en comprobar y modificar las relaciones entre agentes.

6: Hay un número finito de tipos de transacciones.

Dichas afirmaciones previas permiten lo siguiente:

Definición 36 ***(Autómatas Cooperativos)*** *Un sistema de autómatas cooperativos (CA) es una Quintupla* (Σ, S, T, Att, R) *donde* (S, Σ, T) *es un autómata agente (AA), Att es un conjunto finito de atributos y R es un conjunto finito de reglas de transacción. Cada regla* $r = (\mathbf{synch}, \mathbf{guard}, \mathbf{update})$ *consiste en un vector* **synch** *de acciones de sincronización, requerido por los agentes respectivos que se involucran en la transacción, una guarda* **guard** *especificando las relaciones que deben mantenerse entre ellos para permitir la transacción y una actualiación* **update** *de las relaciones en la compleción de la transacción.*

- *Un **vector de sincronización** es un vector* **synch** $= A_1 \cdot a_1 + ... + A_n \cdot a_n$ *donde* a_i *son nombres de acción* ($a_i \in \Sigma$) *y la* A_i *son nombres de agentes con el rango limitado a la regla (limitado en cada instacia a los tokens agente con estados locales* s_i *que producen acciones* a_i*).*

- *Una guarda* **guard** *es una conjunción de restricciones de igualdad (respectivamente desigualdad)* $A_i \cdot u = A_j \cdot v$ *o* $A_i \cdot u = 0$ *(respectivamente* $A_i \cdot u \neq A_j \cdot v$ *o* $A_i \cdot u \neq 0$*) donde* a_i *y* a_j *son acciones regulares o de destrucción en* ${}^\bullet r = \{A_i | a_i \in \Sigma_2 \cup \Sigma_3\}$ *y* u *y* v *son atributos*

- *Una **actualización*** **update** *es una secuencia finita de asignaciones* $A_i \cdot u := A_j \cdot v$ *o* $A_i \cdot u :=$ **new** *donde* a_i *es una acción de creación o regular en* $r^\bullet = \{A_i | a_i \in \Sigma_2 \cup \Sigma_3\}$*,* u *y* v *son atributos tales que no ocurren asignaciones múltiples.*

Un autómata token es un par $x = (s, p)$ donde s$\in S$ es el estado del token x (escrito como $x \cdot estado = s$), y $\rho : Att \rightarrow IN$ guarda los valores $x \cdot u = \rho(u)$ de los atributos u del token x. Se han escogido valores enteros para los atributos, con 0 como valor por defecto, pero cualquier otro conjunto contable habra servido también. Un estado del sistema o marcado es un multiconjunto finito de tokens. El disparo en un marcake M de una regla $r =$ (**synch**, **guard**, **update**) con un vector de sincronización **synch** $= A_1 \cdot a_1 + ... + A_n \cdot a_n$ referido a un marcado $M' = M - R + R$, donde $l = \{x_i | A_i \in {}^\bullet r\}$ y $R = \{x'_i | A_i \in r^\bullet\}$ son los multiconjuntos de tokens respectivamente consumidos o generados por el disparo.

Activación de una regla

Un vínculo $\{A_i = x_i | A_i \in {}^\bullet r \wedge x_i \in M\}$ activa r si y sólo si:

1. $L = \{x_i | A_i \in {}^\bullet r\}$ es más pequeño que M;
2. $x_i \cdot state \xrightarrow{a_i} s_i$ en AA cuando $a_i \in \Sigma_2$;
3. $x_i \cdot state \xrightarrow{a_i} \perp$ en AA cuando $a_i \in \Sigma_3$;
4. la guarda es válida bajo esa ligadura.

Evolución sincronizada de un agente

El disparo de r bajo una ligadura activada supone:

1. La producción de tokens agente x'_i tales que $\perp \xrightarrow{a_i} x'_i \cdot state$ en AA para todas las acciones de creación $a_i \in \Sigma_1$;
2. La transformación de los tokens x_i en tokens x'_i tales que $x_i \cdot state = s_i$ para todas las acciones regulares $a_i \in \Sigma_2$;
3. El borrado de los tokens agente x_i para todas las acciones de destrucción $a_i \in \Sigma_3$.

Actualización de relaciones

Los atributos $u, v, ...$ de los tokens producidos en $R = \{x'_i \mid A_i \in r^\bullet\}$ son junto con los valores respectivos $x'_i \cdot u, x'_i \cdot v, ...$ tales que debe mantenerse con $A_i = x_i$ para $A_i \in {}^\bullet r$ y $A_i = x_i$ para $A_i \in r^\bullet$ lo siguiente:

1. $A'_i \cdot u = A_j \cdot v$ cuando $A_i \cdot u = A_j \cdot v$ esté en **update** y $A_j \in {}^\bullet r$;
2. $A'_i \cdot u = A_i \cdot u$ cuando $A_i \in {}^\bullet r \cap r^\bullet$ y A_i no esté asignado en **update**;
3. $A'_i \cdot u = 0$ cuando $A_i \in r^\bullet \backslash {}^\bullet r$ y A_i no esté asignado en **update**;
4. Todos los valores $A'_i \cdot u$ tales que $A_i \cdot u := new$ esté en **update** son diferentes (para toda i y u), y difieren de los valores de los atributos de todos los tokens de M;
5. $A'_i \cdot u = A'_j \cdot v$ cuando $A_i \cdot u = A_j \cdot v$ esté en **update** y $A_j \in {}^\bullet r \cap r^\bullet$.

De la afirmación 1, el modelo abstrae del tratamiento funcional de datos contenidos por los agentes y centrados en el control, usando los nombres de acción como abreviaturas para los métodos que operan sobre los propios datos. De las asunciones 4 y 5 los datos distribuidos que representan las relaciones inter-agentes y que no pueden ser operados por los agentes. Estos datos protegidos (atributos con su valor) son las propiedades del controlador abstracto descrito por las reglas de transacción. De la afirmación 5, las transacciones síncronas aunque éste no sea el caso en muchos sistemas reales.

Estas afirmaciones facilitan el uso de un simulador del modelo donde los estados puedan visualizarse como una tabla de tokens o como un grafo de relaciones. Esto permite también la subsiguiente traducción de los modelos a un código ejecutable (prototipo del sistema) donde los autómatas agente se expandan a objetos distribuidos que pueden interactuar con las reglas de transacción.

Un sistema marcado de autómatas cooperativos es uno de autómatas cooperativos con un marcado inicial. Usualmente se especifica el marcado inicial va una regla especfica llamada *regla de inicialización*, con una parte condicional cuyo disparo produce el marcado inicial. Esta regla está siempre activa, pero está dispuesta para ser disparada sólo inicialmente.

Una Jerarquía de Autómatas Cooperativos

La memoria local de los tokens se presenta como listas de atributos/valor. El número de atributos parece dar una medida acerca de la expresividad del modelo en términos de flujo de información entre objetos. Sin embargo la expresividad completa se alcanza permitiendo tan solo dos atributos, como se demuestra en **[5]**.

Un sistema de transición $T' \subseteq S' \times A' \times S'$ se dice que implementa un sistema de transición $T \subseteq S \times A \times S$ cuando contiene un subsistema de transición $\widetilde{T} \subseteq \widetilde{S} \times \widetilde{A} \times \widetilde{S}$ para el que existe un par de aplicaciones suprayectivas $\alpha : \widetilde{S} \to S$ y $\beta : \widetilde{A} \to A$ que:

1. $\widetilde{s}_1 \overset{\widetilde{a}}{\to} \widetilde{s}_2$ en $\widetilde{T}$ supone que $\alpha(\widetilde{s}_1) \overset{\beta(\widetilde{a})}{\to} \alpha(\widetilde{s}_2)$ en T, y

2. $\alpha(\widetilde{s}_1) \overset{\beta(\widetilde{a})}{\rightarrow} \alpha(\widetilde{s}_2)$ en T, implica que existe algún $\widetilde{a} \in \widetilde{A}$ y $\widetilde{s}_2 \in \widetilde{S}$ tales que $\beta(\widetilde{a}) = a$, $\alpha(\widetilde{s}_2) = s_2$ y $\widetilde{s}_1 \overset{\widetilde{a}}{\rightarrow} \widetilde{s}_2$ en $\widetilde{T}$

La jerarquía por lo tanto consiste en sólo tres clases de autómatas: La clase completa (con dos o más atributos) está bien adaptada para el diseño de aplicaciones *groupware* y computaciones móviles, la clase intermedia (con sólo un atributo) que se ajusta a las necesidades de las aplicaciones *workflow*, y el modelo básico (sin atributos) que permite sólo la coordinación pura de eventos sin flujo de información.

3.1.5. Un ejemplo práctico

En esta ección se presenta un ejemplo práctico de modelado de un sistema mediante el modelo de los autómatas cooperativos. La idea es describir el comportamiento de un Script, dentro de un sistema operativo. El propósito del Script, no es otro que la instalación y compilación de software. El primer paso es identificar cada autómata que veamos involucrado, es decir, cada una de los procesos independientes o de agentes que intervengan en el proceso completo. El Script no actua aisladamente sino dentro del entorno del sistema, por tanto debemos definir tanto al propio Script como los elementos con los que se relaciona.

En este ejemplo, se pueden distinguir cinco autómatas distintos:

- SCRIPT: Es el autómata que describe los distintos estados por los que atraviesa el propio Script.Automaton. En principio y si no ocurrieran errores, debera tomar ficheros de una lista, copiarlos en su lugar correspondiente, compilarlos, e iterar hasta que se le acabase la lista.

- MANAGER: Es el autómata que representa al manipulador de disco. Este autómata debe realizar las comunicaciones efectivas con el disco, buscando directorios, creándolos en su caso, y escribiendo el archivo en el disco. Por supuesto, atendiendo siempre a la posibilidad de que el disco se llenase. A su vez deberá mantener una colaboración con el Script, que será quien solicite sus servicios.

- DISK: Este autómata representa al disco fsico. En realidad, su única función es dar información acerca del estado (si lleno o no) de la memoria secundaria al manipulador de disco y al compilador, y permitir el cambio entre "lleno" y "vacío" según se haga operaciones en uno u otro sentido.

- LIST: Este autómata representa la información de los distintos ficheros que deben ser instalados. Al igual que ocurre con el disco, su única función es dar información sobre si está o no vacía la lista. A diferencia del disco, no hay posibilidad de rellenar la lista una vez se haya vaciado.

- COMPILER: Este autómata representa al compilador del sistema. Su función sera comprobar que el archivo a instalar es correcto y generar en el disco la versión ya compilada. Por supuesto es una mera represetnación del conjunto de estados posibles y sólo representa posibilidades, no el funcionamiento de un compilador real..

Una vez que se ha identificado a cada autómata en el sistema, el siguiente paso es identificar para cada autómata los distintos estados en los que puede encontrarse, y por supuesto, las distintas acciones que permitan la transición entre estados, teniendo en cuenta que estos estados y estas acciones configurarán el diagrama de transición de estados de cada uno de los autómatas. El comportamiento de los autómatas descritos se encuentra en los diagramas de estados de la Figura 3.2.

Una vez definidos los diagramas de transición de estados (y por tanto, identificado los estados y las acciones) debemos identificar la colaboración entre agentes es decir, el mecanismo de sincronización que utiliamos. Las reglas de transición, mediante los vectores de sincronización, las guardas y las actualizaciones asociadas a cada regla:

```
name: Initialization
 synch:   dsk.initDisk()
 guard:   null
 update:  null


name: Create_system
 synch:   cmp.initCompiler()
          + mng.initManager()
 guard:   null
 update:  null


name: Create_Script
 synch:   scr.newScript()
          + lst.newList()
 guard:   null
 update:  lst.name = new
          lst.item = new
          scr.list = lst.name
          scr.archive = lst.item


name: Creating_or_finding_Directory
 synch:   scr.createfinddir()
          + mng.lookforfile()
 guard:   null
 update:  mng.file = scr.archive


name: Finding_directory
 synch:   mng.createdir()
          + dsk.reserve()
 guard:   null
 update:  null
```

```
name: Panic_Error_Diskfull
 synch:    mng.error_diskfull()
           + dsk.liberate()
           + scr.error_disk()
           + lst.endList()
 guard:    mng.file == scr.archive
           scr.list == lst.name
 update:   null

name: Start_compilation
 synch:   mng.copyfile()
          + dsk.reserve()
          + scr.compilefile()
          + cmp.compilate()
 guard:   mng.file == scr.archive
 update:  cmp.source = scr.archive

name: Saving_compilation
 synch:   cmp.verifyspace()
          + dsk.reserve()
 guard:   null
 update:  null

name: Panic_Error_compilation
 synch:   cmp.error_compiling()
          + lst.endList()
          + scr.error_compilation()
          + dsk.liberate()
 guard:   cmp.source == scr.archive
          scr.list == lst.name
 update:  null

name: Lookfornew_File
 synch:   scr.newfile()
          + lst.nextitem()
 guard:   scr.list == lst.name
 update:  lst.item = new
          scr.archive = lst.item
```

En el modelo podemos encontrar ejemplos de los tres tipos de autómatas, sin atributos (como el autómata DISK) con un único atributo (como los autómatas COMPILER y MANAGER) y finalmente el modelo completo de autómatas, con dos atributos (como los autómatas LIST y SCRIPT)

Además, se pueden encontrar elecciones no deterministas entre acciones.

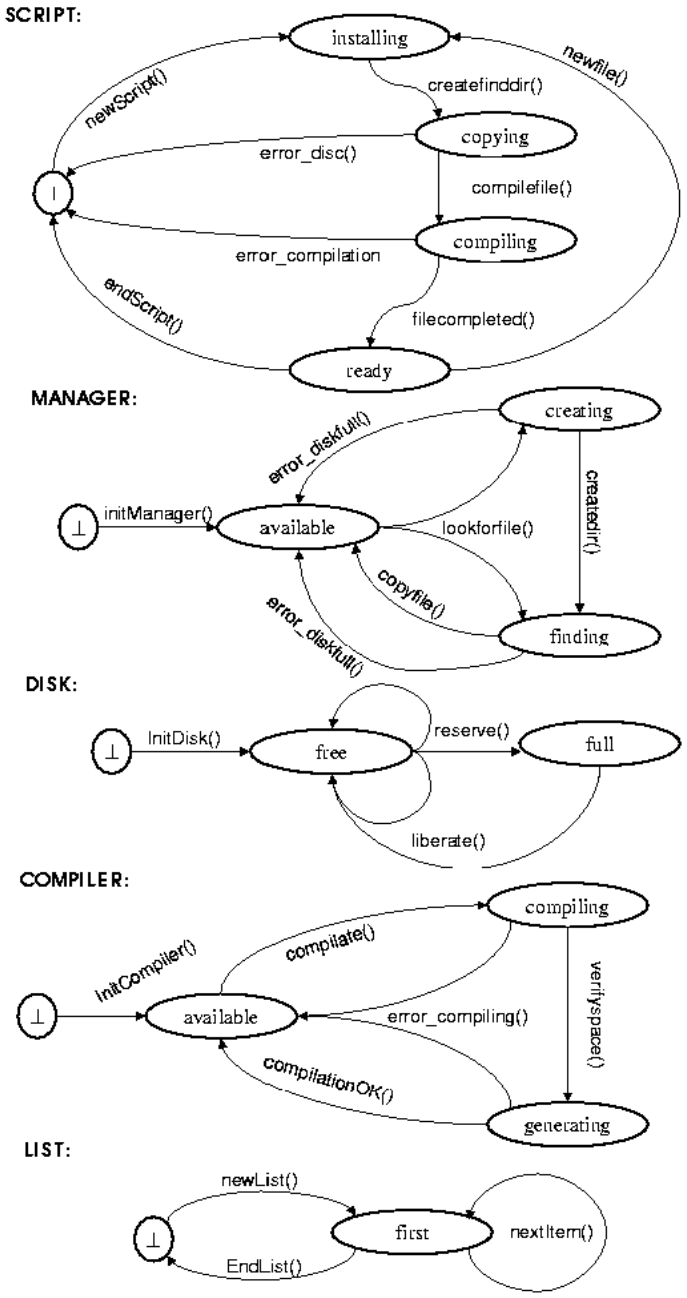


Figura 3.2: Diagrama de estados de los Autómatas cooperativos del ejemplo

3.1.6. Comparación de modelos: Simulación de los Autómatas Team

A continuación se presenta la conversión de un sistema simple descrito con Autómatas Team ya vistos en la Sección 2.4.2 en otro sistema con idénticas características pero construido mediante el modelo de los Autómatas Cooperativos. Mostramos, por tanto, que la capacidad expresiva de los AC es, al menos, igual que la de los Autómatas Team.

Los pasos para construir un sistema de Autómatas Team usando Autómatas Cooperativos son los siguientes:

1. Con cada autómata componente $(Q, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, I)$ construimos un autómata agente (Σ, S, T) en el que $S = Q \cup \{\bot\}$, $\Sigma = \Sigma_{inp} \cup \Sigma_{out} \cup \Sigma_{int} \cup \{init\}$, y $T = \delta \cup \{(\bot, init, I)\}$, de este modo, tenemos un autómata agente con los mismos estados más el estado inicial, el mismo alfabeto de acciones más la acción inicial, y el mismo sistema de transición de estados más la inicial.

2. Para cada autómata T $(\prod_{i\in\mathcal{I}} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, \prod_{i\in\mathcal{I}} I_i)$, se construye un sistema de autómatas cooperativos (Σ, S, T, Att, R), mediante la definición de los autómatas agentes del paso anterior y para cada transición $a \in \delta$ se incluye una regla en R que involucre a los agentes capaces de realizar dicha acción a. (Si se tuviera los mismos nombres de acciones en varios autómatas componentes, es preciso construir una regla para cada combinación de agentes posible). Si la transición es considerada interna, entonces la regla sólo involucra a un agente. Si se trata de una acción *input* u *output* la regla incluirá al resto de agentes involucrados. En este nivel de simulación no precisamos atributos, es evidente que $(Att = \emptyset)$ y, por tanto, en todas esas reglas tanto la guarda como la sincronización serán vacías. Sólo necesitaremos los atributos para identificar los agentes como pertenecientes a un *subTeam* de forma dinámica, mediante un uso obvio.

Resulta evidente (tras el primer paso) que una colección de autómatas componente puede ser simulada de forma sencilla mediante Autómatas Cooperativos. Sólo resta pues en el segundo paso representar la coordinación de acciones para cada grupo de autómatas componente y, por tanto, representar el Autómata Team.

Un ejemplo de conversión se muestra en la Figura 3.3. El ejemplo es muy simple; sin embargo, es útil para comprobar la sencillez de conversión desde Autómatas Team al modelo de los Autómatas Cooperativos. En una conversión completa se deben representar el resto de caractersticas de los Autómatas Team, como son, los roles, los *subTeams*, las ligaduras entre autómatas, dependencias, etc. Sin embargo, todas estas caractersticas son fácilmente convertibles al tratarse de combinaciones de acciones que, por tanto, se pueden incluir como reglas combinando los correspondientes vectores de sincronización obtenidos en el paso 2.

Proposición 2 Cualquier sistema de Autómatas Team es equivalente a algún sistema de Autómatas Cooperativos.

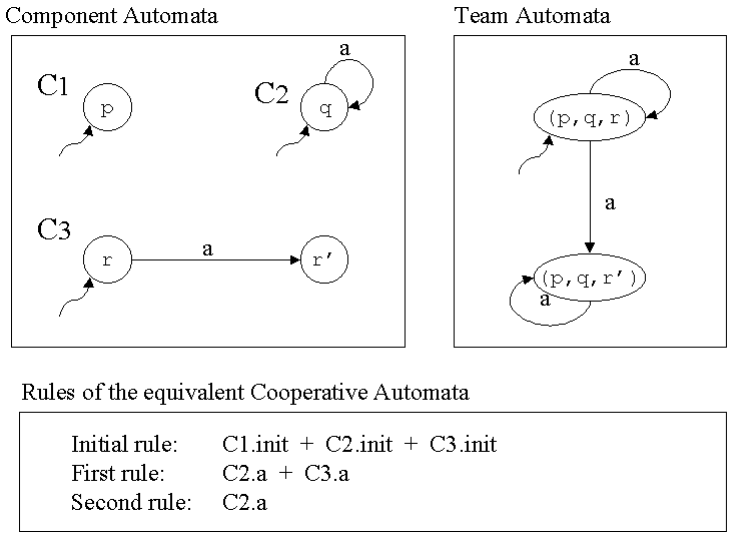


Figura 3.3: Autómatas Team *vs.* Autómatas Cooperativos

3.2. Autómatas Cooperativos Extendidos (ACE)

3.2.1. Justificación de la extensión: un ejemplo sencillo

Como ya se vió en [62], con los Autómatas Cooperativos podemos modelizar sistemas distribuidos, CSCW y aplicaciones *groupware*; sin embargo, en ocasiones es necesario el uso de modelos muy complejos para mostrar situaciones en apariencia simples. Este es el caso de aquellos sistemas en los que el conjunto de estados depende de algún valor numérico externo como, por ejemplo, la posición geográfica.

Siempre que las posibles transiciones entre estados varíen dependiendo de un valor numérico, el diagrama de estados y de transiciones de estados deberá incluir todas estas variaciones, siendo en realidad el autómata resultante equivalente a una serie de autómatas, (uno por cada posible valor numérico), más las transiciones que unan los posibles estados intermedios de estos automatas entre sí. El ejemplo más sencillo es aquel autómata que dependiendo de su situación, de su prioridad o de cualquier valor en el momento de su creación cambia su comportamiento. Este autómata suele presentarse con una transición que traslade al autómata desde el estado *dead* a los estados iniciales de cada uno de los comportamientos, y a partir de ahí funcione como un autómata normal quedando una gran parte del autómata como inalcanzable.

Otra posibilidad aún más traumática en cuanto a la expansión del grafo de transiciones es que el comportamiento del autómata pueda variar en cualquier momento dependiendo de dicho valor numérico. Para ilustrar esto, se presenta un ejemplo, en apariencia sencillo, pero que modelizado bajo el formalismo de los Autómatas Cooperativos da lugar a un autómata muy complejo. Supongamos que deseamos representar un sistema en el que el conjunto de estados posibles (y, por tanto, de acciones posibles) de uno de los autómatas del sistema cambia según la posición, y además, este cambio de posición puede ocurrir en cualquier momento. La solución en el modelo de

los Autómatas Cooperativos consiste en que hay que distinguir como estados distintos aquéllos que tienen lugar en posiciones distintas. A continuación mostramos un ejemplo de un sistema de estas características.

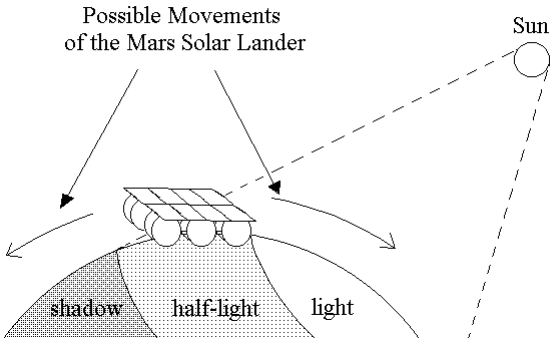


Figura 3.4: Posiciones posibles y movimientos del MSL

El prototipo de la sonda solar marciana (Figura 3.4) Mars Solar Lander (MSL) realiza tres funciones principales: cargar las baterías con energía solar, tomar fotografías infrarrojas y recoger muestras de roca. Sin embargo, estas tres funciones no pueden realizarse en cualquier lugar. Si trazamos una linea imaginaria que traspase las zonas de sol, penumbra y sombra, tenemos que:

- la función de recoger muestras puede realizarse siempre independientemente de la posición del autómata en una de las tres zonas,
- la función de tomar fotografías infrarrojas sólo puede hacerse en la zona de penumbra y en la de sombra pues en la zona iluminada por el sol las fotografías saldrían veladas dada la sensibilidad de la película,
- por último, la función de recarga de baterías sólo puede hacerse en la zona de sol y en la de penumbra pero no en la zona de sombra, pues la intensidad de la luz es demasiado débil.

Por tanto, si identificamos las acciones como *light, half_light* y *shadow* el autómata puede realizar en la posición *light* las funciones de recogida de rocas, la de recarga, y la de moverse a la posición *half-light*, donde podrá realizar las tres funciones antes previstas y también retroceder o moverse a la posición *shadow* donde puede recoger rocas o tomar fotografías y también retroceder mas no cargar baterías. Las acciones que representen el cambio de estados serán del tipo «go_X_Y_Z» es decir, *ir del lugar X al lugar Y a realizar la función Z* o bien acciones *Z,* esto es, *realizar la función Z sin cambiar de sitio* o finalmente acciones «return_Y_X_Z» que indica *volver del lugar Y al lugar X a realizar la función Z*. Está claro que por cada cambio posible tendremos una acción asociada y en la Figura 3.5 se ha representado el diagrama de estados así como las transiciones posibles entre ellos, No obstante no se ha identificado cada acción puesto que su elevado número haría ilegible el grafo. Para clarificar este diegrama de estados y transiciones de estados, se han representado los distintos estados como

parejas (función, lugar) siendo las funciones *Collect, Charge* y *TakeP*, y representando el lugar como p1, p2 y p3 (zona de sol, de penumbra y de sombra, respectivamente).

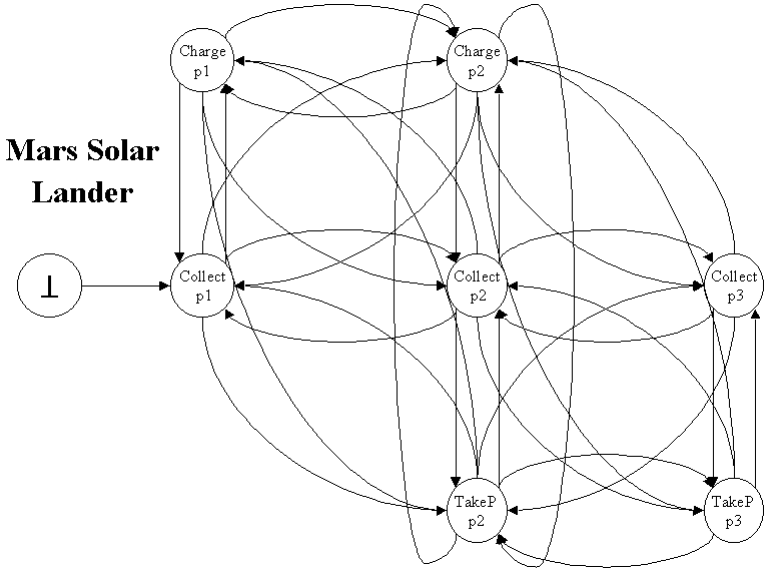


Figura 3.5: Ejemplo de MSL con un Autómata Cooperativo

Es evidente la confusión que este autómata presenta. Y no sólo tenemos que el autómata resulta confuso, sino que, además, como tenemos que representar individualmente las posibles acciones nos encontramos con que es preciso definir una regla para cada una de ellas, lo que nos da un total de 34 reglas. Estamos hablando de un sistemas simple, con un único agente y que no requiere sincronización con ningún otro agente. Si tuviéramos que añadir además más agentes en el sistema como por ejemplo un satélite a quien enviar los resultados, un laboratorio estático donde analizar las muestras o cualquier otro tipo de agente con quien interactuar, cooperar, competir o sincronizarse habría que añadir al conjunto de reglas todas aquellas que permitieran dichas interacciones, cooperaciones, competiciones o sincronizaciones.

Queda claro que pese a que el sistema es de fácil definición y sólo contiene un único autómata, el modelo resultante es muy complejo por el distinto comportamiento que presenta el autómata en las distintas posiciones. ¿que pasaría si se tratase de un sistema más complejo?

3.2.2. Una solución: Autómatas Cooperativos Extendidos

¿Cómo podría simplificarse este modelo? Una respuesta a esta pregunta consiste en considerar la posibilidad de utilizar un tipo especial de atributos para representar la posición del MSL: atributos numéricos. No obstante, estos atributos plantean a su vez nuevos interrogantes. ¿Estos atributos numéricos tendrán las mismas características de los atributos clásicos de los Autómatas Cooperativos? ¿Indican la pertenencia a una tarea? ¿Se atienen a las mismas reglas?

Estos nuevos atributos deben tener un papel completamente distinto de los anteriores. Para las cuestiones relacionadas con la sincronización entre tareas sólo se utilizarán los atributos originales, llamados también por este motivo atributos de tarea o identificadores, puesto que identifican a la tarea a la cual pertenece el autómata. Si se quiere mantener las características deseables de los Autómatas Cooperativos en los Autómatas Cooperativos Extendidos, hay que limitar de forma drástica el papel de los atributos numéricos, considerándolos como información adicional del estado del autómata y manteniendo por tanto los tres niveles de autómatas (sin atributos, con un atributo y con dos o más atributos) referidos únicamente a los atributos de tarea y sin tener en cuenta la existencia o no de los atributos numéricos. Sin embargo, dado que estos atributos numéricos son modificables en tiempo de ejecución, la complejidad del árbol de derivación de estados se incrementará considerablemente. Si se quiere, por tanto, mantener la decibilidad de propiedades del modelo original es necesario limitar de forma tajante el conjunto de valores posibles de los atributos numéricos y exigir que sean de tipo entero y dentro de un rango finito de valores.

As pues, se considera la existencia de guardas implcitas que limitan el rango de valores de estos nuevos atributos. Para simplificar el modelo no se incluirán estas guardas implcitas en las reglas, pero s se considerarán como necesarias para permitir el disparo de una regla.

Definición 37 (Sistema de Autómatas Cooperativos Extendido (ACE)) *Un Sistema de Autómatas Cooperativos Extendido es una quintupla* $(\Sigma, S, T, (Att_1, Att_2), R)$ *donde:*

(Σ, S, T) *es un autómata agente (AA),*

Att_1 *es un conjunto finito de atributos de tarea,*

Att_2 *es un conjunto finito de atributos numéricos en el cual* $\forall u \in Att_2, (-\varsigma) < u < \tau$ *con* $\varsigma, \tau \in \mathbb{N}$ *y tal que* $Att_2 \cap Att_2 = \emptyset$ *y*

Res un conjunto finito de reglas de transición. Cada regla $r =$ (synch, guard, update) *consta de un vector* synch *de acciones sincronizadas requeridas por los agentes respectivos involucrados en la transacción, una guarda* guard *que muestra las relaciones que deben mantener los atributos para poder realizar la transacción, y una actualización* update *de las relaciones al completarse la transacción*

- *Un vector de sincronización es un vector synch* $= A_1 \cdot a_1 + ... + A_n \cdot a_n$ *donde* $a_\imath$ *son nombres de acción* ($a_\imath \in \Sigma$) *y* $A_\imath$ *son nombres de agentes, con ámbito limitado a la regla (es decir, limitado en cada instanciación a tokens agente con estados locales* $s_\imath$ *y acciones* $a_\imath$*).*

- *Una guarda es una restricción (o restricciones) que depende del tipo de atributo:*

  si $u, v \in Att_1$ *entonces* $A_\imath.u = A_j.v$ *(o* $\neq$*),* $A_\imath.u = default$ *(o* $\neq$*)*

  si $u, v \in Att_2$ *entonces* $A_\imath.u = A_j.v$ *(o* $\neq$*),* $A_\imath.u = const$ *(o* $\neq$*),* $A_\imath.u \geqslant A_j.v$ *(o* $<$*),* $A_\imath.u > A_j.v$ *(o* $\leqslant$*)*

  donde $a_\imath$ *y* a_j *son acciones regulares o de destrucción en* ${}^\bullet r = \{A_\imath | a_\imath \in \Sigma_2 \cup \Sigma_3\}$ *y* u*,* v *son atributos*

- *Una actualización es un conjunto finito de asignaciones (en el caso de atributos de tarea) u operaciones simples y asignaciones (en el caso de atributos numéricos):*

- *si* $u, v \in Att_1$ *entonces* $A_i.u := A_j.v$, $A_i.u := new$
- *si* $u, v \in Att_2$ *entonces* $A_i.u := A_j.v$, $A_i.u := const$, $A_i.u+ = const$, $A_i.u- = const$, $A_i.u := new$ *enditemize donde* a_i *y* a_j *son acciones regulares o de creación en* $r^\bullet = \{A_i | a_i \in \Sigma_1 \cup \Sigma_2\}$ *y u,v son atributos tales que no hay asignaciones múltiples. Se asume que en las actualizaciones de los atributos numéricos se mantienen las restricciones que limitan los valores posibles.*

Una vez definida la extensión debemos preguntarnos dos cosas: ¿La extensión realizada, permite definir de forma más cómoda problemas con diagramas de estados y transiciones entre estados con dependencia geográfica o numérica de algún tipo? y ¿Esta extensión aumenta la expresividad del modelo original impidiendo la decibilidad de propiedades?

Veamos pues el ejemplo de la sección anterior visto bajo el prisma de este nuevo modelo. En esta ocasión, tendremos sólo como estados del vehículo aquellos que indican qué función está realizando en cada momento. La Figura 3.6 muestra el diagrama de estados y transiciones resultante.

Está claro que sólo se ven acciones que relacionen el paso de una actividad a otra independientemente de la situación geográfica del autómata. Esta información está presente en las reglas de transición que se definen a continuación. Internamente, el autómata MSL tiene un atributo numérico denominado *place* para identificar su posición. El rango de valores posibles es {0, 1, 2} y se usa para limitar las acciones posibles en las guardas.

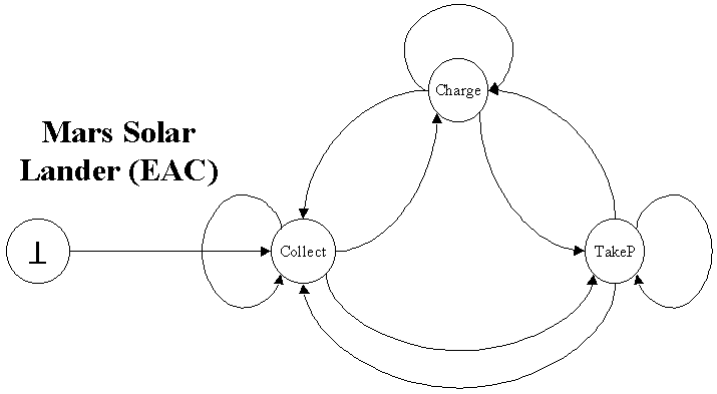


Figura 3.6: El autómata MSL versión ACE

```
name: Collect_to_Charge_stopped
 synch:   MSLander.co_ch()
 guard:   MSLander.place ⩽ 2
 update:  null

name: Collect_to_TakeP_stopped
 synch:   MSLander.co_ta()
 guard:   MSLander.place ⩾ 2
 update:  null
```

```
name: Collect_to_Charge_going
 synch:   MSLander.co_ch()
 guard:   MSLander.place = 1
 update:  MSLander.place += 1

name: Collect_to_Charge_returning
 synch:   MSLander.co_ch()
 guard:   MSLander.place >1
 update:  MSLander.place -= 1

name: Collect_to_TakeP_going
 synch:   MSLander.co_ta()
 guard:   MSLander.place <3
 update:  MSLander.place += 1

name: Collect_to_TakeP_returning
 synch:   MSLander.co_ta()
 guard:   MSLander.place = 3
 update:  MSLander.place -= 1

name: Collect_to_Collect_going
 synch:   MSLander.co_co()
 guard:   MSLander.place <3
 update:  MSLander.place += 1

name: Collect_to_Collect_returning
 synch:   MSLander.co_co()
 guard:   MSLander.place >1
 update:  MSLander.place -= 1

name: Charge_to_Collect_stopped
 synch:   MSLander.ch_co()
 guard:   null
 update:  null

name: Charge_to_TakeP_stopped
 synch:   MSLander.ch_ta()
 guard:   MSLander.place = 2
 update:  null

name: Charge_to_Collect_going
 synch:   MSLander.ch_co()
 guard:   null
 update:  MSLander.place += 1

name: Charge_to_Collect_returning
 synch:   MSLander.ch_co()
 guard:   MSLander.place = 2
 update:  MSLander.place -= 1
```

```
name: Charge_to_TakeP_going
 synch:   MSLander.ch_ta()
 guard:   null
 update:  MSLander.place += 1

name: TakeP_to_Charge_returning
 synch:   MSLander.ta_ch()
 guard:   null
 update:  MSLander.place -= 1

name: Charge_to_Charge_going
 synch:   MSLander.ch_ch()
 guard:   MSLander.place = 1
 update:  MSLander.place += 1

name: Charge_to_Charge_returning
 synch:   MSLander.ch_ch()
 guard:   MSLander.place = 2
 update:  MSLander.place -= 1

name: TakeP_to_ Collect_stopped
 synch:   MSLander.ta_co()
 guard:   null
 update:  null

name: TakeP_to_ Charge_stopped
 synch:   MSLander.ta_ch()
 guard:   MSLander.place = 2
 update:  null

name: TakeP_to_Collect_going
 synch:   MSLander.ta_co()
 guard:   MSLander.place = 2
 update:  MSLander.place += 1

name: TakeP_to_TakeP_going
 synch:   MSLander.ta_ta()
 guard:   MSLander.place = 2
 update:  MSLander.place += 1

name: TakeP_to_TakeP_returning
 synch:   MSLander.ta_ta()
 guard:   MSLander.place = 3
 update:  MSLander.place -= 1
```

Como puede verse, con este sistema tenemos que las reglas son 22 frente a las 34 del modelo original y el diagrama de transición de estados es evidentemente más sencillo y clarificador.

Si podemos demostrar que el modelo extendido es equivalente al original, tendremos las caractersticas deseables de aquél, más la facilidad de representación de éste. Este es el objetivo de la siguiente sección.

3.2.3. Transformación de un sistema ACE en uno AC

La propiedad de acotamiento en el modelo de Autómatas Cooperativos es decidible en el nivel de tareas, que contiene únicamente autómatas con, como máximo, un atributo. No obstante, se encontrarán problemas adicionales si se habla del modelo extendido, puesto que el árbol de derivación ya no depende sólo de los atributos de tarea sino también de los atributos numéricos. Demostrando que todo sistema extendido es a su vez una versión más legible de un sistema de Autómatas Cooperativos y sabiendo que un sistema original es por definición también uno extendido, se pueden trasladar las características del segundo sobre el primero. Se demuestra la equivalencia de ambos sistemas considerando cada posible valor de los atributos numéricos como parte del grafo de estados y aplicando el siguiente algoritmo:

Algoritmo:

1. Repetir:

   - Para toda regla R en cuya guarda aparezca una expresión sobre un atributo numérico del tipo $''A.u == B.v''$, $''A.u! = B.v''$, $''A.u >= B.v''$, $''A.u > B.v''$, $''A.u < B.v''$ o $''A.u <= B.v''$ y/o una actualización del tipo $''A.u = B.v''$, $''A.u+ = B.v''$, $''A.u- = B.v''$, (donde A, B sean nombres de Autómatas y u, v sean atributos numéricos de dichos autómatas respectivos) es decir, aquellas expresiones que en la parte izquierda no tengan una constante sino un valor de atributo numérico,
     - generar tantas reglas como valores posibles tenga el rango del atributo v en el autómata B, de manera que en la parte izquierda de las reglas que contenían el atributo $B.y$ sólo aparezcan constantes.
     - eliminar la regla que hemos expandido en las anteriores.
     - eliminar las reglas que aparezcan con guardas imposibles (por ejemplo $A.x == 5 + A.x == 4$)

   hasta que no queden atributos numéricos en la parte izquierda de una guarda o actualización.

2. Para cada autómata con atributos numéricos hacer:

   Sea $\Sigma = \{s_1, ..., s_m\}$ el conjunto de acciones del autómata y $R = \{r_1, ..., r_p\}$ el conjunto de reglas en el que se ve involucrado.

3. Para cada atributo numérico u, suponiendo $min \leq u \leq max$ con $min, max \in Integer$:

4. Construir, con el conjunto de estados $Q = \{\perp\} \cup \{q_1, ..., q_n\}$, un nuevo conjunto de estados Q' de un nuevo autómata AC sin el atributo u como sigue: $Q' = \{\perp\} \cup \{q_{1_{\text{mín}}}, ..., q_{n_{max}}, ..., q_{1_{\text{mín}}}, ..., q_{n_{\text{máx}}}\}$.

5. Sea $\Sigma' = \Sigma'_1 \cup \Sigma'_2 \cup \Sigma'_3$ como sigue $\Sigma'_1 = \{s'_\perp\}$ donde $\perp \xrightarrow{s_\perp} q_{ini_k}$ que es el estado inicial correspondiente a la inicialización del atributo y $\Sigma'_2 = \emptyset$ y $\Sigma'_3 = \emptyset$.

6. Sea $R' = \{r'_{ini}\}$ donde r'_{ini} se construye cambiando en r_{ini} la acción $s_\perp$ con la acción $s'_\perp$ en la regla inicial.

7. Para cada acción $s_\imath$, que se ejecute en una regla r_j pero donde el atributo no aparezca en la guarda ni en la actualización:

   - añadir al conjunto Σ'_1 la acción $s'_\imath$ tal que si teníamos la transición $q_a \xrightarrow{s_\imath} q_b$ ahora tendremos $q_{a_k} \xrightarrow{s'_\imath} q_{b_k} : \forall k : min \leq k \leq max$.

   - añadir a R' la regla r'_j obtenida tras cambiar en r_j la acción $s_\imath$ con $s'_\imath$.

8. Para cada acción $s_\imath$, que se ejecute en una regla r_j pero donde el atributo no aparezca en la guarda pero sí en la parte izquierda de la actualización:

   - añadir al conjunto Σ'_2 la acción $s^\alpha_\imath$ tal que si teníamos la transición $q_a \xrightarrow{s_\imath} q_b$ ahora tendremos $q_{a_k} \xrightarrow{s^\alpha_\imath} q_{b_{k+\alpha}} : \forall k : min \leq k \leq max - \alpha$ (o respectivamente, $\forall k : min + \alpha \leq k \leq max$).

   - añadir a R' la regla r'_j obtenida tras cambiar en r_j la acción $s_\imath$ con $s^\alpha_\imath$.

9. Para cada acción $s_\imath$, que se ejecute en una regla r_j pero donde el atributo aparezca en la guarda como «==x» o «!=x»:

   - añadir al conjunto Σ'_3 la acción $s_{i==x}$ (o respectivamente !=x) tal que si teníamos la transición $q_a \xrightarrow{s_\imath} q_b$ ahora tendremos $q_{a_x} \xrightarrow{s_{i==x}} q_{b_{x+\alpha}}$ (o respectivamente, !=x) donde α es el valor de la constante añadida en la actualización (si es que existe).

   - añadir a R' la regla r'_j obtenida tras cambiar en r_j la acción $s_\imath$ con $s_{i==x}$ (o respectivamente, !=x).

10. Para cada acción $s_\imath$, que se ejecute en una regla r_j pero donde el atributo aparezca en la guarda como «¡=x», «¡x», «¿x», o «¿=x»:

    - proceder como en el paso 7 pero añadiendo al conjunto Σ'_3 la acción $s_{i_{comp}}$ (con comp $\in$ {«¡=x», «¡x», «¿x», «¿=x»}) y tal que si teníamos la transición $q_a \xrightarrow{s_\imath} q_b$ ahora tendremos $q_{a_y} \xrightarrow{s_{i_{comp}}} q_{b_{y+\alpha}}$ para valores válidos de y en cada caso y en cada actualización (sin salirnos de rango).

    - añadir a R' la regla r'_j obtenida tras cambiar en r_j la acción $s_\imath$ con $s_{i_{comp}}$.

11. Cambiar el autómata original y sus reglas ACE por el nuevo autómata y las nuevas reglas AC.

12. Iterar con el siguiente atributo volviendo al paso 2 o con el siguiente autómata del sistema volviendo al paso 1.

Veamos ahora un ejemplo de transformación de un sistema genérico y sencillo de Autómatas Cooperativos Extendidos (pero que incluye la mayora de los casos) en otro equivalente de Autómatas Cooperativos. Sean *Aut1* y *Aut2* los autómatas de la Figura 3.7 y las reglas siguientes:

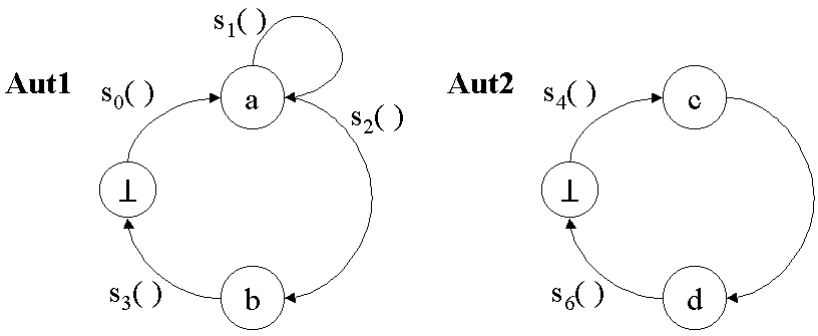


Figura 3.7: Sistema de Autómatas Cooperativos Extendidos

```
name: Rule1
 synch:   Aut1.s0()
          + Aut2.s4()
 guard:   null
 update:  Aut1.V = 0

name: Rule2
 synch:   Aut1.s1()
 guard:   null
 update:  Aut1.V += 1

name: Rule3
 synch:   Aut1.s2()
          + Aut2.s5()
 guard:   Aut1.V = 2
 update:  null

name: Rule4
 synch:   Aut1.s2()
          + Aut2.s5()
 guard:   Aut1.V != 2
 update:  null

name: Rule5
 synch:   Aut1.s3()
          + Aut2.s6()
 guard:   null
 update:  null
```

Este ejemplo contiene dos autómatas y cinco reglas. A su vez, el autómata *Aut1* contiene un atributo numérico que puede tomar diferentes valores. Por simplicidad reducimos el ejemplo a que pueda tomar valores entre 0 y 3 ambos incluidos.

El conjunto de estados del autómata incluye tres estados: el estado inicial *dummy*, el estado a y el estado b. As pues, $Q = \{\perp\} \cup \{a, b\}$.

Aplicando el algoritmo sobre el conjunto de estados, obtenemos un nuevo conjunto de estados de un autómata sin atributos numéricos $Q' = \{a_0,\ a_1,\ a_2,\ a_3,\ b_0,\ b_1,\ b_2,\ b_3\}$.

Otro tanto hacemos con el conjunto de acciones $\Sigma = \{s_0, s_1, s_2, s_3\}$ obteniendo los conjuntos $\Sigma'_1 = \{s'_0, s'_3\}$, $\Sigma'_2 = \{s^1_1\}$ y $\Sigma'_3 = \{s_{2_2}, s_{2_{!2}}\}$.

El autómata resultante puede verse en la Figura 3.8 y las reglas obtenidas aplicando el algoritmo son las siguientes:

```
name: Rule1
 synch:   Aut1'.s'0()
          + Aut2.s4()
 guard:   null
 update:  null

name: Rule3
 synch:   Aut1'.s2_2()
          + Aut2.s5()
 guard:   null
 update:  null

name: Rule4
 synch:   Aut1'.s2_!2()
          + Aut2.s5()
 guard:   null
 update:  null

name: Rule5
 synch:   Aut1'.s'3()
          + Aut2.s6()
 guard:   null
 update:  null

name: Rule2
 synch:   Aut1'.s1^1()
 guard:   null
 update:  null
```

En este caso, el número total de reglas es el mismo, pero este número será normalmente mucho mayor, del mismo modo que se incrementa el número de acciones y de estados.

Limitando el rango de valores posibles de los atributos numéricos obtenemos un número finito de estados en el modelo de AC con la misma expresividad del original ACE. El modelo de ACE sólo decrementa la complejidad del modelo resultante en aquellos casos en los que es importante reflejar las posiciones relativas de un autómata entre otros sin cambiar la capacidad expresiva con respecto a la versión sin esta extensión. Esta disminución de la complejidad es importante en aquellos sistemas en los que tenemos que comparar posiciones de un autómata con otros o con un valor

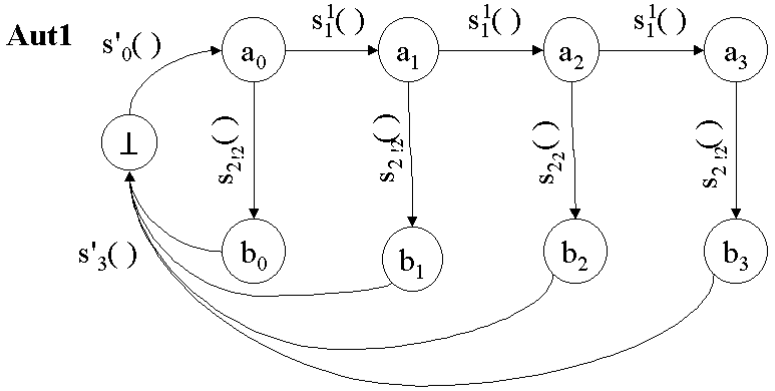


Figura 3.8: Autómata Cooperativo sin atributos numéricos

dado, lo cual es trivial en un sistema ACE pero incrementa exponencialmente la complejidad, al menos la complejidad visual, del modelo AC.

Puede concluirse, por tanto, que el modelo extendido es igual de expresivo que el original de los Autómatas Cooperativos, y, entonces, trasladar las conclusiones acerca de la decibilidad de propiedades obtenidas en el modelo de original al modelo extendido, por ejemplo como ocurre con la propiedad de acotamiento.

Proposición 3 *La propiedad de acotamiento de un sistema de Autómatas Cooperativos Extendidos con un solo atributo identificador es decidible.*

3.2.4. Un ejemplo práctico

En esta sección presentamos un ejemplo completo de la utilización y justificación de nuestra extensión de los autómatas cooperativos. En este caso no sólo nos ceñiremos a un objeto como en el caso del ejemplo comentado en la Figura 3.4, sino que planteamos un sistema completo con cuatro tipos de autómatas, que son definidos genéricamente en la medida de lo posible, pero que muestran serias dependencias con respecto a su funcionamiento basadas en su posicionamiento geográfico y en determinados valores numéricos.

El ejemplo se va a plantear inicialmente bajo el prisma de los autómatas cooperativos, con la problemática y complejidad que ello conlleva y trasladado después al ámbito de los autómatas cooperativos extendidos para, de ese modo, mostrar las ventajas que aporta nuestra extensión en cuanto a la modelización sencilla de sistemas complejos.

Supongamos que debemos modelizar un sistema de montaje en una fábrica. Tenemos una serie de objetos a construir que constan de una serie de piezas colocadas en un cierto orden. Para ello, tenemos dispuesta una cinta transportadora que traslada cada objeto entre los robots que deben poner dichas piezas. Asimismo tenemos robots situados en una cierta ubicación, junto con contenedores de piezas. Si alguno de estos

contenedores quedara vacío un robot auxiliar que se desplaza por la fábrica repone dichas piezas desde el almacén general. Algo parecido a lo que podría representarse según la Figura 3.25

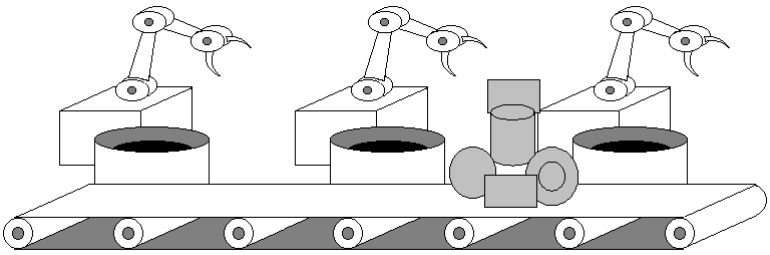


Figura 3.9: Cinta, robots, contenedores y objetos

El planteamiento es sencillo, sobre todo si los objetos admiten sólo un tipo de montaje. Por ejemplo, si tuviéramos un objeto simple como el de la Figura 3.10.en el que se representan distintos tipos de piezas (con formas y números diferentes) y un orden de acoplamiento, representado en este caso por pa superposición de piezas, siendo las más profundas las que se considera deben ser colocadas primero.

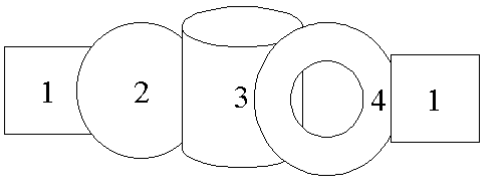


Figura 3.10: Un objeto sencillo ya construidos

La representación de las fases de construcción de un objeto de este tipo mediante un autómata debería representar la posición en la que se encuentra el objeto, la fase de construcción y las distintas transiciones, ya sean movimientos en la cinta o colocaciones de piezas. Un ejemplo de estas fases sería el autómata que aparece en la figura 3.11:

Como puede verse, tendríamos los distintos estados que representan la llegada de un objeto a un lugar, la culminación de una fase (entendiendola como el acoplamiento de una pieza concreta) y las acciones estarían referidas a movimientos concretos y a piezas concretas. No obstante, y como debemos conocer de antemano el número de piezas, el orden y las posiciones que ocupan, el modelo resultante está absolutamente dirigido por el autómata y no existe ninguna genericidad. Por ejemplo, este autómata sería completamente distinto de otro que aún conteniendo también las mismas piezas las admitiera en otro orden y de una manera más compleja. Así en el caso de un objeto no tan sencillo como el anterior sino con alternativas a la hora de situar las piezas como el de la Figura 3.26, podría complicar el autómata resultante considerablemente, como puede verse en la Figura 3.13.

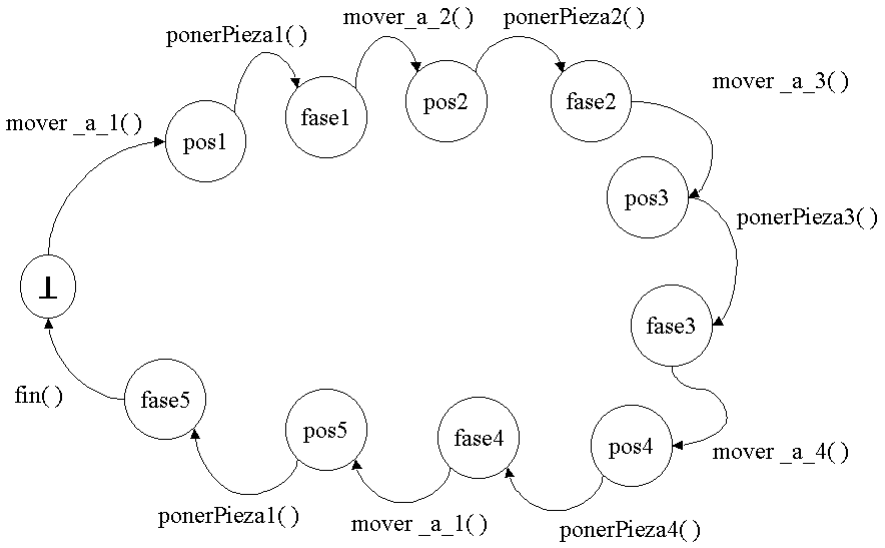


Figura 3.11: Representación del objeto sencillo según AC

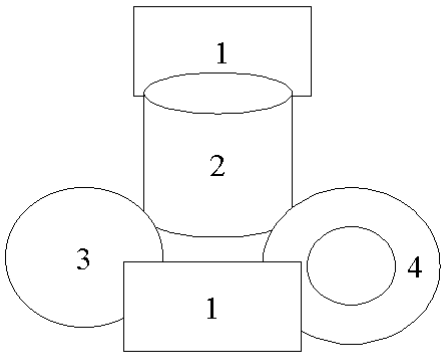


Figura 3.12: Un objeto con varias posibilidades de montaje

En este objeto queda claro que si bien la pieza número uno va en primer lugar y la número dos en segundo, luego tenemos dos caminos posibles, esto es, poner la pieza tres y luego la cuatro o al contrario, la cuatro y luego la tres, para finlamente acabar colocando la pieza 1.

Un Autómata de estas características debe presentar tantos caminos posibles como alternativas de montaje aparezcan en el modelo. Si hubiera más posibles combinaciones el resultado sería ilegible. En el ejemplo que nos ocupa, inicialmente el automata que representa al objeto complejo en construcción se sitúa inicialmente en el estado pos1, después se le coloca la pieza uno concluyendo de esta forma la fase 1 y situándonos en ese mismo estao. Tras una acción de cambio de estado vamos al único estado

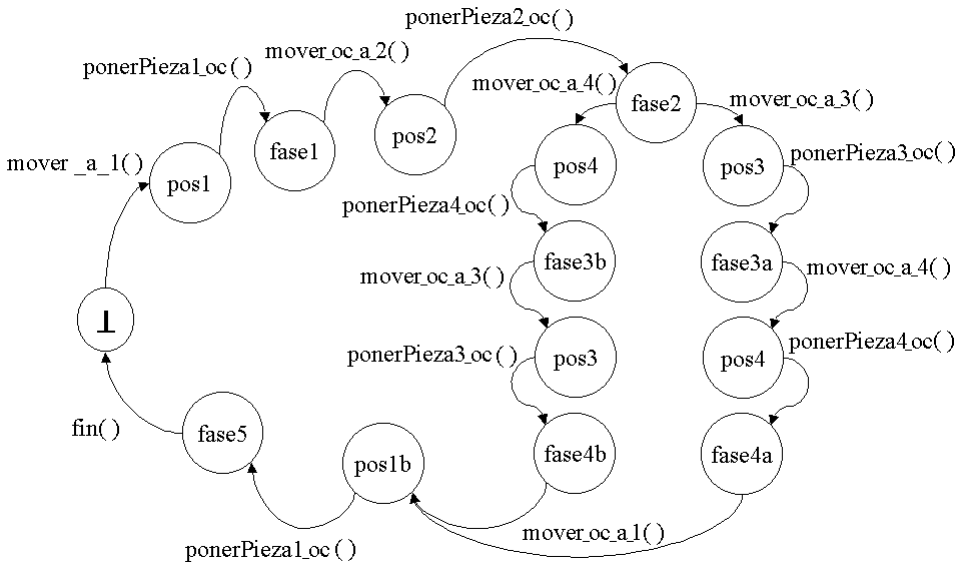


Figura 3.13: AC de las fases de diseño del objeto complejo

alcanzable desde aquí, pos2, y tras colocar la pieza 2 llegamos al estado fase 2 y nos encontramos con con la alternativa de movernos a la posición 3 o a la posición 4. Si optamos por la tres deberemos poner la pieza 3 y luego desplazarnos a la posición 4 para poner la pieza 4 y si optamos por la posición 4 deberemos poner la pieza 4, desplazarnos a la 3 y luego poner la pieza 3. En cualquiera de los dos casos sólo quedará como posible movimiento el desplazar el objeto a la posición 1 de nuevo y colocar la última pieza.

Hasta este momento, el hecho de presentar distintas posiciones no ha provocado mayor inconveniente que la simple aparición de estados especiales que indican los distintos valores posibles de la posición del objeto, y por tanto, las operaciones asociadas a los distintos cambios de estados como cambios de ubicación, sin embargo, a la hora de representar alternativas en el orden de ejecución de ciertas acciones el aspecto del autómata resulta cada vez más complicado.

No obstante, dicha representación sigue siendo posible, aunque complicada de entender de un vistazo. Esta complicación es más evidente si cabe, en el caso de la representación del resto de agentes que intervienen en el modelo, y, sobre todo, si queremos representarlos de forma genérica. Nos referimos a los agentes Robot y Contenedor de piezas. Teniendo en cuanta que la ejecución del autómata "objeto" requiere un conocimiento exacto de la posición de cada pieza y de su identificación, el hecho de definir genéricamente un robot o un contenedor de piezas, requiere elegir entre un tipo de ejecución y otro (con nombres distintos de acción y de estado) dependiendo de la ubicación inicial de cada agente.

Un contendor genérico de piezas (de los cuatro tipos que admitirían los autómatas "objeto" antes mencionados) debería ser como el representado en la Figura 3.14

Asimismo, el agente "Robot" genérico también tendrá un aspecto similar (véase

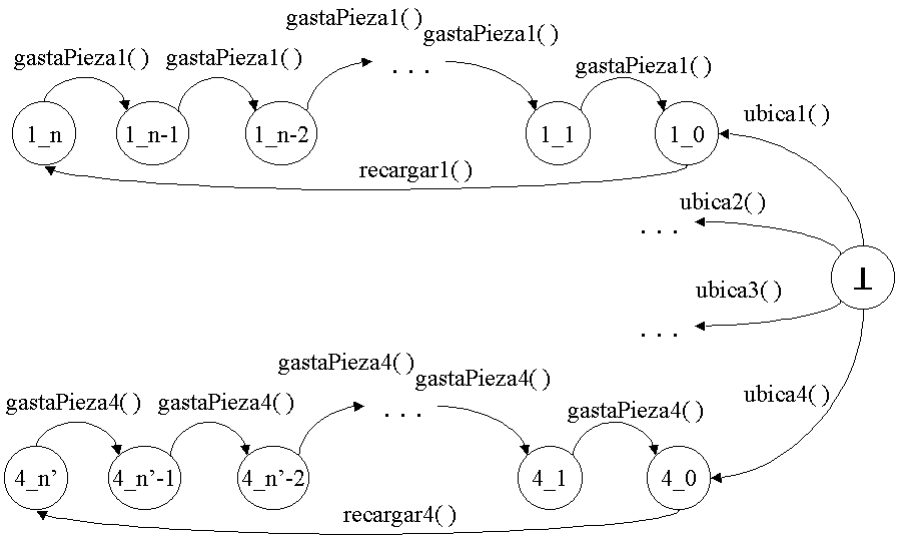


Figura 3.14: Representación del agente contenedor de piezas genérico

Figura 3.15), dependiendo del lugar donde se ubique en el momento de su creación. Nótese que pueden existir distintos robots en la misma posición de la cadena de montaje y que el objeto podrá sincronizar con cualquiera de ellos siempre y cuando tenga una pieza preparada para su inserción.

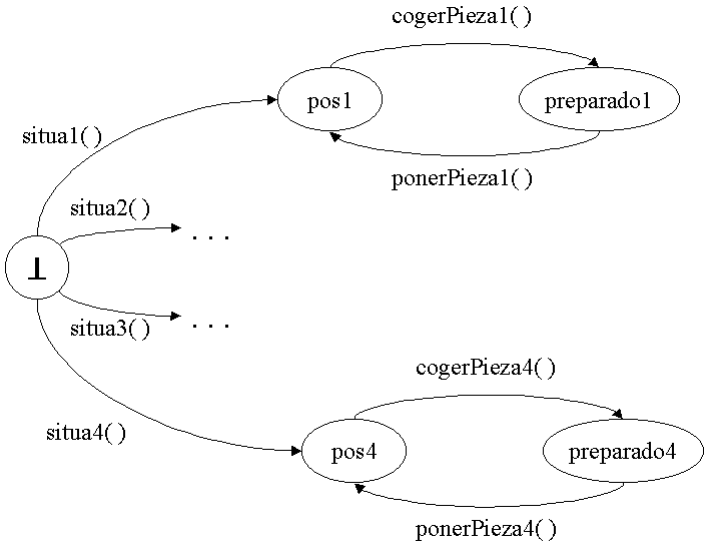


Figura 3.15: Autómata Robot bajo la perspectiva de los Autómatas Cooperativos

Finalmente, y dado que no está asociado a ningun lugar en concreto el autómata

correspondiente al reponedor de piezas es el más sencillo de todos, resultando como puede verse en la Figura 3.16.

Este autómata permanece idle mientras no realiza acciones.En las sincronizaciones con los contenedores de piezas ejecuta la acción "reponer_pieza()" que conlleva una transición, pero no un cambio de estado. Es posible complicar este autómata complicando el modelo con alternativas como hallarse ubicado en una linea de montaje concreta y no responder a otras (incluyendo un atributo "linea" que indicase su pertenencia a una tarea) haciéndolo sincronizarse con un almacén general que llevase la contabilidad del total de piezas (que sería muy complejo de mostrar a no ser que le incluyésemos atributos) etc.

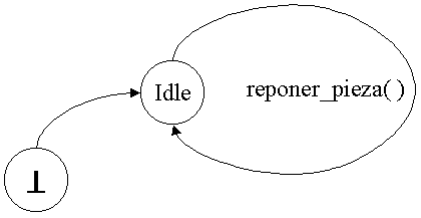


Figura 3.16: Autómata reponedor, no sujeto a posición alguna

En cuanto a las reglas de transición (y considerando que no vamos a usar de momento más que el modelo básico, sin atributos) serían simples pero muy abundantes, pues deberían representar todos y cada uno de los movimientos y colocaciones de piezas (uno por ubicación) más las capturas y reposiciones de piezas (también una por ubicación). Si consideráramos la existencia de varias cadenas de montaje nos veríamos obligados a incluir el modelo extendido (con un atributo representando a cada cinta o tarea) pudiendo entonces competir por el uso de uno o varios autómatas, reponedores o contenedores. O podríamos considerar la posibilidad de tener múltiples cadenas de montaje que pudiesen unirse o colaborar entre sí, necesitando entonces del modelo completo (con dos o más atributos) para representar el modelo.

Las reglas de transición del modelo básico para objetos complejos OC serían, considerando un único objeto en la cinta transportadora, las siguientes:

Inicialmente tendríamos las constructoras que llevarían los autómatas del estado "muerto" a uno de los estados válidos de su diagrama de estados. Esta inicialización, podría sincronizarse para que todos estuvieran creados simultáneamente o podría ser individualizada para el objeto, el reponedor, los robots y los contenedores. Dado que no es preciso que se creen nuevos atributos ni se hagan actualizaciones sincrónicas, y que el objeto sólo tiene una posibiolidad de creación que es la transacción "Movimiento_1" descrita más adelante, es lógico que la creación de autómatas en el sistema sea individual y no colectiva. Dado que las transiciones sólo consistirán en la elección de una primera acción y que en el caso de los Robots y Contenedores son una elección determinista tendremos cuatro posibles creaciones de estos autómatas:

Para los Robots:

```
name: NewRobot_posicion1
 synch:   Robot.situa1()
 guard:   null
 update:  null

name: NewRobot_posicion2
 synch:   Robot.situa2()
 guard:   null
 update:  null

name: NewRobot_posicion3
 synch:   Robot.situa3()
 guard:   null
 update:  null

name: NewRobot_posicion4
 synch:   Robot.situa4()
 guard:   null
 update:  null
```

Y otro tanto ocurre para los contenedores:

```
name: NewCont_posicion1
 synch:   Contenedor.ubica1()
 guard:   null
 update:  null

name: NewCont_posicion2
 synch:   Contenedor.ubica2()
 guard:   null
 update:  null

name: NewCont_posicion3
 synch:   Contenedor.ubica3()
 guard:   null
 update:  null

name: NewCont_posicion4
 synch:   Contenedor.ubica4()
 guard:   null
 update:  null
```

Para el reponedor y dado que su existencia es necesaria para el funcionamiento de todos los demás (los contenedores se crean vacíos y se deben reponer las piezas, y sin piezas los robots no pueden prepararse, y sin robots preparados el objeto no puede cambiar de fase) su acción inicial debe ser previa a las demás, así que supondremos que se realiza de forma automática al comenzar con la ejecución del modelo.

Tras las reglas de construcción de los robots y contenedores, veamos ahora las reglas que permiten los movimientos. Dado que carecemos de un verdadero sistema de posicionamiento más allá de la existencia de estados especiales del autómata OC (objeto complejo) deberemos tener una regla por cada posible movimiento a una posición. Partiendo de un sistema con, inicialmente cuatro lugares tendremos cuatro reglas sin guardas ni actualizaciones. Sólo tendremos guardas y actualizaciones si ampliamos el problema a varias cintas transportadoras, en cuyo caso añadiremos un atributo al OC para indicar su pertenencia a una de las cintas (tarea)

```
name: Movimiento_oc_1
 synch:   OC.mover_oc_a_1()
 guard:   null
 update:  null

name: Movimiento_oc_2
 synch:   OC.mover_oc_a_2()
 guard:   null
 update:  null

name: Movimiento_oc_3
 synch: OC.mover_oc_a_3()
 guard: null
 update: null

name: Movimiento_oc_4
 synch:   OC.mover_oc_a_4()
 guard:   null
 update:  null
```

Para el objeto simple, los movimientos serían los mismos, no obstante y dado que debemos mostrar en las reglas de transición qué objetos

Una vez definidas las reglas de transición de estados para representar los movimientos del objeto entre distintos lugares, debemos definir las reglas que indican interacción entre el robot de una determinada posición y el objeto. Evidentemente es preciso concretar qué pieza está acoplando el robot y por tanto tenemos una acción específica para esa pieza en el OC sincronizada con la acción específica equivalente del Robot. Esto nos obliga, como ya se ha visto a tener un autómata Robot muy grande, pues debe replicarse la funcionalidad básica del mismo para permitir sincronizar específicamente con una pieza en concreto situada en una ubicación en concreto. Es por tanto la idea de "posición geográfica" la que de nuevo nos produce el problema del incremento de estados y la complicación del autómata..

En este caso, como los atributos clásicos son considerados identificadores de tareas y creados en tiempo de ejecución, y dado que es preciso conocer específicamente la ubicación de la pieza y del robot para poder realizar la sincronización deberíamos tener un atributo por pieza, alcanzando la complejidad de la máquina de Turing en el modelo. Como es posible modelizarlo sin estos atributos, no tiene sentido utilizarlos y perder de esa manera la decibilidad depropiedades. Por esto, tenemos las cuatro

reglas de transición para acoplar una pieza en el objeto, una por cada pieza o posición posible.

```
name: Acoplar_pieza_1
 synch:   OC.ponerPieza1_oc()
          + Robot.ponerPieza1()
 guard:   null
 update:  null

name: Acoplar_pieza_2
 synch:   OC.ponerPieza2_oc()
          + Robot.ponerPieza2()
 guard:   null
 update:  null

name: Acoplar_pieza_3
 synch:   OC.ponerPieza3_oc()
          + Robot.ponerPieza3()
 guard:   null
 update:  null

name: Acoplar_pieza_4
 synch:   OC.ponerPieza4_oc()
          + Robot.ponerPieza4()
 guard:   null
 update:  null
```

Una vez el robot ha colocado una pieza en el objeto, debe recoger una nueva pieza para volver al estado "preparado" para ello, debe sincronizar con el contenedor de piezas de la misma posición (otra vez tendremos el autómata con un aspecto de ser una elección de múltiples autómatas más sencillos) y este contenedor debe gastar una de sus piezas acercándose al estado en el que se quede sin ellas.

```
name: Preparar_1
 synch:   Contenedor.gastarPieza1()
          + Robot.cogerPieza1()
 guard:   null
 update:  null

name: Preparar_2
 synch:   Contenedor.gastarPieza2()
          + Robot.cogerPieza2()
 guard:   null
 update:  null

name: Preparar_3
 synch:   Contenedor.gastarPieza3()
          + Robot.cogerPieza3()
 guard:   null
 update:  null
```

```
name: Preparar_4
 synch:   Contenedor.gastarPieza4()
          + Robot.cogerPieza4()
 guard:   null
 update:  null
```

Por último, cuando un contenedor queda sin piezas debe recargarse para que el robot asociado pueda obtener una nueva pieza mediante la sincronización con la acción "gastarPieza()" del contenedor, y para ello aquí presentamos las últimas cuatro reglas de transición.

```
name: Reponer_1
 synch:   Contenedor.recargar1()
          + Reponedor.reponer()
 guard:   null
 update:  null

name: Reponer_2
 synch:   Contenedor.recargar2()
          + Reponedor.reponer()
 guard:   null
 update:  null

name: Reponer_3
 synch:   Contenedor.recargar3()
          + Reponedor.reponer()
 guard:   null
 update:  null

name: Reponer_4
 synch:   Contenedor.recargar4()
          + Reponedor.reponer()
 guard:   null
 update:  null
```

Frente a esto, vamos a ver el modelo resultante bajo la perspectiva de los autómatas cooperativos extendidos. Dado que ambos objetos tienen el mismo número de piezas en su construcción, ambos tendrán la misma forma. Tanto el objeto simple como el complejo (Figura 3.17) tienen un aspecto menos complicado por dos motivos. El primero es la eliminación de los estados que representan los lugares. El segundo es que al pasar el control de las transacciones a las guardas de las reglas en lugar de al autómata ya no es preciso mantener los dos caminos separados en el caso del complejo y por tanto tenemos un único autómata de cinco piezas. En realidad, el objeto simple tendrá el mismo diagrama de estados que el complejo, variando sólo la denominación de las acciones para poder distinguirlos entre sí. La verdadera diferencia entre ellos será que tendremos como guarda la obligación de haber colocado la pieza anterior en el caso del simple y en el caso del objeto complejo, tendremos además que controlar el número de piezas de tipo tres y cuatro para no excedernos.

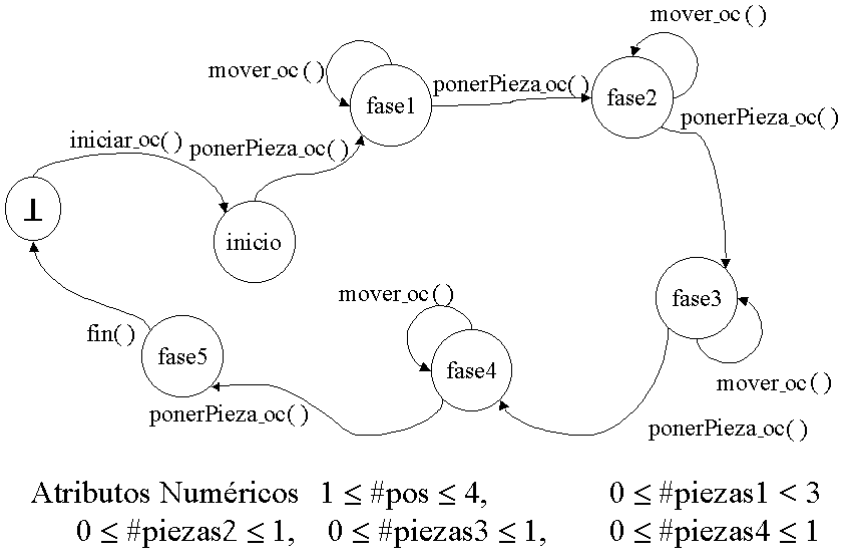


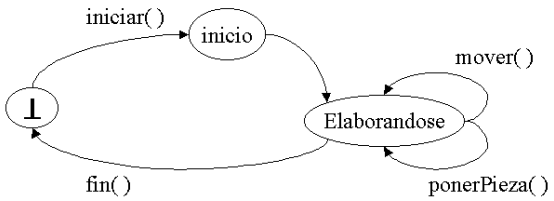
Figura 3.17: Objeto Complejo bajo la perspectiva ACE

En este objeto tendremos varios atributos numéricos, que representarán el número de piezas introducidas en cada fase y la posición que ocupa el objeto en este momento. El motivo es que al tener varias posibilidades de montaje no debemos permitir, por ejemplo poner más de una pieza de tipo 3 aunque exista aparentemente un camino en el autómata, dado que no se especifica que tipo de pieza se sitúa. Esto lo conseguiremos con las guardas, de manera que sólo si se cumple que no hayamos introducido ninguna pieza de tipo 3 y sí de uno y dos entonces podemos introducir la tercera.

Es más, aun perdiendo la perspectiva visual, es posible representar de una forma mucho más simple el autómata, dado que no existe diferencia (al menos en cuanto a las acciones involucradas) entre la fase 2 y la fase 3 por ejemplo. Si representásemos textualmente el autómata de la Figura 3.17 veríamos esta repetición:

```
Automaton OcACE
{
    dead ->inicio : iniciar_oc()
    inicio ->fase1 : ponerPieza_oc()
    fase1 ->fase2 : ponerPieza_oc()
    fase1 ->fase1 : mover_oc()
    fase2 ->fase3 : ponerPieza_oc()
    fase2 ->fase2 : mover_oc()
    fase3 ->fase4 : ponerPieza_oc()
    fase3 ->fase3 : mover_oc()
    fase4 ->fase5 : ponerPieza_oc()
    fase4 ->fase4 : mover_oc()
    fase5 ->dead : fin()
}
```

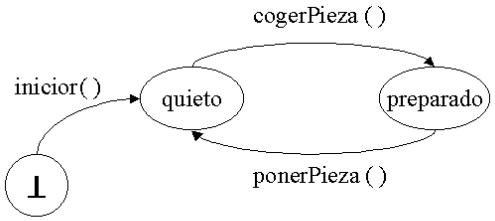
El autómata evitando estas repeticiones sería tan simple como el de la Figura 3.18 y todo el control quedaría en manos de las reglas (que no variarían), pero evidentemente perderíamos la perspectiva visual que indica el número de piezas y las fases por las que pasa el autómata. De ahí que utilicemos la versión con repeticiones.



Atributos Numéricos $1 \leq \#pos \leq 4$, $0 \leq \#piezas1 \leq 2$
$0 \leq \#piezas2 \leq 1$ $0 \leq \#piezas3 \leq 1$, $0 \leq \#piezas4 \leq 1$

Figura 3.18: Objeto Complejo en su versión más reducida según la perspectiva ACE

Por otro lado, el autómata correspondiente al robot (Figura), será mucho más simple que el anterior, pues la ubicación ha pasado de ser un estado en si mismo a ser el atributo numérico #pos. Es por ello que el robot será un autómata que sólo incluirá una transición entre los estados de quieto a preparado, sincronizandose con el contenedor de piezas para obtener el estado de preparado y con el objeto para regresar al estado quieto.



Atributos Numéricos $0 \leq \#pos \leq n$,

Figura 3.19: Robot bajo la perspectiva ACE

El autómata contenedor (véase Figura 3.20) tampoco deberá duplicarse para identificar por separado a las piezas, puesto que al disponer de un atributo numérico #pos será más simple, como sucede con el Robot ACE. Además, en lugar de mostrar explícitamente el número de piezas que quedan, es más sencillo considerar dicho número como otro atributo numérico en este caso #cantidad, que al llegar a 0 (sincronizando con el robot cuando coja una pieza) provocará un cambio de estado hacia el estado vacío.

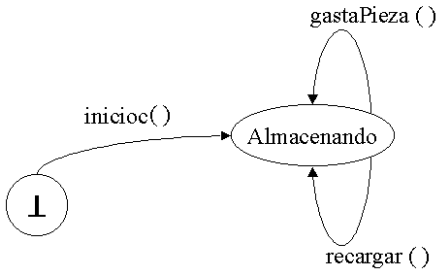


Figura 3.20: Autómata contenedor de piezas bajo la perspectiva ACE

En cuanto al autómata reponedor y dado que no estaba sujeto a limitaciones de posición geográfica ni de número de piezas o de cantidades de ningún tipo, es absolutamente independiente de los atributos numéricos utilizados y por tanto no requiere modificación alguna.

La construcción de los autómatas desde el estado *dead* al estado inicial será muy parecida a la anterior, pues no olvidemos que pese a que hemos reducido el número de estados visibles en los autómatas el número de estados subyacentes por la introducción de atributos numéricos es el mismo.

Los robots se construirán de la siguiente manera:

```
name: NewRobot_posicion1
 synch:   Robot.inicior()
 guard:   null
 update:  Robot.#pos = 1

name: NewRobot_posicion2
 synch:   Robot.inicior()
 guard:   null
 update:  Robot.#pos = 2

name: NewRobot_posicion3
 synch:   Robot.inicior()
 guard:   null
 update:  Robot.#pos = 3

name: NewRobot_posicion4
 synch:   Robot.inicior()
 guard:   null
 update:  Robot.#pos = 4
```

Y otro tanto ocurre para los contenedores:

```
name: NewCont_posicion1
 synch:   Contendor.inicioc()
 guard:   null
 update:  Contenedor.#pos = 1


name: NewCont_posicion2
 synch:   Contendor.inicioc()
 guard:   null
 update:  Contenedor.#pos = 2

name: NewCont_posicion3
 synch:   Contendor.inicioc()
 guard:   null
 update:  Contenedor.#pos = 3

name: NewCont_posicion4
 synch:   Contendor.inicioc()
 guard:   null
 update:  Contenedor.#pos = 4
```

Los movimientos se reducen a 2, ya no es necesario desplazarse a todas las posiciones "directamente" sino que podemos repetir los avances y retrocesos hasta encontrar la posición adecuada. No obstante, pueden mantenerse los movimientos directos de la versión AC, simplemente cambiando los incrementos de las actualizaciones por una asignación de una constante.

```
name: MovimentoAdelante
 synch:   OC.mover_oc()
 guard:   OC.#pos <4
 update:  OC.#pos += 1
name: MovimientoAtras
 synch:   OC.mover_oc()
 guard:   OC.#pos >1
 update:  OC.#pos -= 1
```

En el objeto complejo (OC) las reglas para introducir una pieza tienen que ver con el órden de introducción. La primera pieza de tipo 1 sólo requiere que no hayan piezas de ese tipo introducidas, así pues, la condición es que el objeto esté situado en la posicion adecuada y que el número de piezas sea cero.

```
name: Acoplar_pieza_1a
 synch:   OC.ponerPieza_oc()
          + Robot.ponerPieza()
 guard:   OC.#pos == Robot.#pos
          OC.#pos == 1
          OC.#pieza1 == 0
 update:  OC.#pieza1 += 1
```

Para acoplar la última pieza (que vuelve a ser la pieza 1) debemos asegurarnos que no hemos introducido ya todas las piezas1 (en realidad es obvio que no es así, pues sólo tenemos un número finito de llamadas a ponerPieza()) y que las anteriores (en este caso, pueden ser la 3 o la 4) ya han sido introducidas.

```
name: Acoplar_pieza_1b
 synch:   OC.ponerPieza_oc()
          + Robot.ponerPieza()
 guard:   OC.#pos == Robot.#pos
          OC.#pos == 1
          OC.#pieza1 == 1
          OC.#pieza3 == 1
          OC.#pieza4 == 1
 update:  OC.#pieza1 += 1
```

Nótese que en la primera regla para acoplar la pieza 1 tendríamos como guarda que el número de piezas de tipo uno sea igual a 0, (y evidentemente que el número de piezas de tipos 3 y 4 también, pero esto es innecesario de decirlo) y en el segundo, que el número de piezas de tipo 1 sea igual a 1 y el número de piezas de tipos 3 y 4 también. Es por esto, que se pueden unificar ambas reglas en una sola de la siguiente manera:

```
name: Acoplar_pieza_1
 synch:   OC.ponerPieza_oc()
          + Robot.ponerPieza()
 guard:   OC.#pos == Robot.#pos
          OC.#pos == 1
          OC.#pieza1 == OC.#pieza3
          OC.#pieza3 == OC.#pieza4
 update:  OC.#pieza1 += 1
```

Para insertar la segunda pieza, además de estar en la posición adecuada y no haber introducido ninguna pieza de ese tipo (#piezas2==0) es obligatorio que se haya introducido una y sólo una pieza del tipo 1, si no, estaríamos todavía en la fase anterior.

```
name: Acoplar_pieza_2
 synch:   OC.ponerPieza_oc()
          + Robot.ponerPieza()
 guard:   OC.#pos == Robot.#pos
          OC.#pos == 2
          OC.#pieza2 == 0
          OC.#pieza1 == 1
 update:  OC.#pieza2 += 1
```

Para introducir la tercera o cuarta pieza, sólo hemos de fijarnos en no haberla introducido todavía y en que ya se haya introducido la anterior, que en este caso es la segunda. La existencia de la primera pieza está garantizada por la aparición de la segunda y así sucesivamente si hubiese anteriores..

```
name: Acoplar_pieza_3
 synch:   OC.ponerPieza_oc()
          + Robot.ponerPieza()
 guard:   OC.#pos == Robot.#pos
          OC.#pos == 3
          OC.#pieza3 == 0
          OC.#pieza2 == 1
 update:  OC.#pieza3 += 1
name: Acoplar_pieza_4
 synch:   OC.ponerPieza_oc()
          + Robot.ponerPieza()
 guard:   OC.#pos == Robot.#pos
          OC.#pos == 4
          OC.#pieza4 == 0
          OC.#pieza2 == 1
 update:  OC.#pieza4 += 1
```

Cuando un robot se sincroniza con su contenedor sólo hemos de controlar que estemos en la misma posición y no es preciso conocer de antemano la posición en la que nos encontrábamos.

```
name: Preparar
 synch:   Contenedor.gastarPieza()
          + Robot.cogerPieza()
 guard:   Contenedor.#pos == Robot.#pos
 update:  Contenedor.#cantidad -= 1
```

Y para reponer las piezas sólo hemos de controlar que, independientemente de la posición, la cantidad de piezas esté en 0.

```
name: Reponer
 synch:   Contenedor.recargar()
          + Reponedor.reponer()
 guard:   Contenedor.#cantidad == 0
 update:  Contenedor.#cantidad -= N
```

Como resulta evidente, el número de reglas resultante en la resolución del problema en el caso de los autómatas cooperativos es superior al del caso de los autómatas cooperativos extendidos. Asimismo, la complejidad de estas reglas ha aumentado, pero no ha vuelto ilegible el modelo. Además, en modelos de aparente difícil solución aplicando el algoritmo de transformación podemos encontrar un sistema de Autómatas Cooperativos para expresar sistemas que de otra forma tratando de hacerlos directamente caeríamos en la tentación de utilizar atributos clásicos y perdiendo con ello la decibilidad de propiedades.

3.2.5. los ACE frente a los Team automata

En la sección 3.1.6 se mostró cómo se modelizaba un sistema de permisos de lectura y escritura mediante el modelo de los autómatas Team, con la complejidad

del autómata resultante que representaba a tres autómatas componente del mismo. En esta sección se muestra cómo representar este mismo modelo bajo el punto de vista de los Autómatas Cooperativos y nuestra versión extendida.

Recordando el problema

Se tenía un sistema que consistía en una serie de usuarios en un sistema. Cada usuario tenía asignados unos permisos de acceso, a saber nulo, sólo lectura o lectura y escritura. Además se establecía una jerarquía entre usuarios, de manera que cada usuario era a su vez responsable de otro usuario y podía conceder o revocar los permisos de este.

El problema consistía en que si un usuario con un determinado nivel de permisos concedía este mismo nivel de permisos a su subordinado (y así sucesivamente) en el caso de que el usuario de nivel superior perdiese sus permisos, obviamente, sus usuarios subordinados también debían perder estos permisos en cascada.

Para modelizar este problema de forma sencilla, se suponía que los permisos se perdían o ganaban de manera escalonada, es decir, de acceso nulo se podía mejorar a sólo lectura y de éste descender de nuevo o ascender a acceso total.

Con esta convención, el modelo de autómata Componente k-ésimo resultaba ser el de la Figura 3.21:

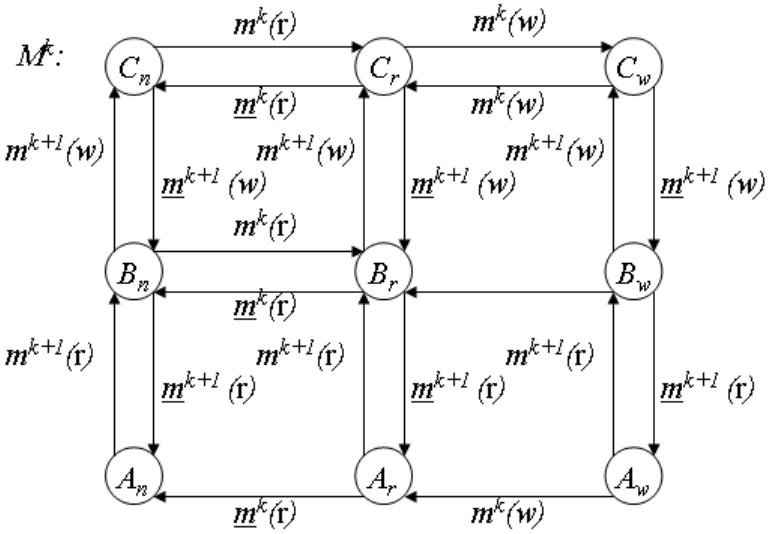


Figura 3.21: Autómata Componente M^k, meta-acceso en la capa k

En este autómata se podía identificar los estados A B y C (acceso nulo, sólo lectura y acceso total, respectivamente) con los subíndices n, r, w, que indican que el autómata ha concedido permisos a su autómata esclavo (situado en la capa k-1) de acceso nulo, sólo lectura y acceso total respectivamente.

Entre estos nueve estados se pueden encontrar ocho diferentes tipos de acciones que podemos englobar en esta clasificación:

- Acciones Horizontales: Se trata de acciones output, que indican un cambio de permisos del subordinado. al ser output provocan un cambio en cascada en el componente esclavo situado en la capa k-1. Considerando los estados como A, B, C, lo que cambia realmente son los subíndices de esos estados. Podemos distinguir cuatro tipos:
  - $m^k(r)$ es una acción de concesión, cambia al esclavo de un permiso de sólo lectura desde el acceso nulo. Nótese que la acción puede significar tanto un cambio desde el estado C_n al C_r como desde el B_n al B_r
  - $\underline{m}^k(r)$ es una acción de revocación, cambia al esclavo de un permiso de acceso nulo desde el de sólo lectura. Nótese que la acción puede significar tanto un cambio desde el estado C_r al C_n como desde el B_r al B_n y como desde el A_r al A_n
  - $m^k(w)$ es una acción de concesión, cambia al esclavo de un permiso de acceso total desde el de sólo lectura. Nótese que la acción puede significar únicamente un cambio desde el estado C_r al C_w
  - $\underline{m}^k(w)$ es una acción de revocación, cambia al esclavo de un permiso de sólo lectura desde el acceso total. Nótese que la acción puede significar tanto un cambio desde el estado C_w al C_r como desde el B_w al B_r y como desde el A_w al A_r
- Acciones Verticales: Se trata de acciones input, que indican un cambio de permisos forzado por el superusuario que se halla en la capa k+1 y que tiene la facultad de conceder o revocar permisos en el usuario de la capa k. Cambia el estado del autómata componente dejando inalterado el subíndice, es decir, los permisos concedidos al usuario siguiente. Podemos distinguir cuatro tipos:
  - $m^{k+1}(r)$ es una acción de ganancia, cambia al autómata a un permiso de sólo lectura desde el acceso nulo. Nótese que la acción puede significar tanto un cambio desde el estado A_n al B_n como desde el A_r al B_r y como desde el A_w al B_w
  - $\underline{m}^{k+1}(r)$ es una acción de pérdida, cambia al autómata a un permiso de acceso nulo desde el de sólo lectura. Nótese que la acción puede significar tanto un cambio desde el estado B_n al A_n como desde el B_r al A_r y como desde el B_w al A_w
  - $m^{k+1}(w)$ es una acción de ganancia, cambia al autómata a un permiso de acceso total desde el de sólo lectura. Nótese que la acción puede significar tanto un cambio desde el estado B_n al C_n como desde el B_r al C_r y como desde el B_w al C_w
  - $\underline{m}^{k+1}(w)$ es una acción de pérdida, cambia al autómata a un permiso de sólo lectura desde el acceso total. Nótese que la acción puede significar tanto un cambio desde el estado C_n al B_n como desde el C_r al B_r y como desde el C_w al B_w

Reformulando el problema en términos de ACE

Vamos a considerar el problema partiendo de la solución planteada por los autómatas TEAM e iremos eliminando redundancias paso a paso.

En primer lugar y por simplicidad, se van a cambiar los nombres de las acciones output m^k y $\underline{m}^k$ por acciones *conceder* y acciones *revocar* y las acciones input m^{k+1} y $\underline{m}^{k+1}$ *conseguir* y *perder* en el caso de las input.

Hasta el momento no se ha hecho más que traducir cada autómata componente M_k en uno AC. ahora para tener el modelo completamente traducido basta con definir las reglas que permitiesen cooperar dos a dos a los k autómatas implicados.

También se pueden usar los atributos clásicos para definir sólo una clase de autómatas, y relacionar cada autómata individual con otro perteneciente a su misma tarea (su master) y con un tercero perteneciente a otra tarea (su esclavo). Pero dado que partimos de un número finito de autómatas esto no es necesario.

Si se usa el modelo extendido y añadimos atributos numéricos el planteamiento es aún más sencillo. Si se añadens tres atributos numéricos que indiquen respectivamente la capa, los permisos que tiene el autómata y los permisos que ha concedidos sucede que se puede asumir que con esas cuatro acciones tenemos la misma funcionalidad del problema.

El Autómata Cooperativo que representa a un usuario de esta manera equivalente a un autómata Componente k-ésimo del ejemplo TEAM, resulta ser el de la Figura 3.22

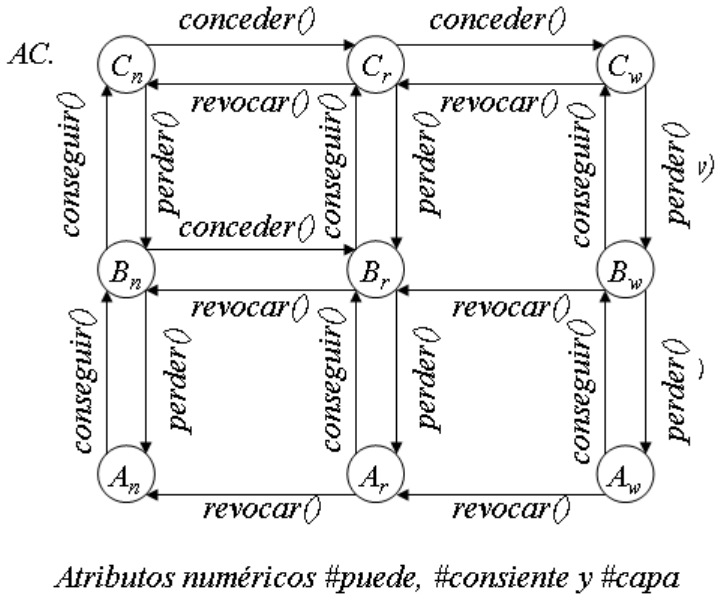


Figura 3.22: Autómata Cooperativo de la capa k, versión redundante

Un autómata cooperativo de estas características contiene mucha información redundante (que se irá eliminando más adelante)

En primer lugar tenemos los mismos estados que antes y como información adicional los tres nuevos atributos.

Los atributos son:

- #capa: indica la capa donde se encuentra el usuario. Se halla delimitado entre 0 y N.
- #puede: indica qué permisos puede conceder. Se halla delimitado entre 1 y 3.
- #consiente: indica qué permisos ha concedido. Se halla delimitado entre 1 y 3.

Queda claro que los dos últimos atributos junto con los diferentes estados representan información redundante.

Las reglas de transición que irían asociadas a estos autómatas serían las siguientes:

```
name: RevocarPermisos
 synch:   A2.revocar()+ A1.perder()
 guard:   A2.#capa == A1.#capa+1
 update:  A1.#puede -= 1
          A2.#concede -= 1


name: ConcederPermisos
 synch:   A2.conceder()+ A1.conseguir()
 guard:   A2.#capa == A1.#capa+1
 update:  A1.#puede += 1
          A2.#concede += 1
```

Las reglas marcadas con * son una simplificación. En realidad el modelo ACE no admite este tipo de operaciones en las guardas, pero es fácilmente imitable incluyendo otro atributo "anterior" que indicase cual es la capa anterior a la actual. Estos dos atributos "capa" y "anterior" serían inicializados en la regla inicial de construcción de los autómatas y no se modificarían en todo la vida del sistema. Se ha optado por esta sintaxis por razones de claridad en el ejemplo.

Las acciones "output" del autómata situado en la capa 0 serían algo distintas, pues al no haber *esclavo* no implicarían acción asociada a ningún otro autómata. Como no tendrían sentido se ha optado por no incluirlas.

Otra cosa sería las acciones del autómata situado en la capa más alta, el superusuario. Este autómata no tendría otro *master* que le dictase las acciones y por tanto sus acciones de *conseguir()* y de *perder()* serían acciones libres y por tanto merecerían otro tratamiento y sus propias reglas de transición.

Nótese que, en problema original de los autómatas Team, modelizar la relación con más de un autómata esclavo sería muy complicado dado que habría que añadir los autómatas componentes esclavos y realizar sincrónicamente sus acciones. En el modelo de los Autómatas Cooperativos realizarlo sincrónicamente consiste tan sólo en añadir estos autómatas a la regla y también es posible realizarlo de manera asíncrona simplemente desglosando cada regla *ConcederPermisos* y *RevocarPermisos* en dos, *ConcederPermisos*, *ConseguirPermisos* y *PerderPermisos*, *RevocarPermisos* respectivamente a de tal manera que sólo se pudiera acceder a la regla *ConseguirPermisos* si el atributo *#concede* del autómata de la capa superior así lo permite y en caso contrario sólo se pudiese realizar la regla *PerderPermisos* de la siguiente manera:

```
name: RevocarPermisos
 synch:   A2.revocar()
 guard:   A2.#concede>1
 update:  A2.#concede -= 1
name: PerderPermisos
 synch:   A1.perder()
 guard:   A2.#capa == A1.#capa+1
          A2.#concede <A1.#puede
 update:  A1.#puede -= 1

name: ConcederPermisos
 synch:   A2.conceder()
 guard:   A2.#concede<3
 update:  A2.#concede += 1
name: ConcederPermisos
 synch:   A1.conseguir()
 guard:   A2.#capa == A1.#capa+1
          A2.#concede >A1.#puede
 update:  A1.#puede += 1
```

Sin embargo, como ya se ha dicho, este modelo tiene mucha información redundante que podría ser eliminada sin alterar en absoluto ni las reglas ni los atributos.

Un ejemplo de ello sería el autómata que se muestra en la Figura 3.23. En el autómata AC2, se han eliminado los subestados señalados por los subíndices n, r y w, dado que dicha información estaría ya implícita en el atributo $\#concede$ que contiene valores entre 1 y 3.

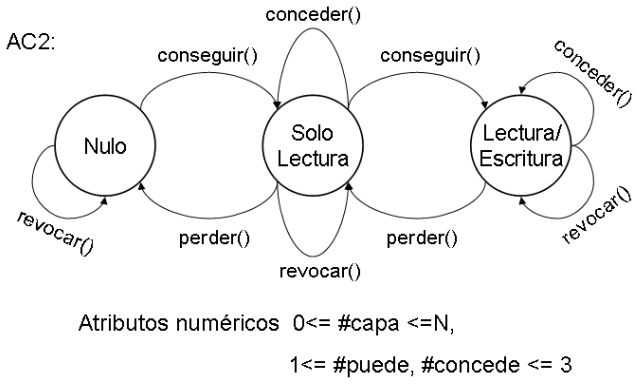


Figura 3.23: Autómata Cooperativo de la capa k, versión menos redundante

Bajo este punto de vista puede verse que un usuario de cualquier capa (excepto la capa 0, claro está) puede conseguir o perder sus permisos en cualquier momento, y que dependiendo de qué permisos tiene, puede conceder o revocar los permisos de su subordinado.

Incluso, sin modificar las reglas en absoluto se puede reducir hasta el mínimo el autómata a la manera de la Figura 3.23

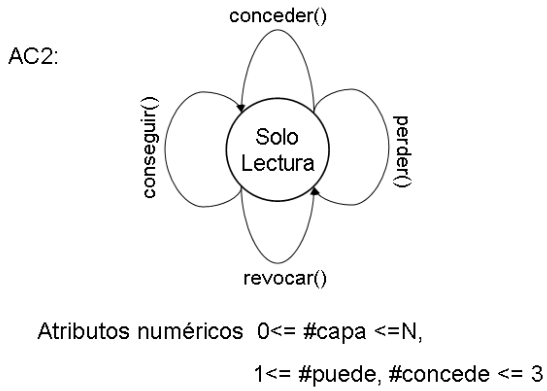


Figura 3.24: Autómata Cooperativo de la capa k, versión sin redundancia

Puede verse que en el modelo de los AC, se sigue teniendo la posibilidad de una componente visual clara que sin embargo se puede perder en la versión más reducida. Llegar a un compromiso entre la concreción del modelo y su comprensibilidad es tarea del diseñador.

3.2.6. Transformación de un modelo ACE a un modelo de Redes de Referencias

En esta sección se plasma la similitud entre el modelo extendido resultante y la propuesta de redes de referencias, para ilustrar la cercanía del modelo de los Autómatas Cooperativos Extendidos a las redes de Petri pese a su aspecto visual más cercano a la teoría de autómatas El mecanismo de transformación es muy simple:

1. Transformar cada tipo de autómata en una Red de Referencias, donde:
   - Cada estado se convierte en un lugar.
   - Cada acción se convierte en una transición.
   - Cada nombre de acción se convierte en una inscripción como un canal síncrono en la transición.
   - Los atributos del autómata se convierten en una tupla que se usará como token.
   - Las acciones que conlleven cambios en los atributos tendrán un doble juego de valores, para los atributos originales y para los atributos post-disparo.
2. Transformar el conjunto de reglas de transacción en una Red de Referencias, donde:
   - Regla de creación de un autómata se convierte en la creación de una instancia de la red que lo representa y su paso como token al lugar correspondiente.

- Cada Regla con una guarda se convierte en una transición con una inscripción de guarda sobre los valores de la tupla-token.
- Cada Regla con una actualización se convierte en una transición donde el token contendrá la tupla con los valores originales y los valores actualizados.
- Cada Regla que afecte a un token tendrá como inscripción un canal síncrono con la instancia que corresponda a cada autómata y con el nombre de la transición a disparar en dicha instancia.
- Cada Regla de destrucción se representará como una transición a un lugar sin salida.

3. Se dispondrá de un lugar repositorio para cada tipo de autómata que servirá para mantener las instancias de red construidas en los pasos anteriores.

Un ejemplo sencillo de transformación

En esta sección se recuerda el de la construcción de un objeto complejo o sencillo con el sistema de robots visto anteriormente recordando brevemente la modelización según el modelo ACE y, para acabar, la transformación de dicho modelo en el simulador Renew.

El problema Recuerdese el problema de modelizar modelizar un sistema de montaje en una fábrica. Se tienen una serie de objetos a construir que constan de una serie de piezas colocadas en un cierto orden. Para ello, se tiene dispuesta una cinta transportadora que puede trasladar cada objeto adelante y atrás entre los robots que deben poner dichas piezas. Se dispone, asimismo, de tantos robots situados en una cierta ubicación como de piezas (por simplificar no se implementará el mecanismo de reposición de piezas). Una representación artística del sistema aparece de nuevo en la Figura 3.25.

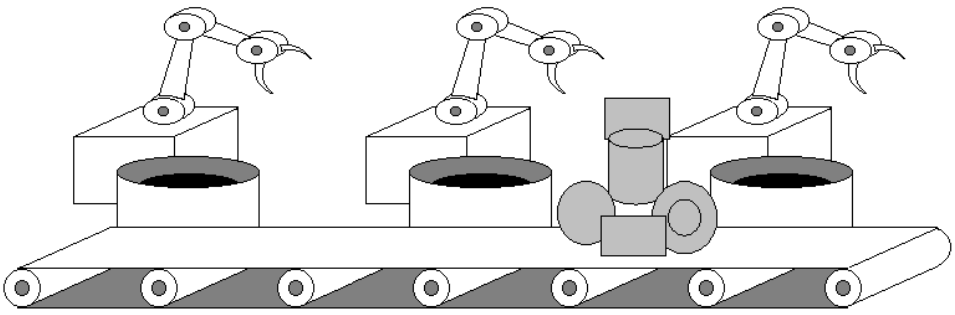


Figura 3.25: Cinta, robots, y objetos

El planteamiento del modelo es sencillo sobre todo si los objetos a construir admiten sólo un orden de montaje de sus piezas. Sin embargo, si el objeto a construir aún conteniendo las mismas piezas admite diferentes órdenes de montaje, es decir,

con alternativas respecto a qué pieza poner en cada momento como el de la Figura 3.26 (donde las piezas deben ponerse según el orden de superposición del dibujo) se experimenta un considerable aumento en la complejidad del autómata ya que se deben representar también todos los posibles "caminos" que conformen los estados intermedios.

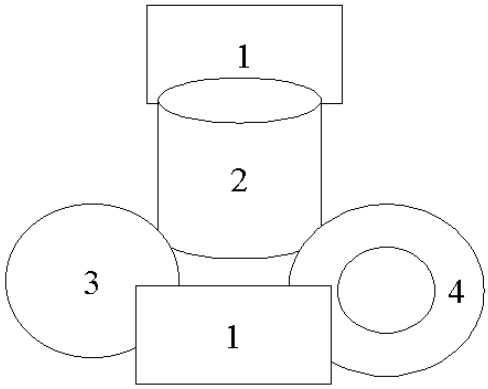


Figura 3.26: Un objeto con varias posibilidades de montaje

El autómata que representaba a un objeto en construcción quedaba como puede verse en la Figura 3.27 puesto que las que delimitan las restricciones son las propias guardas de las reglas de transacción y no los estados o las acciones que contiene el propio autómata. En este caso, el autómata consta tanto de los propios estados y acciones como de los atributos numéricos con la posición y del número de las distintas piezas que contiene el objeto en proceso de fabricación.

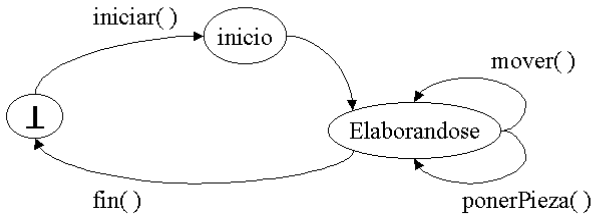
Recordemos que las redes de referencias (Reference Nets) [92, 93, 94] son redes de alto nivel en las que las transiciones contienen anotaciones (canales síncronos) que indican que el disparo de dichas transiciones se produce síncronamente con el de otras redes referenciadas.

Los tokens contienen información y se utilizan para comprobar las guardas de disparo de las transiciones.

En la introducción de los ACE se decía que podían verse como redes de alto nivel, en las que los tokens son entidades activas con una porción de memoria privada. En este apartado se va a utilizar el modelo de autómatas del apartado anterior representando a los mismos como redes de Petri y referenciándolos desde una red principal que representará las reglas de transacción.

En la Figura 3.28 se muestra al objeto complejo que se vio en la Figura 3.26 representando al autómata de la Figura 3.27. Se puede apreciar que se mantiene la misma estructura del objeto cambiando estados por lugares y acciones por transiciones.

Nótese que sólo se considerará el objeto completado cuando se cumpla la guarda y se hayan puesto todas las piezas de cada tipo que son precisas. La información que



Atributos Numéricos $1 \leq \#pos \leq 4$, $0 \leq \#piezas1 \leq 2$
$0 \leq \#piezas2 \leq 1$ $0 \leq \#piezas3 \leq 1$, $0 \leq \#piezas4 \leq 1$

Figura 3.27: Autómata que representa un objeto complejo con atributos numéricos

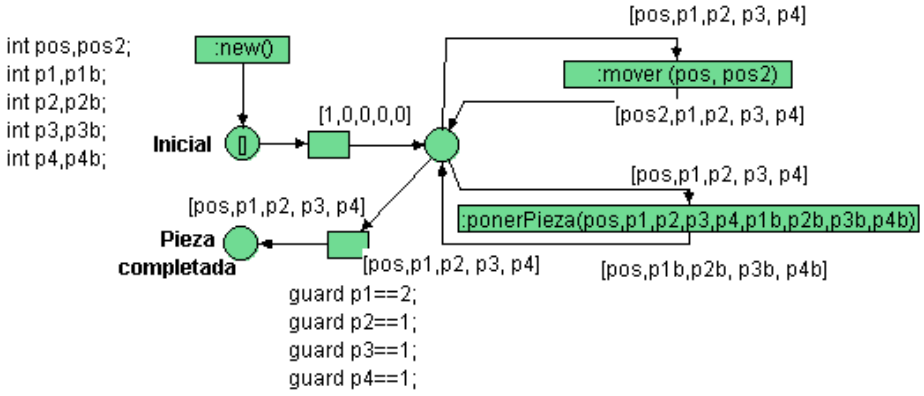


Figura 3.28: Red que representa un objeto complejo

antes estaba contenida en la memoria privada de los autómatas se pasa ahora en los tokens de la red. Esta información, consiste en: la posición en la que se encuentra el objeto y el número de piezas de tipo 1, 2, 3 y 4, respectivamente.

Por otro lado, el autómata ACE correspondiente al robot, que puede verse en la Figura 3.29, es muy simple ya que sólo incluye una transición entre los estados de quieto a preparado, sincronizándose con el objeto para colocar su pieza.

En la Figura 3.30 se muestra al robot del autómata de la Figura 3.29 con el mismo formato que la red anterior, es decir cambiando estados por lugares y acciones por transiciones.

Como puede verse en el caso del robot la única información local es la posición que ocupa (que coincide con el tipo de pieza que va a poner) y no está restringido como en el caso anterior ya que no tiene que finalizar. Además, para la regla de transacción *Preparar* por simplificar el modelo no se ha definido anotación en la

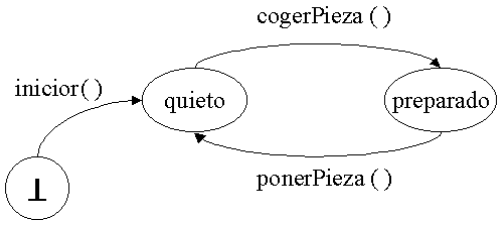


Figura 3.29: Autómata Cooperativo Extendido que representa al robot

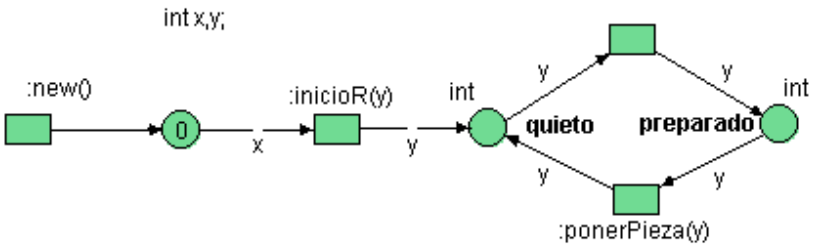


Figura 3.30: Red que representa al Robot del ejemplo

transición correspondiente y se ha dejado interna a la red.

Los robots se construyen según las siguientes reglas de transacción:

```
name: NewRobot_posicion1
 synch:   Robot.inicior()
 guard:   null
 update:  Robot.#pos = 1

name: NewRobot_posicion2
 synch:   Robot.inicior()
 guard:   null
 update:  Robot.#pos = 2

name: NewRobot_posicion3
 synch:   Robot.inicior()
 guard:   null
 update:  Robot.#pos = 3

name: NewRobot_posicion4
 synch:   Robot.inicior()
 guard:   null
 update:  Robot.#pos = 4
```

Los movimientos de la cinta transportadora (que cambian la posición del objeto) se reducen a 2 reglas de transacción sobre el propio objeto:

```
name: MovimentoAdelante
 synch:   OC.mover_oc()
 guard:   OC.#pos <4
 update:  OC.#pos += 1

name: MovimientoAtras
 synch:   OC.mover_oc()
 guard:   OC.#pos >1
 update:  OC.#pos -= 1
```

En el objeto (OC) las reglas para introducir una pieza tienen que ver con el orden de introducción. Las reglas de transacción que definen la inserción de cada una de las piezas incluyen además en sus guardas y actualizaciones la información necesaria para asegurar que el objeto se construya correctamente.

En la regla de transacción para acoplar la pieza 1 se tiene como guarda que el número de piezas de tipo uno sea igual a 0, (y, evidentemente, que el número de piezas de tipos 3 y 4 también, pero esto es innecesario decirlo) o bien que el número de piezas de tipo 1 sea igual a 1 y el número de piezas de tipos 3 y 4 también quedando, por tanto, de la siguiente manera:

```
name: Acoplar_pieza_1
 synch:   OC.ponerPieza_oc()   +   Robot.ponerPieza()
 guard:   OC.#pos == Robot.#pos
          OC.#pos == 1
          OC.#pieza1 == OC.#pieza3
          OC.#pieza3 == OC.#pieza4
 update:  OC.#pieza1 += 1
```

Para insertar la segunda pieza, además de estar en la posición adecuada y no haber introducido ninguna pieza de ese tipo (#piezas2==0) es obligatorio que se haya introducido una y sólo una pieza del tipo 1, si no, nos encontraríamos todavía en la fase anterior:

```
name: Acoplar_pieza_2
 synch:   OC.ponerPieza_oc()  +   Robot.ponerPieza()
 guard:   OC.#pos == Robot.#pos
          OC.#pos == 2
          OC.#pieza2 == 0
          OC.#pieza1 == 1
 update:  OC.#pieza2 += 1
```

Para introducir la tercera o cuarta pieza se debe asegurar no haberla introducido todavía y en que ya se haya introducido la anterior (que en este caso es la segunda). La existencia de la primera pieza está garantizada por la aparición de la segunda y así sucesivamente si hubiese anteriores:

```
name: Acoplar_pieza_3
 synch:   OC.ponerPieza_oc()  +   Robot.ponerPieza()
 guard:   OC.#pos == Robot.#pos
          OC.#pos == 3
          OC.#pieza3 == 0
          OC.#pieza2 == 1
 update:  OC.#pieza3 += 1
```

```
name: Acoplar_pieza_4
 synch:   OC.ponerPieza_oc()  +  Robot.ponerPieza()
 guard:   OC.#pos == Robot.#pos
          OC.#pos == 4
          OC.#pieza4 == 0
          OC.#pieza2 == 1
 update:  OC.#pieza4 += 1
```

Cuando un robot pone una pieza su siguiente paso es volver al estado en que se encontraba antes (por ejemplo, coger una pieza nueva, volver a la posición de reposo, etc.); para ello se define una regla que sólo afecta al robot y que simplemente lo deja preparado para su siguiente interacción con otro objeto en la cadena de montaje:

```
name: Preparar
 synch:   + Robot.cogerPieza()
 guard:
 update:
```

En cuanto a las reglas de transacción en la traducción por una red de referencias se ha definido una nueva red que contiene tantas transiciones como reglas teníamos en la versión ACE. Dicha red puede verse en la Figura 3.31.

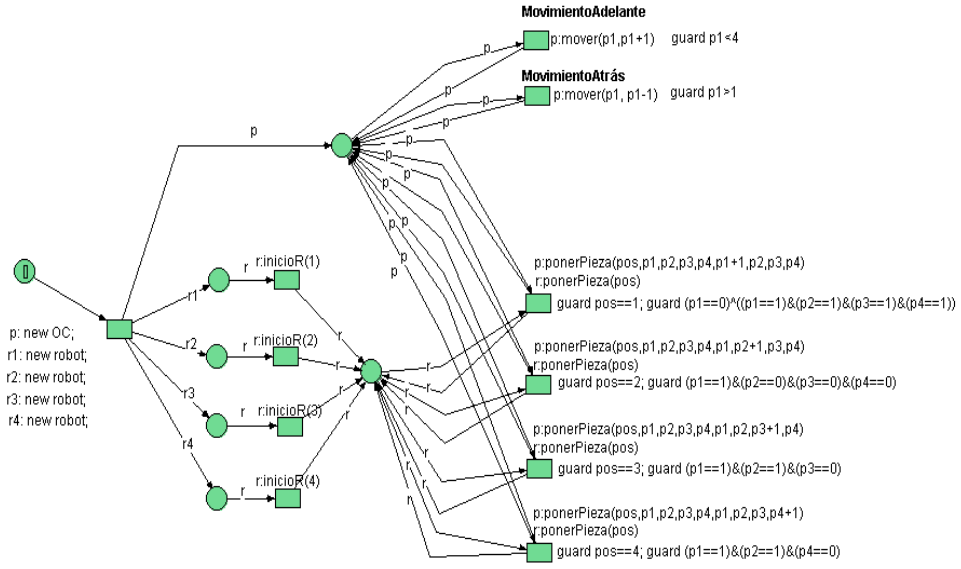


Figura 3.31: Red que representa las reglas de transacción de la Fábrica

En la red principal, que representa a la fábrica de la Figura 3.25 se pueden distinguir las inscripciones en las transiciones que indican la creación del objeto *p:new OC* y la creación de los robots *r1.new robot, r2.new robot, r3.new robot y r4.new robot*, la inicialización de los mismos asignándoles una posición *r.inicioR(1), r.inicioR(2),*

r.inicioR(3) y r.inicioR(4). En cuanto a las transiciones que representan al resto de reglas de transacción (las situadas más a la derecha de la figura) se puede ver que incluyen tanto los canales síncronos que relacionan esta y las otras redes como las guardas, por lo que se tratan con detalle a continuación:

- La transición etiquetada como *MovimientoAdelante* como puede verse utiliza el canal *p:mover(p1, p1+1)* que permitirá el cambio del atributo *p1* del token utilizado en la red OC (objeto complejo) por *p1+1*, es decir, en términos de ACE, corresponde a la regla de transacción homónima en la que coincide la guarda y la actualización es *#pos+=1;*

- La transición etiquetada como *MovimientoAtrás* como puede verse utiliza el canal *p:mover(p1, p1-1)* que permitirá el cambio del atributo *p1* del token utilizado en la red OC (objeto complejo) por *p1-1*, es decir, en términos de ACE, corresponde a la regla de transacción homónima en la que coincide la guarda y la actualización es *#pos-=1;*

- La siguiente transición utiliza el canal *:ponerPieza* sobre dos redes, p y r, es decir, que sincronizará dos disparos en dos redes. Concretamente a una red r de tipo Robot y a una red p de tipo OC cuyos tokens provienen de los lugares que mantienen estos dos tipos de objetos. En términos de ACE, corresponden con la regla de transacción AcoplarPieza1. La guarda es idéntica a la de ésta (debe coincidir la posición 1 y el número de piezas de tipo 1, 2, 3 y 4 ha de ser =0 o bien , =1 en todos los casos) la actualización consiste en que el número de piezas de tipo 1 se incrementa en uno).

- La siguiente transición utiliza el mismo canal *:ponerPieza* sobre las dos redes, pero cambia el contenido del token y la guarda. En términos de ACE, corresponden con la regla de transacción AcoplarPieza2. La guarda es idéntica a la de ésta (debe coincidir la posición 2 y el número de piezas de tipo 1 y 2 ha de ser 1 y 0 respectivamente) la actualización consiste en que el número de piezas de tipo 2 se incrementa en uno.

- La siguiente transición utiliza el mismo canal *:ponerPieza* sobre las dos redes, pero cambia el contenido del token y la guarda. En términos de ACE, corresponden con la regla de transacción AcoplarPieza3. En este caso, la guarda es equivalente a la de la regla de transacción, y como alternativa hace referencia también al número de piezas de 1 aunque en realidad no sería necesario. La actualización consiste en que el número de piezas de tipo 3 se incrementa en uno.

- La siguiente transición utiliza el mismo canal *:ponerPieza* sobre las dos redes, pero cambia el contenido del token y la guarda. En términos de ACE, corresponden con la regla de transacción AcoplarPieza4. En este caso, la guarda es equivalente a la de la regla de transacción, y como alternativa hace referencia también al número de piezas de 1 aunque en realidad no sería necesario. La actualización consiste en que el número de piezas de tipo 4 se incrementa en uno.

Para simplificar no se ha considerado relevante el mecanismo de reposición de piezas, ni el contenedor de las mismas de las cuales las coge el robot, ni el resto de mecanismos que pudiesen intervenir en una cadena de montaje real.

Capítulo 4

Análisis a priori de un sistema

4.1. Una justificación al análisis previo

La complejidad de los nuevos sistemas y consecuentemente sus respectivos modelos se ha ido incrementando constantemente y exponencialmente. En este sentido, simplificar un sistema complejo en subsistemas más simples, pero también, detectar problemas como cuellos de botella puede resultar crucial en las primeras etapas de diseño. Realizar este tipo de detección durante la fase de implementación o, peor aún, en la de implantación de un sistema puede incrementar dramáticamente el presupuesto de la construcción del mismo pues puede suponer la duplicación del tiempo de ejecución al tener que reiniciar el diseño replanteando el problema encontrado.

Hoy en día, además de los modelos expuestos en capítulos anteriores, existen muchos modelos formales para diseñar de manera sencilla y completa sistemas de estructura compleja, multiagente y distribuidos. Si hablamos de bajo nivel, se pueden representar procesos como grafos de estados etiquetados con las propiedades del sistema donde las aristas sean las transiciones entre estados, como en las estructuras de Kripke [91]. En un nivel mayor de detalle un ejemplo claro son las tecnologías UML [125] que permiten, usando diagramas de estructuras y de comportamiento, definir sistemas que más tarde sean implentados. Estos y otros modelos pretenden capturar la dinámica de diferentes sistemas en un amplio rango de disciplinas, desde los sistemas biológicos hasta los sistemas socio-técnicos adaptativos.

La simulación de estos modelos es un problema que plantea una serie de desafíos, dada su naturaleza no determinista con un comportamiento centrado en los datos. Resulta evidente que cuanto mayor sea la escala de estos modelos, mayores desafíos se encuentran para estudiar y simularlos. Es por ello necesario seccionar estos grandes sistemas en componentes menos complejas y más manejables si se quiere realizar la simulación y estudiarla adecuadamente y resolver los problemas que se planteen en este proceso de manera más sencilla, como se muestra en [3]. Existe en la literatura toda una serie de trabajos para garantizar que las simulaciones se realicen de una manera segura y correcta, como por ejemplo en [134] y también para detectar problemas como los cuellos de botella durante la etapa de simulación [128] y [123]. Sin embargo, si dichos estudios se realizasen en la etapa de diseño del modelo en lugar de en la etapa de simulación, el coste en tiempo de resolución de dichos problemas podría

reducirse considerablemente. Sería además crucial, además, si se pudiese realizar el estudio de simplificación y de detección de problemas a priori de manera automática, ya que el diseñador podría cambiar de rumbo en su modelización de un sistema en una etapa mucho más temprana de lo habitual, con el ahorro en tiempo y, por tanto, en coste que ello conllevaría.

En esta sección se muestra cómo realizar este tipo de análisis sobre un modelo que esté siendo diseñado bajo la perspectiva de los Autómatas Cooperativos Extendidos. Además, dicho análisis está ya integrado en una herramienta desarrollada para la edición de estos modelos, que garantiza la corrección sintáctica de los mismos. Dicho análisis puede realizarse en cualquier momento del diseño y provee información sobre los procesos independientes y los autómatas o reglas que conforman los posibles cuellos de botella de un sistema.

En este capítulo primero se definen las relaciones entre los distintos elementos que conforman un modelo con los Autómatas Cooperativos, es decir, las reglas y los autómatas. El grado de certeza o también llamada la fortaleza de estas relaciones van desde la mera posibilidad (más débil) hasta la certeza absoluta (más fuerte), sin embargo, como se ha comentado anteriormente, la ejecución de estos modelos suele depender de los datos concretos del sistema y sólo puede verse en tiempo de ejecución. En cualquier caso, descartando de entrada los problemas y simplificando los sistemas utilizando las versiones más débiles de las relaciones ya se adelantan las soluciones a la propia fase de diseño no teniendo que esperar a la de simulación o lo que es peor a la de implementación o implantación.

A continuación se analizan dichas relaciones y se aplica un algoritmo que separa el conjunto de reglas en conjuntos más pequeños identificando procesos independientes si los hay y posibles cuellos de botella. Nótese que estos cálculos se pueden realizar desde fases muy iniciales y, por tanto, pueden orientar al diseñador del sistema en la modelización del mismo y, por consiguiente, en la toma de decisiones.

Queda claro que este tipo de análisis y conclusiones pueden ser deducidas por un programador con experiencia que tenga un conocimiento profundo de la semántica del sistema. Sin embargo, la dificultad se plantea en términos de escala, lo que puede resultar asequible en un sistema pequeño, cuando se habla de gran escala la heurística necesaria y el conocimiento profundo del sistema completo es algo mucho más complicado, de ahí la idea de realizar este estudio previo de manera automática para proporcionar una base al diseñador con la que trabajar sin necesidad de esa heurística y conocimientos del sistema completo que sí necesitará más adelante. Para ilustrar el proceso de análisis automático se plantea un problema modelado con los Autómatas Cooperativos Extendidos, sobre el que, si eliminamos los nombres de métodos, estados y autómatas, resulta complicado entender la funcionalidad del mismo. Sobre este modelo genérico se plantea el análisis a priori para ilustrar el mismo.

4.2. Vinculación, Concurrencia y Competencia

En un sistema modelado con la propuesta de los Automatas Cooperativos Extendidos los cambios de estado de cada autómata o conjunto de autómatas son causados por el disparo de una regla de transacción. El nexo de unión entre esos autómatas y las reglas de transacción son las acciones comunes presentes tanto en el conjunto de acciones de la definición del autómata como en la del correspondiente vector de

sincronización de las reglas. En estos términos se puede decir que existen relaciones estables entre los autómatas y las reglas a través de dichas acciones comunes.

Como ya se explicó previamente en el Capítulo 3, un autómata puede ligarse a una regla a través de más de una acción, pero, sin embargo, sólo una de dichas acciones intenvendrá en el disparo de la regla por cada autómata dado que en cada autómata el cambio de estado vendrá indicado por dicha regla. Una vez se establezcan claramente las relaciones entre los autómatas y las reglas de transacción, el siguiente paso lógico será la definición de las relaciones entre autómatas, dado que, si sabemos qué autómatas están conectados con qué reglas a través de qué acciones, es posible identificar la incompatibilidad entre autómatas en un sistema dado (esto es, aquellos que tratarían de participar en el disparo de una regla a través de la misma acción del vector de sincronización de la misma y que, por esta razón, no podrán ser ejecutados concurrentemente y, por contra, de manera equivalente es posible identificar a aquellos autómatas que puedan (y en algunos casos posiblemente deban) verse implicados simultáneamente en el disparo de una misma regla de transacción. Ambos tipos de relaciones permiten definir un tercer tipo de relaciones, las relaciones entre reglas. Cuando dos reglas compitan por un mismo autómata claramente su ejecución en paralelo será imposible y, por contra, si jamás competirán por autómatas comunes las reglas podrán ser consideradas independientes.

La relación entre un autómata y una regla a través de una acción se llama relación de vinculación o *linking relation* y se representa por el símbolo $\triangleright\!\triangleright$. Dependiendo del grado de certidumbre de dicha relación se definirán cuatros niveles de fortaleza siendo el más fuerte cuando inevitablemente exista dicha relación, es decir, que el disparo de una regla implique obligatoriamente la participación del autómata.

El disparo de las reglas en el modelo de los Autómatas Cooperativos Extendidos se realiza de una en una ya que en cada instante podría haber un conflicto entre dos reglas por el uso de un autómata y, por tanto, bajo esta perspectiva dos autómatas sólo serán concurrentes si y solo sí ambos participan en el disparo de la misma regla de transacción en el mismo instante. Esta relación entre autómatas en una regla se denomina relación de concurrencia o *concurrency relation* y se representa por el símbolo $\bowtie$. Dependiendo del grado de certidumbre de dicha relación se definirán cuatros niveles de fortaleza siendo el más fuerte cuando inevitablemente exista dicha relación, es decir, que el disparo de la regla de transacción que los relaciona implique obligatoriamente la participación de ambos autómatas.

Conociendo qué automatas están vinculados con qué reglas a través de qué acciones y qué autómatas son concurrentes entre sí en qué reglas el siguiente paso es la definición de las relaciones entre las reglas. Se denomina relación de competencia entre reglas o *competition relation* y se representa por el símbolo $\triangleleft\!\triangleright$ a la relación existente entre dos reglas cuando compiten por la participación en su disparo de un autómata. Otra vez dependiendo del grado de certidumbre de dicha relación se definirán cuatros niveles de fortaleza siendo el más fuerte cuando inevitablemente exista dicha relación, es decir, ambas reglas compitan por el mismo autómata y que, por tanto, el disparo de una regla de transacción implique que posiblemente la otra regla no pueda dispararse pues no se garantiza que el autómata por el que compiten no haya cambiado de estado tras el disparo de la primera.

En resumen, los tres tipos de relaciones que se definen a continuación son:

- Vinculación entre un autómata y una regla de transacción: $A \overset{*}{\triangleright\!\triangleright}_e r$ (linking

relation)

- Concurrencia entre dos autómatas: $A_0 \overset{*}{\bowtie}_r A_1$ (concurrency relation)
- Competencia entre dos reglas de transacción: $r_0 \overset{*}{\triangleleft\triangleright} r_1$ (competition relation)

En cada relación la fortaleza de la misma se representa mediante un superíndice $*$ que puede tomar los valores: vw, w, s, y vs, que corresponden respectivamente a muy débil (very weak), débil(weak), fuerte (strong) y muy fuerte(very strong). En las relaciones de vinculación e es la acción que enlaza al autómata A con la regla de transacción r. En las relaciones de concurrencia r es la regla en la que ambos autómatas A_0 y A_1 son concurrentes. Por último en las relaciones de competencia r_0 y r_1 son las reglas que están ligadas a uno o más autómatas de manera simultánea

La siguiente notación se ha usado para simplificar las definiciones de esta sección:

- Sea $\Sigma_{r_{Synch}}$ el conjunto de acciones del vector de sincronización $Synch$ de la regla de transacción r,
- sea $comp_r(e)$ el componente de r_{Synch} que contiene la acción e,
- sea A el conjunto de autómatas del sistema modelizado con los Autómatas Cooperativos Extendidos,
- sea k un autómata del sistema ($k \in A$),
- sea Σ_k el conjunto de acciones del autómata k,
- sea S_k el conjunto de estados del autómata k,
- sea s_k el estado actual del autómata k donde $s_k \in S_k$,
- sea s'_k cualquier estado del autómata k donde $s'_k \in S_k$,
- sea T_k sea el conjunto de transiciones de k,
- sea $Att1_k$ sea el conjunto de atributos esttándar o de tarea de k
- sea $Att2_k$ el conjunto de atributos numéricos de k,
- sea $Guard_r(k)$ sea una función lógica sobre $k \in A$ y $r \in R$ cuyo resultado es *cierto* cuando $\forall\ a \in Att1_k \cup Att2_k$ es decir, que cada atributo a cumple con la restricción de *guard* en la regla de transaccitón r y *false* en caso contrario,
- sea H_r^k el conjunto de acciones habilitadas del autómanta k de acuerdo con la regla de transición r, en otras palabras, aquellas acciones de k que estén en $\Sigma_{r_{Synch}}$ y cuyo estado inicial sea el estado actual del autómata: $H_r^k = \{\ \ e \mid e \in (\Sigma_k \cap \Sigma_{r_{Synch}}) \wedge (s_k e s'_k) \in T_k\}$,
- en lo sucesivo cuando se habla de dos acciones iguales e y e' en un vector de sincronización $Synch$ de una regla r se entiende no sólo que se trate de acciones con el mismo nombre sino también que estén situadas en la misma posición de $Synch$, en caso contrario no se considerarán acciones iguales a efectos de las relaciones en las que se utilicen.

4.2.1. Relaciones de Vinculación entre autómatas y reglas

La relación de Vinculación entre un autómata y una regla de transacción viene de la existencia o no de la misma acción en el conjunto de acciones del autómata y en el vector de sincronización de la regla. A continuacion se presenta la definición de la relación de vinculación entre autómatas y reglas.

Definición 38 (Vinculación a través de una acción ($i \triangleright\triangleright_e r$) [69, 70]) *Para todo autómata i y regla de transacción r de un sistema de Autómatas Cooperativos Extendidos ($\forall\, i \in A,\ \wedge\, r \in R$) i está* vinculado *a r a través de una acción e:*

- muy débilmente *($i \overset{vw}{\triangleright\triangleright}_e r$) si y sólo si la acción e está incluida en el conjunto de acciones del autómata i y también en el vector de sincronización de r ($e \in \Sigma_i \ \wedge\ e \in \Sigma_{r_{Synch}}$);*
- débilmente *($i \overset{w}{\triangleright\triangleright}_e r$) si y sólo si i está muy débilmente vinculado a la regla r a través de la acción e y, además, la acción e esta incluida en el conjunto de acciones habilitadas del automata i ($i \overset{vw}{\triangleright\triangleright}_e r \ \wedge\ s_i es'_i) \in T_i$;*
- fuertemente *($i \overset{s}{\triangleright}\triangleright_e r$) si y sólo si i está débilmente vinculado a la regla r a través de la acción e y, además, todos los atributos de i cumplen con las restricciones de r ($i \overset{w}{\triangleright\triangleright}_e r \ \wedge\ \forall\, a_i \in Att1_i \cup Att2_i\ ,\ Guard_r(a_i)$);*
- muy fuertemente *($i \overset{vs}{\triangleright\triangleright}_e r$) si y sólo si i está débilmente vinculado a la regla r a través de la acción e y, además, no hay ningún otro autómata que esté fuertemente vinculado a r a través de la misma acción*[1] *e ($i \overset{s}{\triangleright\triangleright}_e r \ \wedge\ \nexists\, k \in A\ ,\ (k \neq i \ \wedge\ k \overset{s}{\triangleright\triangleright}_e r)$.*

Informalmente, un autómata:

- está *muy débilmente vinculado* a una regla de transacción si, en algún momento durante la evolución del sistema, el autómata puede realizar una acción incluida en la regla,
- está *débilmente vinculado* a una regla de transacción si, en el estado actual del sistema (y, por tanto, del autómata), este puede realizar una acción incluida en la regla sin considerar si los valores actuales de sus atributos cumplen las guardas de la regla,
- está *fuertemente vinculado* a una regla de transacción si, en el estado actual del sistema (y, por tanto, del autómata) teniendo en consideración si los valores actuales de los atribtos del mismo cumplen las restricciones de la guarda de la regla, el autómata, puede realizar una acción incluida en dicha regla,
- está *muy fuertemente vinculado* a una regla de transacción si, en el estado actual del sistema (y, por tanto, del autómata) dicho autómata es el único de todo el sistema que cumple todas las condiciones para realizar una acción concreta de la regla y, por tanto, en el caso del disparo de dicha regla el autómata se verá necesariamente involucrado.

[1] considerando la misma acción según lo indicado en la notación descrita previamente.

Como puede apreciarse, el nivel de fortaleza y de certeza de las relaciones definidas está descrito de manera incremental. Esto significa que si existe una relación muy fuerte entre un autómata y una regla también existirá la relación fuerte y así sucesivamente, aunque no necesariamente en sentido inverso.

4.2.2. Relaciones de Concurrencia entre autómatas

Las relaciones de concurrencia entre dos autómatas están definidas sobre una regla de transacción concreta. En este caso, dos autómatas se considerarán concurrentes siempre que exista una regla que contenga en su vector de sincronización acciones de ambos autómatas, recordando una vez más que las acciones deben ser distintas, esto es, estar situadas en diferentes posiciones del vector de sincronización de la regla. El nivel de certidumbre de estas reglas viene dado a su vez por las restricciones que se vayan cumpliendo por los atributos del autómata (obviamente, si se está en una fase de diseño previa a la ejecución o simulación del sistema, el único nivel de certeza que se puede considerar es el muy débil). Se puede definir asimismo la regla opuesta, esto es, la independencia de dos autómatas en una regla como se ha comentado previamente, pero a efectos del análisis resulta redundante.

A continuación, se definen las relaciones de concurrencia entre dos autómatas en una regla.

Definición 39 (Concurrencia de dos autómatas i y j en una regla r ($i \bowtie_r j$) [69, 70]) *Para cada par de autómatas diferentes entre sí de un sistema de Autómatas Cooperativos Extendidos* ($\forall\ i,j \in A\ ,\ i \neq j$) *ambos son* concurrentes *entre sí en una regla de transacción* $r \in R$ *muy débilmente* ($i \overset{vw}{\bowtie}_r j$), *débilmente* ($i \overset{w}{\bowtie}_r j$), *fuertemente* ($i \overset{s}{\bowtie}_r j$), *o muy fuertemente* ($i \overset{vs}{\bowtie}_r j$) *si y solo si, respectivamente, i está muy débilmente, débilmente, fuertemente o muy fuertemente vinculado a r a través de una acción* e_i *y, a su vez, respectivamente, j está muy débilmente, débilmente, fuertemente o muy fuertemente vinculado a r a través de una acción* e_j *y ambas acciones son diferentes bien por su nombre o bien son diferentes componentes del vector de sincronización synch de la regla r* ($i \overset{c}{\triangleright\!\triangleright}_{e_i} r \ \wedge\ j \overset{c}{\triangleright\!\triangleright}_{e_j} r \wedge\ (e_i \neq e_j \vee comp_r(e_i) \neq comp_r(e_j)) \wedge c \in \{vw, w, s, vs\}$).

Informalmente:

- Dos autómatas son *muy débilmente concurrentes* entre sí en una regla si ambos contienen acciones presentes en diferentes componentes del vector de sincronización de la regla. En este caso no se tienen en cuenta ni los estados de los autómatas, ni los valores de sus atributos ni las guardas de las reglas. Debe entenderse, por tanto, esta relación como una posibilidad de que durante algún momento de la ejecución del sistema ambos autómatas concurran en una regla.
- Dos autómatas son *débilmente concurrentes* entre sí en una regla si en el estado actual de ambos (y, por tanto, durante la ejecución del sistema) ambos ambos pueden realizar acciones presentes en diferentes componentes del vector de sincronización de la regla. Tampoco se tienen en cuenta restricciones de atributos ni guardas y por tanto, debe entenderse esta relación como la posibilidad en un instante concreto de producirse en las circunstancias adecuadas por las guardas de una concurrencia de ambos autómatas en esa regla.

- Dos autómatas son *fuertemente concurrentes* entre sí en una regla si en el estado actual de ambos (y, por tanto, durante la ejecución del sistema) ambos ambos pueden realizar acciones presentes en diferentes componentes del vector de sincronización de la regla teniendo en cuenta, esta vez si, que los valores de los atributos de ambos cumplan las restricciones impuestas por la guarda de la regla. Esta relación debe entenderse, por tanto, como que es seguro que ambos autómatas pueden ser concurrentes en este instante concreto de ejecución en la regla. Sin embargo, no hay seguridad de que vayan a hacerlo porque pueden encontrarse más autómatas compitiendo en las mismas condiciones.

- Dos autómatas son *fuertemente concurrentes* entre sí en una regla si en el estado actual de ambos (y, por tanto, durante la ejecución del sistema) es obligatorio que en caso de que se dispare la regla ambos participen dado que son los únicos que pueden realizar acciones presentes en diferentes componentes del vector de sincronización de la regla teniendo en cuenta que los valores de los atributos de ambos cumplan las restricciones impuestas por la guarda de la regla. Esto implica que si la regla se dispara se producirá inevitablemente la concurrencia de ambos.

Las relaciones de concurrencia entre autómatas se utilizan para encontrar aquellos agentes que, en un sistema dado, puedan evolucionar en paralelo o de manera probable, posible o segura concurran cuando se produzca el disparo de una regla. Conocer este tipo de relaciones es muy útil para identificar aquellos agentes críticos cuya ejecución va siempre ligada a la ejecución de los demás y, por tanto, se pueden convertir en cuellos de botella del sistema.

Además es posible simplificar un sistema si se identifican autómatas que son concurrentes en todos los procesos en los que participen, esto es, en todas las reglas. Ello significaría que, probablemente, los autómatas que se encuentren en estas condiciones no deban ser considerados por separado, sino como parte de una misma entidad o agente y, por tanto, el número de autómatas, atributos, restricciones y demás pueda verse reducido y, así, el sistema simplificado.

Sin embargo, la relación de concurrencia entre autómatas no cumple la propiedad asociativa, al menos no en los tres grados de menor certidumbre. Esto puede comprobarse fácilmente con un ejemplo:

> Imagínese un sistema con una regla r con el vector de sincronización $synch = A.e_1, B.e_2$ y tres autómatas i, j y k.
>
> Considérese entonces que se encuentran las siguientes relaciones de concurrencia entre autómatas en estas reglas. $i \overset{s}{\rhd\!\rhd}_{e_1} r \wedge j \overset{s}{\rhd\!\rhd}_{e_2} r \wedge k \overset{s}{\rhd\!\rhd}_{e_1} r$.
>
> En este caso ocurre que $i \overset{s}{\bowtie}_r j \wedge j \overset{s}{\bowtie}_r k$ pero por contra ($i \overset{s}{\not\bowtie}_r k$) porque i y k pueden realizar exactamente la misma acción en la misma componente del vector de sincronización de la regla.

Este comportamiento indica que no es posible deducir la concurrencia entre reglas directamente de la concurrencia de los autómatas en las mismas, pero en cambio, sí es posible estudiar directamente la competencia entre reglas.

4.2.3. Relaciones de Competencia entre reglas de transacción

Una vez que las relaciones de vinculación entre autómatas y reglas a través de acciones y las relaciones de concurrencia entre autómatas en una regla han quedado adecuadamente establecidas, el siguiente paso es definir una jerarquía de mayor nivel de relaciones: Las relaciones de Competencia entre Reglas de Transacción.

Aunque la ejecución de un sistema modelizado con la propuesta de los Autómatas Cooperativos Extendidos es básicamente síncrona y sólo una de las reglas de transacción se dispara en un instante dado, más de una regla puede ser candidata simultáneamente a ser disparada y es preciso realizar una elección (o bien manualmente o bien de manera no determinista) sobre qué regla se escoge. Bajo este punto de vista, si más de una regla puede vincularse a un mismo autómata y dichas reglas pueden ser disparadas, en realidad están compitiendo por el uso de dicho autómata.

Definición 40 (Competencia entre dos reglas de transacción ($r_1 \diamond\!\!\triangleright\!\!\triangleleft\!\!\diamond r_2$) [69, 70]) *Dos reglas de transacción en un sistema modelizado con los Autómatas Cooperativos Extendidos* r_1 *y* r_2 compiten *entre sí muy débilmente* ($r_1 \overset{vw}{\diamond\!\!\triangleright\!\!\triangleleft\!\!\diamond} r_2$) *(o débilmente* ($r_1 \overset{w}{\diamond\!\!\triangleright\!\!\triangleleft\!\!\diamond} r_2$) *o fuertemente* ($r_1 \overset{s}{\diamond\!\!\triangleright\!\!\triangleleft\!\!\diamond} r_2$) *o muy fuertemente* ($r_1 \overset{vs}{\diamond\!\!\triangleright\!\!\triangleleft\!\!\diamond} r_2$)*, respectivamente) si y sólo si existe al menos un autómata* i *en el sistema que está muy débilmente (o débilmente, fuertemente o muy fuertemente) vinculado a ambas reglas de transacción* ($\exists i \in A \,/\; i \overset{c}{\triangleright\!\!\diamond} r_1 \wedge i \overset{c}{\triangleright\!\!\diamond} r_2 \wedge c \in \{vw, w, s, vs\}$).

De manera informal:

- Dos reglas de transacción *compiten muy débilmente* si puede ocurrir que no puedan ser disparadas en paralelo porque ambas están vinculadas muy débilmente al mismo autómata por lo que en algún momento de la ejecución ambas pueden tratar de utilizar el mismo autómata.

- Dos reglas de transacción *compiten débilmente* si en un momento dado de la ejecución de un sistema puede ocurrir que no puedan ser disparadas en paralelo porque ambas están vinculadas débilmente al mismo autómata (sin tener en consideración si los valores de los atributos del autómata cumplen las restricciones de las guardas de las reglas).

- Dos reglas de transacción *compiten fuertemente* si en un momento dado de la ejecución de un sistema puede ocurrir que no puedan ser disparadas en paralelo porque ambas están vinculadas fuertemente al mismo autómata. En este caso hay una gran probabilidad de que sólo una de ellas pueda ser finalmente disparada con ese autómata dado que si afectasen al mismo autómata y una de las dos se disparase probablemente este cambiaría de estado o de valores de atributos y dejaría de estar vinculado a la otra regla y esto obligaría a la segunda regla a escoger un autómata diferente con el que estuviese fuertemente vinculado.

- Dos reglas de transacción *compiten muy fuertemente* si en un momento dado de la ejecución de un sistema ambas sólo pueden dispararse utilizando exactamente el mismo autómata, con lo que las probabilidades de que si se disparase una de las dos la otra no pudiera hacerlo a continuación son muy altas.

En un modelo de *ejecución paralela* dos reglas de transacción jamás podrán ser procesadas en paralelo si existe al menos un autómata que con absoluta certeza deba participar en ambos disparos simultáneamente. En un modelo de *ejecución secuencial* una regla de competencia entre reglas implica que la única manera de asegurar que la secuencia de disparo $r_1 \rightarrow r_2$ es posible es garantizando que no hay competición entre r_1 y r_2 o que la acción común e entre r_1 y el autómata i es tal que $s_i e s_i \in T_i$; en conclusión, o bien cuando no hay competencia por un autómata o cuando el autómata no tiene cambio de estado al ejecutar la acción.

Analizando las relaciones de competición entre reglas de transacción existentes en un sistema está claro que es posible obtener subconjuntos de reglas cuyo disparo en paralelo es seguro y, al contrario, subconjuntos para los que el disparo simultáneo es improbable o imposible. Debido al hecho de que las relaciones débiles, fuertes y muy fuertes se definen usando el estado actual del sistema estas relaciones sólo pueden ser obtenidas y usadas durante la ejecución o simulación del sistema. Sin embargo, si lo que se desea es realizar un análisis previo es posible hacerlo obteniendo las relaciones muy débiles, para las que sólo la definición estática del sistema es suficiente.

4.2.4. Algoritmo de análisis automático de las relaciones

Como se ha comentado, los tres tipos de relaciones muy débiles pueden obtenerse durante la etapa de diseño. Esto implica que, incluso antes de finalizar el modelado del sistema, es posible comenzar a establecer conclusiones acerca del mismo, como por ejemplo identificar agentes que sean cuellos de botella. Otra posibilidad de análisis consiste en identificar los subprocesos independientes del sistema, así como las reglas independientes cuya ejecución no afecta a ningún otro elemento del modelo más que a la propia regla y a sus autómatas vinculados. Para ello se propone a continuación un algoritmo que, integrado en la propia herramienta de diseño y edición, proporciona información muy útil al diseñador para replantear el modelo completo.

Algoritmo 1 Clasificando las reglas de un sistema en conjuntos independientes

1: Sea S_0 el conjunto de reglas de trasnacción que contiene todas las reglas del sistema
2: Sea i un índice con $i = 1$
3: **while** $|S_0| > 1$ **do**
4: define a nuevo conjunto vacío de reglas S_i
5: extrae una regla r_i aleatoriamente de S_0 y añadirla a S_i
6: **for all** regla $r_j \in S_i$ **do**
7: **for all** regla $r_k \in S_0$ **do**
8: **if** $r_j \overset{vw}{\lhd\rhd} r_k$ **then**
9: extraer r_k de S_0 y añadirla a S_i
10: **end if**
11: **end for**
12: **end for**
13: i++;
14: **end while**

En base a las relaciones muy débiles es posible, aplicando el Algoritmo 1 durante

la etapa de diseño, identificar los subconjuntos independientes de reglas. Estos subconjuntos incluirán las reglas (y, por añadidura, los autómatas) que conformarán los subprocesos independientes del sistema, lo que permite a su vez una simplificación inicial del problema aunque este sea muy grande y no se tenga un conocimiento completo del mismo, por ejemplo al integrar las diferentes partes realizadas por diferentes grupos de diseñadores. Si el algoritmo se observa detenidamente resulta bastante sencillo concluir que el coste asintótico del mismo basado en el número de reglas n pertenece al orden de $\Theta(n^2)$.

De acuerdo con el algoritmo mostrado los conjuntos unarios son la reglas independientes del sistema y cada conjunto disjunto de reglas compone un subproceso independiente. En cualquier secuencia de disparo secuencial, el disparo de las reglas independientes no alteran el estado final del resto de autómatas no involucrados con ellas y, por tanto, el estado final del sistema.

Proposición 4 *El estado final al que se llega tras una secuencia de disparo de reglas en un sistema de Autómatas Cooperativos Extendidos es siempre el mismo sin importar el orden de ejecución de cada subproceso representado por los conjuntos disjuntos de reglas obtenidos en el algoritmo 1. Esto sucede siempre que las secuencias internas de disparo de cada subproceso se mantengan en el mismo orden y no cambien.*

Prueba 2 *El estado del sistema puede ser descrito como el estado de cada autómata incluido en él. Sean $\mathbb{S}_{ini}$ y $\mathbb{S}_{fin}$ los estados inicial y final, respectivamente, del sistema. Debido a que cada subproceso independiente implica a un conjunto disjunto de autómatas y reglas de transacción, cada subproceso puede ser considerado como un subsistema independiente. Sea $\mathbb{S}_{ini_p}$ y $\mathbb{S}_{fin_p}$ los estados inicial y final, respectivamente, del subproceso p siendo $0 \leq p < n$ donde n es el número de procesos independientes de los que se compone el sistema completo. Por tanto, el estado inicial del sistema es la unión de los estados iniciales de cada subproceso $\mathbb{S}_{ini} = \cup_{p=0}^{n}(\mathbb{S}_{ini_p})$. Y dado que como se ha establecido previamente, los conjuntos de autómatas y reglas son disjuntos entre los procesos, cada subsistema puede ser aislado del resto y no importa si la ejecución de dichos subprocesos es en paralelo o secuencial o si se intercalan disparos de uno u otro proceso, dado que el estado final también viene dado por la unión de los estados finales de cada subproceso independiente. $\mathbb{S}_{fin} = \cup_{p=0}^{n}(\mathbb{S}_{fin_p})$.*

Es decir, como queda establecido en la Proposición 4 las subsecuencias de disparos de reglas pertenecientes a diferentes conjuntos disjuntos pueden ser intercaladas sin cambiar el estado final del sistema. Así si dado un sistema se realiza el análisis y tras aplicar el algoritmo se obtienen los conjuntos disjuntos de reglas $R_a = \{r_{a_1}, r_{a_2}\}$ y $R_b = \{r_{b_1}, r_{b_2}\}$ se puede asegurar que tras las secuencias de disparo siguientes el estado del sistema es el mismo en todos los casos:

- $r_{a_1} \rightarrow r_{a_2} \rightarrow r_{b_1} \rightarrow r_{b_2}$,
- $r_{a_1} \rightarrow r_{b_1} \rightarrow r_{a_2} \rightarrow r_{b_2}$,
- $r_{a_1} \rightarrow r_{b_1} \rightarrow r_{b_2} \rightarrow r_{a_2}$,
- $r_{b_1} \rightarrow r_{a_1} \rightarrow r_{a_2} \rightarrow r_{b_2}$,
- $r_{b_1} \rightarrow r_{a_1} \rightarrow r_{b_2} \rightarrow r_{a_2}$,

- $r_{b_1} \rightarrow r_{b_2} \rightarrow r_{a_1} \rightarrow r_{a_2}$,

Obviamente, si hubiera alguna regla independiente el resultado final no variaría si se situase su disparo en cualquier posición de la secuencia.

Este resultado puede permitir disgregar un sistema muy complejo completo en un conjunto de sistemas menores e independientes. Desde el mismo instante en el que se comienza a diseãr el sistema es posible aplicar el Algoritmo 1 y obtener el listado de los diferentes procesos independientes del sistema en ese instante así como separar las reglas y autómatas involucrados, para continuar desarrollándolos por separado y reintegrarlos si es preciso más adelante.

4.3. Un ejemplo práctico para analizar

En esta sección se presenta un ejemplo de un sistema en el que el conocimiento del significado de cada agente y de cada regla de transacción es crucial para poder identificar los cuellos de botella o los procesos independientes. Sin embargo, se puede llegar a las mismas conclusiones desconociendo absolutamente dicho significado utilizando tan solo el análisis automático descrito en la Sección 4.2.

Imagínese el esquema simplificado de una compañía dedicada a la fabricación, promoción y venta de ciertos artículos, donde diversos procesos se entremezclan o funcionan independientemente involucrando a difentes agentes dentro del propio sistema. Dado el pequeño tamaño del sistema resultante y del conocimiento preciso de cada proceso es posible identificar los riesgos y las partes independientes. Sin embargo, si el sistema creciese en tamaño y complejidad y fuese necesario realizar el diseño entre un equipo bastante grande, resultaría mucho más complicado disponer de ese conocimiento del sistema completo necesario para realizar ese mismo análisis. Para ilustrar la dificultad del proceso y su solución automática a través del análisis propuesto en la Sección 4.2 se presenta a continuación el ejemplo elegido con los nombres que ilustran cada autómata y regla.

Para el ejemplo descrito a continuación se tendrán en cuenta únicamente siete tipos de autómatas diferentes, que representarán agentes bien diferenciados en el esquema de la compañía descrita. En concreto, los autómatas que componen este modelo son el que representa a los vendedores (*SalesPerson*), los expertos en marketing (*MarketingExpert*), los Creativos (*Creative*), el personal de seguridad (*SecurityOfficer*), El almacén (*Warehouse*), el servicio de Recursos Humanos (*HumanResources*) y el de fabricación (*Manufacturer*). Existen diferentes procesos en el sistema, algunos de los cuales pueden ser independientes y otros están estrechamente interrelacionados; a saber: el servicio de ventas, la fabricación de productos, las campañas de marketing y los controles de seguridad.

Las funcionalidades básicas del sistema son las siguientes. El vendedor recibe algunas órdenes de compra por parte de los clientes. Entonces el vendedor solicita al almacén los productos para entregar la mercancía al cliente. En el almacén se envía una petición a fábrica que a su vez calcula los componentes necesarios para fabricar los productos solicitados y se los solicita, a su vez al almacén. Éste envía las componentes y recibe los productos manufacturados y cuando los tiene se los envía al cliente para que el vendedor pueda cerrar la venta. Cuando esto ocurre el servicio de recursos humanos calcula y añade la comisión por la venta al salario del vendedor. Por otro

lado existen parejas de empleados que trabajan conjuntamente, son los creativos y los expertos en marketing que lanzan campañas nuevas de publicidad y analizan el impacto en el mercado de esas campañas, respectivamente. Finalmente, el personal de seguridad elabora informes periódicos sobre diversas áreas de la empresa y el servicio de Recursos Humanos elabora las órdenes de pago de los salarios cada mes.

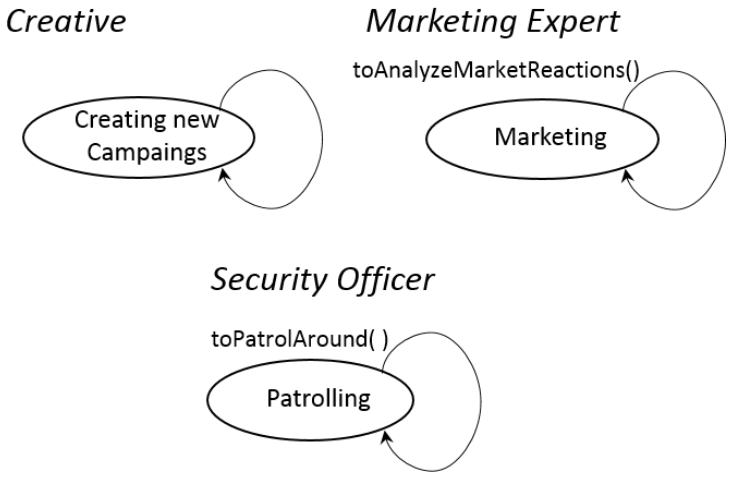


Figura 4.1: Autómatas *Creative*, *Marketing_Expert* y *Security_Officer*, versión gráfica

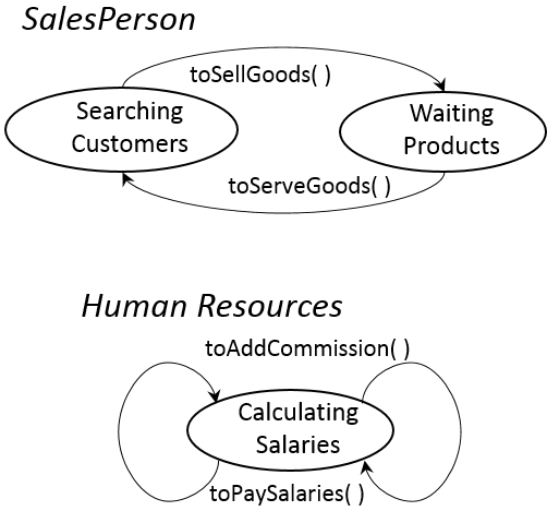


Figura 4.2: Autómatas *SalesPerson* y *Human_Resources*, versión gráfica

En la Figura 4.1 se muestran los autómatas correspondientes a los creativos, a

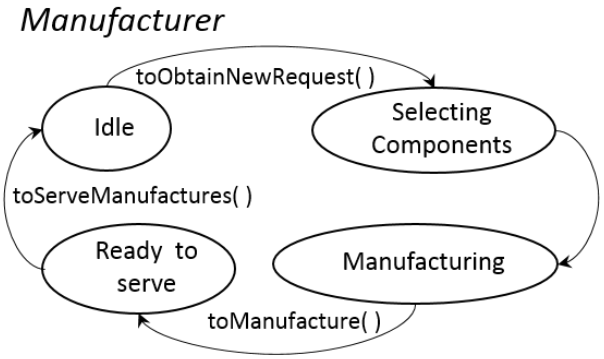


Figura 4.3: Autómata *Manufacturer*, versión gráfica

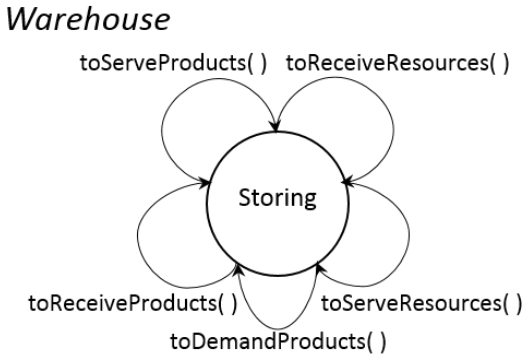


Figura 4.4: Autómata *Warehouse*, versión gráfica

los expertos en marketing y al oficial de seguridad en la versión gráfica. A su vez, en la Figura 4.2 se muestran los autómatas correspondientes a ventas y a recursos humanos. Por otro lado, en la Figura 4.3 se muestra el autómata con más estados del sistema, el autómata correspondiente a fábrica. Finalmente en la Figura 4.4 se muestra el autómata que representa el almacén.

La transcripción de estos autómatas en la versión textual puede verse en la Figura 4.5. Para no complicar el modelo con acciones que oculten las funcionalidades básicas, en la versión gráfica de los mismos se ha omitido el estado inicial y las acciones de creación de los autómatas y, en ambas versiones, las acciones de destrucción de los mismos.

Para este ejemplo (aunque no haya diferencia en el análisis dado que se realiza a priori y, por tanto, no han sido creadas aún las instancias) pueden existir múltiples instancias de los autómatas *Creative* y *Marketing Expert* pero solamente existirá una instancia del resto de tipos de autómatas.

Un sistema de Autómatas Cooperativos Extendidos puede evolucionar si cual-

```
Automaton Creative
{
    dead -> creatingNewCampaigns : create_Creative()
    creatingNewCampaigns -> creatingNewCampaigns : toLaunchNewCampaign()
}

Automaton MarketingExpert
{
    dead -> marketing : create_MarketingExpert()
    marketing -> marketing : toAnalyzeMarketReactions()
}

Automaton SecurityOfficer
{
    dead -> securityChecking : create_Security()
    securityChecking -> securityChecking : toProduceReports()
}

Automaton SalesPerson
{
    dead -> searchingCustomers : create_SalesPerson()
    searchingCustomers -> waitingProducts : toSellGoods()
    waitingProducts -> searchingCustomers : toServeGoods()
}

Automaton HumanResources
{
    dead -> calculatingSalaries : create_HumanResources()
    calculatingSalaries -> calculatingSalaries : toPaySalaries()
    calculatingSalaries -> calculatingSalaries : toAddCommission()
}

Automaton Manufacturer
{
    dead -> idle : create_Manufacturer()
    idle -> selectingComponents : toObtainRequest()
    selectingComponents -> manufacturing : toReceiveComponents()
    manufacturing -> readyToServe : toManufacture()
    readyToServe -> idle : toServeManufactures()
}

Automaton Warehouse
{
    dead -> storing : create_Warehouse()
    storing -> storing : toServeProducts()
    storing -> storing : toReceiveResources()
    storing -> storing : toServeResources()
    storing -> storing : toDemandProducts()
    storing -> storing : toReceiveProducts()
}
```

Figura 4.5: Autómatas del modelo de la compañía, versión textual.

quiera de las reglas de transacción definida se dispara. El disparo de una regla es solamente posible si todas las condiciones de las reglas se cumplen. Cada regla contiene el nombre de la misma, el vector de sincronización, la guarda que contiene las

restricciones que deben cumplir los atributos de los autómatas que participen en el disparo y la actualización de dichos atributos. El autómata que puede realizar las acciones y al que pertenecen los atributos queda inespecificado y se selecciona en tiempo de ejecución dado que es posible que múltiples autómtas puedan efectuar la acción en concreto y tengan los atributos correspondientes con el valor correcto. Si la guarda o la actualización están vacías se representa con la palabra *null*. En este ejemplo, pueden verse los dos tipos de atributos definidos en los Autómatas Cooperativos Extendidos. Primero, los *atributos de pertenencia a una tarea* que pueden encontrarse en los autómatas *Creative* y *Marketing Expert* y que representan el emparejamiento de dichos autómatas. Dado que pueden existir diferentes instancias de ambos autómatas puede establecerse la pertenencia de ambos autómatas a una tarea común con sus respectivos atributos *campaign* que son comprobados en la guarda de la regla de transacción $Marketing\_Campaign$ y serían actualizados en la creación de cada pareja. Un ejemplo del segundo tipo de atributos, los *atributos numéricos*, puede encontrarse en el autómata *SecurityOfficer*. Un atributo numérico se distingue de un atributo de tarea sintácticamente por el símbolo # que le precede. En este caso el atributo denominado $\#repCount$ indica un contador con el número de informes realizados por el servicio de seguridad y, por tanto, pertenece al autómata *Security Officer*.

En la Figura 4.6 se representan las diferentes reglas de transacción de este modelo. En ellas las variables A, B y C representan autómatas. Las reglas de creación han sido deliberadamente omitidas. Si se añadiese una regla de creación global (aunque posteriormente tenga sentido) el análisis final quedaría falseado ya que todos los autómatas aparecerían vinculados a ella y no habría procesos independientes.

Claramente el sistema se puede complicar más y más sólo añadiendo autómatas de diferentes tipos. Por ejemplo, algunos vendedores podrían estar trabajando simultáneamente y en este caso deberían tener atributos pues la comunicación con el almacén y con el servicio de recursos humanos deberían estar conectados mediante una tarea. Tampoco se han representado agentes externos como clientes o proveedores y sólo hay un gestor de almacén y de fabricación.

Si tenemos una idea clara y profunda del sistema en su conjunto es relativamente sencillo extraer algunas conclusiones sobre él, esto es más cierto cuanto más sencillo es el sistema y más profundo nuestro conocimiento sobre el mismo. En el ejemplo resulta sencillo ver a simple vista que las acciones del autómata *Warehouse* se ven implicadas en 5 de las reglas, luego deducir que este autómata puede convertirse en un cuello de botella es una conclusión lógica antes incluso de simular o ejecutar el sistema. Sin embargo, en este caso sólo tenemos unos pocos procesos, siete autómatas y nueve reglas. Un sistema complejo real puede fácilmente componerse de cientos de reglas, autómatas y miles de acciones diferentes, muchas de ellas presentes en más de un tipo de autómata. Con este tipo de sistemas la mera tarea de identificar procesos independientes puede ser una tarea realmente complicada. Es en estos sistemas cuando disponer de una metodología y un proceso automático de análisis se puede apreciar en su totalidad.

Está claro que el estudio completo sólo podría hacerse en tiempo de simulación ya que las relaciones débiles, fuertes y muy fuertes (sobre todo estas últimas que son las que identifican de forma inequívoca la vinculación, concurrencia y competencia). Sin embargo, si el análisis previo permite separar el proceso en otros más pequeños y

```
Rules
{
        name:      Customer_Order
        synch:     A.toSellGoods() + B.toDemandProducts() + C.toObtainRequest()
        guard:     null
        update:    null

        name:      Resources_Demanding
        synch:     A.toReceiveResources()
        guard:     null
        update:    null

        name:      Resources_Dispatching
        synch:     A.toServeResources() + B.toReceiveComponents()
        guard:     null
        update:    null

        name:      Manufacturing_Products
        synch:     A.toManufacture()
        guard:     null
        update:    null

        name:      Products_Dispatching
        synch:     A.toServeManufactures() + B.toReceiveProducts()
        guard:     null
        update:    null

        name:      Security_Tasks
        synch:     A.toProduceReports()
        guard:     null
        update:    A.#repCount++

        name:      Finishing_Sale
        synch:     A.toServeGoods() + B.toServeProducts() + C.toAddCommission()
        guard:     null
        update:    null

        name:      Marketing_Campaign
        synch:     A.toLaunchNewCampaign() + B.toAnalyzeMarketReactions()
        guard:     A.campaign==B.campaign
        update:    null

        name:      Salaries
        synch:     A.toPaySalaries()
        guard:     null
        update:    null
}
```

Figura 4.6: Reglas de Transaccion del ejemplo de la Compañía.

centrar la atención en los candidatos a ser cuellos de botella y transacciones o acciones críticas el trabajo posterior se simplifica considerablemente. El objetivo en cualquier caso es plantear este análisis previo en la etapa más temprana del diseño posible y sin ser necesario un conocimiento heurístico completo del sistema para realizarlo ya que se proporciona un método automático para alcancar las conclusiones esperadas antes incluso de que el sistema esté concluido o en fase de simulación y, desde luego, mucho

antes de su implementación e implantación, etapas durante las cuales la resolución de problemas sería, sin duda, mucho más traumática y costosa.

4.4. Un ejemplo de análisis a priori

El modelo propuesto en la sección anterior es bastante claro y su análisis puede parecer sencillo. Sin embargo, muchas veces los modelos se van construyendo a partir del análisis de requisitos del sistema y sin posteriores refinamientos el resultado puede ser un esquema un tanto caótico y difícil de analizar. En muchos procesos industriales los agentes, procesos, condiciones, etc, vienen codificados con códigos internos cuyo significado no siempre es evidente y realizar el análisis que se propone sin la metodología adecuada puede ser una tarea ardua.

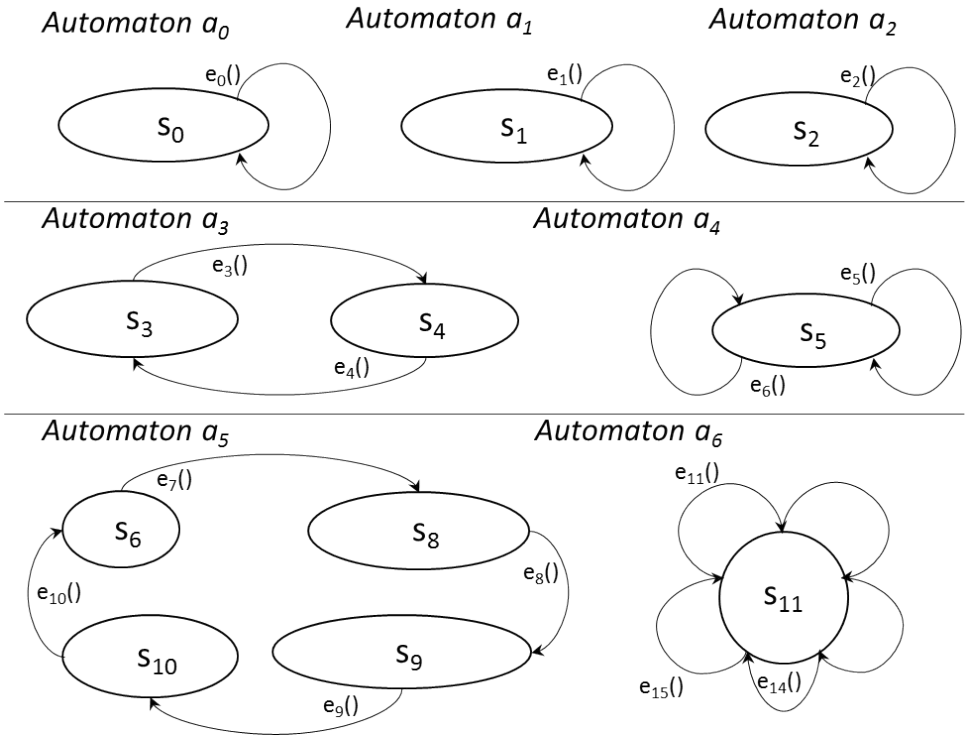


Figura 4.7: Autómatas del modelo codificado. Versión gráfica.

Para ilustrar esto y dado que en el ejemplo propuesto los nombres de acciones y autómatas aportan mucha información al diseñador, a continuación se presenta el mismo problema donde se han eliminado y sustituido por referencias alfabéticas. De esta manera se muestra que el resultado del análisis obtenido de forma automática es similar al que el conocimiento del significado de los procesos completos aportaría al diseñador.

En la Figura 4.7 se muestra de manera gráfica el mismo conjunto de autómatas visto en las Figuras 4.1, 4.2, 4.3 y 4.4 pero con los nombres codificados de manera "neutral" con el código alfanumérico antes mencionado. En esta versión ya no resulta en absoluto sencillo identificar qué hacen estos agentes ni prever con quién se relacionarían. Del mismo modo, en la Figura 4.8 aparece la descripción de los mismos autómatas en su versión textual.

```
Automaton a0
{
    dead -> s0 : create_a0()
    s0 -> s0 : e0()
}
Automaton a1
{
    dead -> s1 : create_a1()
    s1 -> s1 : e1()
}
Automaton a2
{
    dead -> s2 : create_a2()
    s2 -> s2 : e2()
}
Automaton a3
{
    dead -> s3 : create_a3()
    s3 -> s4 : e3()
    s4 -> s3 : e4()
}
Automaton a4
{
    dead -> s5 : create_a4()
    s5 -> s5 : e5()
    s5 -> s5 : e6()
}
Automaton a5
{
    dead -> s6 : create_a5()
    s6 -> s7 : e7()
    s7 -> s8 : e8()
    s8 -> s9 : e9()
    s9 -> s10 : e10()
}
Automaton a6
{
    dead -> s11 : create_a6()
    s11 -> s11 : e11()
    s11 -> s11 : e12()
    s11 -> s11 : e13()
    s11 -> s11 : e14()
    s11 -> s11 : e15()
}
```

Figura 4.8: Autómatas del modelo codificado. Versión textual.

Del mismo modo, las reglas definidas en la Figura 4.6 se han codificado con el mismo código alfanumérico descrito, quedando la descripción de reglas de transacción

del sistema que puede verse en la Figura 4.9.

```
name:      r0
synch:     A.e3() + B.e14() + C.e7()
guard:     null
update:    null

name:      r1
synch:     A.e12()
guard:     null
update:    null

name:      r2
synch:     A.e13() + B.e8()
guard:     null
update:    null

name:      r3
synch:     A.e9()
guard:     null
update:    null

name:      r4
synch:     A.e10() + A.e15()
guard:     null
update:    null

name:     r5
synch:      A.e2()
guard:     null
update:    A.#att1++

name:      r6
synch:     A.e4() + B.e11() + C.e5()
guard:     null
update:    null

name:      r7
synch:     A.e0() + B.e1()
guard:     A.att2==B.att3
update:    null

name:      r8
synch:     A.e6()
guard:     null
update:    null
```

Figura 4.9: Reglas del modelo codificado.

Aplicando las definiciones descritas en la Sección 4.4 se obtienen las relaciones siguientes. Es adecuado recordar que las relaciones que se pueden buscar en esta fase del diseño son las relaciones muy débiles, dado que aún no se han inicializado valores en los atributos y, por tanto, no es posible evaluar las guardas y actualizaciones.

1. Relaciones de Vinculación entre autómatas y reglas a través de acciones:

$$a_0 \overset{vw}{\rhd\!\rhd}_{e_0} r_7, \quad a_1 \overset{vw}{\rhd\!\rhd}_{e_1} r_7, \quad a_2 \overset{vw}{\rhd\!\rhd}_{e_2} r_5, \quad a_3 \overset{vw}{\rhd\!\rhd}_{e_3} r_0, \quad a_3 \overset{vw}{\rhd\!\rhd}_{e_4} r_6,$$
$$a_4 \overset{vw}{\rhd\!\rhd}_{e_5} r_6, \quad a_4 \overset{vw}{\rhd\!\rhd}_{e_6} r_8, \quad a_5 \overset{vw}{\rhd\!\rhd}_{e_7} r_0, \quad a_5 \overset{vw}{\rhd\!\rhd}_{e_8} r_2, \quad a_5 \overset{vw}{\rhd\!\rhd}_{e_9} r_3,$$
$$a_5 \overset{vw}{\rhd\!\rhd}_{e_{10}} r_4, \quad a_6 \overset{vw}{\rhd\!\rhd}_{e_{11}} r_6, \quad a_6 \overset{vw}{\rhd\!\rhd}_{e_{12}} r_1, \quad a_6 \overset{vw}{\rhd\!\rhd}_{e_{13}} r_2, \quad a_6 \overset{vw}{\rhd\!\rhd}_{e_{14}} r_0,$$
$$a_6 \overset{vw}{\rhd\!\rhd}_{e_{15}} r_4.$$

A la vista de las relaciones de vinculación y sin más información ya puede deducirse que probablemente exista un cuello de botella en el autómata a_6 considerando que hay nueve reglas del sistema que lo están relacionando con r_0, r_1, r_2, r_4 y r_6 y del resto de reglas hay 3 que en su vector de sincronización *synch* sólo tienen una acción (con lo que sólo involucran a un autómata) y es posible que otro tanto ocurra con el autómata a_5 puesto que también está relacionado con 4 reglas.

Un análisis más pormenorizado del problema detectaría que el autómata a_6 está relacionado con más reglas, pero la ejecución de cualquiera de ellas no impide la ejecución de las demás ya que no hay cambio de estado. Sin embargo, la ejecución de una regla con el autómata a_5 provoca un cambio de estado que bloquea la ejecución de tres reglas y habilita una.

Sin embargo, el análisis permite descartar de inmediato al resto de autómatas que sin duda no serán motivo de un cuello de botella.

Del mismo modo, a partir de las relaciones de vinculación se calculan las relaciones de concurrencia entre autómatas y el resultado de la búsqueda de las mismas es el siguiente:

2. Relaciones de Concurrencia entre autómatas:

$$a_0 \overset{vw}{\bowtie}_{r_7} a_1, \quad a_3 \overset{vw}{\bowtie}_{r_0} a_5, \quad a_3 \overset{vw}{\bowtie}_{r_0} a_6, \quad a_3 \overset{vw}{\bowtie}_{r_6} a_4, \quad a_4 \overset{vw}{\bowtie}_{r_6} a_6,$$
$$a_5 \overset{vw}{\bowtie}_{r_0} a_6, \quad a_5 \overset{vw}{\bowtie}_{r_2} a_6, \quad a_5 \overset{vw}{\bowtie}_{r_4} a_6.$$

El hecho de que el autómata a_6 pueda concurrir con otros autómatas en cinco de las nueve relaciones de concurrencia encontradas confrma que, efectivamente, a_6 puede ser sin duda el cuello de botella de todo el proceso. A estas alturas ya se sabe que hay que prestar un especial cuidado y una estrecha vigilancia a este autómata y que sería adecuado cambiar el diseño del proceso o crear copias del mismo para evitar el bloqueo de todo el proceso. También puede verse que los autómatas a_0 y a_1 sólo son concurrrentes entre ellos y con ninguno más y que el autómata a_2 no es concurrente con nadie.

A continuación se obtienen las reglas de competencia entre reglas que permitirán aplicar el algoritmo de búsqueda de procesos independientes y de conjuntos unarios.

3. Relaciones de Competencia entre reglas:

$$r_0 \overset{vw}{\lhd\!\rhd} r_1, \quad r_0 \overset{vw}{\lhd\!\rhd} r_2, \quad r_0 \overset{vw}{\lhd\!\rhd} r_3, \quad r_0 \overset{vw}{\lhd\!\rhd} r_4, \quad r_0 \overset{vw}{\lhd\!\rhd} r_6,$$
$$r_1 \overset{vw}{\lhd\!\rhd} r_2, \quad r_1 \overset{vw}{\lhd\!\rhd} r_4, \quad r_1 \overset{vw}{\lhd\!\rhd} r_6, \quad r_2 \overset{vw}{\lhd\!\rhd} r_3, \quad r_2 \overset{vw}{\lhd\!\rhd} r_4,$$
$$r_4 \overset{vw}{\lhd\!\rhd} r_6, \quad r_6 \overset{vw}{\lhd\!\rhd} r_8,$$

Tras obtenerlas el resultado del Algoritmo 1 son los siguientes conjuntos disjuntos de reglas: $R_0 = \{r_5\}$, $R_1 = \{r_7\}$ and $R_2 = \{r_0,\ r_1,\ r_2,\ r_3,\ r_4,\ r_6,\ r_8\}$.

Puede verse que R_0 y R_1 contienen reglas independientes del sistema. Estas reglas se pueden situar en cualquier secuencia de disparo que afecte al resto del modelo sin que este se vea afectado, por tanto, la primera simplificación consiste en la separación del sistema completo en partes más pequeñas que pueden ser modeladas individualmente.

La regla r_7 incluye a parejas de instancias de los autómatas a_0 and a_1. Dichas parejas están conectadas por una tarea y un vistazo a la guarda y actualización de la regla permite comprobar que cada par se relaciona no "muy débilmente" sino en realidad dado que los valores de los atributos no se actualizan ambos autómatas son siempre los mismos y, por tanto, la relación es "muy fuerte", ya que en cada disparo de la regla r_7 siempre se relacionarán la misma pareja de autómatas. Bajo estas condiciones queda claro que estas parejas de autómatas son candidatos perfectos a la unificación en un único agente dado que no pueden evolucionar por separado. Por añadidura, ambos autómatas no sufren cambio de estado alguno con el disparo de la regla, por lo que a efectos de simulación tampoco tiene demasiado sentido su existencia.

Algo diferente ocurre con el autómata a_2 en la regla r_5 puesto que, si bien está claro que es independiente del resto, su existencia sí está justificada puesto que sí sufre un cambio de estado (es bueno recordar en este punto que los atributos numéricos son una extensión del conjunto de estados del autómata) ya que en la guarda el valor del atributo numérico se ve incrementado en una unidad.

Aplicando el análisis automático a priori definido en la Sección 4.2 sobre el sistema descrito en esta sección se ha obtenido información relevante acerca de un cuello de botella, tres procesos independientes (uno de ellos que podría ser reducido o incluso eliminado del sistema sin perjuicio para el resto) posibles reducciones o desapariciones de reglas y autómatas, etc. Analizando a la manera tradicional el modelo de la Sección 4.3 es posible que se hubiera llegado a las mismas conclusiones, pero únicamente porque el sistema es muy pequeño y accesible y su conocimiento profundo no requiere de mucho esfuerzo. Sin embargo, si representásemos una organización real con detalle el número de agentes y transiciones y, por tanto, de autómatas y reglas de transacción aumentaría exponencialmente. Otro tanto pasaría, obviamente, con las relaciones que se establecerían entre dichos agentes y las reglas.

El objetivo último de este análisis es establecer las bases para un medio estandarizado (y automático en lo posible) para simplificar los sistemas modelizados con los Autómatas Cooperativos Extendidos en busca de su forma canónica. Este sería un primer paso para la definición de la relación de bisimulación [15] en los Autómatas Cooperativos Extendidos.

Otros trabajos destinados a la simplificación y análisis de modelos de sistemas suelen establecerse en posteriores fases de diseño, como por ejemplo durante la simulación del mismo como [128], [29] y [123] o en el análisis heurístico del diseñador, sobre la sintaxis o semántica del modelo aplicando por ejemplo reglas de *model checking* [48].

Sin embargo, en esta propuesta el análisis se efectúa de manera automática durante el propio diseño del sistema, no habiendo encontrado análisis similares en tan temprana fase del ciclo de vida de estos modelos en la literatura. Simplificar tan pronto un sistema y evitar posibles riesgos puede facilitar la consecución de mejores solucio-

nes y alternativas de procesos que quedarían ocultos entre amasijos de información redundante o no relacionada con el lugar donde se produce el problema.

Este análisis compuesto por la categorización y cálculo de las relaciones muy débiles de vinculación, concurrencia y competencia así como el Algoritmo 1 ha sido integrado en una herramienta de edición y análisis de modelos tal y como se describe en el próximo capítulo.

Capítulo 5

ECA-Tool, una herramienta de Edición y Análisis

5.1. La herramienta ECA-TOOL

Durante el modelado de un sistema gran parte del trabajo de un diseñador consiste en plasmar las funcionalidades del sistema y los agentes involucrados en la sintaxis del modelo elegido, de ahí la existencia de multitud de herramientas de modelado.

Las herramientas de modelado permiten crear un "simulacro" del sistema, a bajo costo y riesgo mínimo. A bajo costo porque, al fin y al cabo, es un conjunto de gráficos y textos que representan el sistema, pero no son el sistema físico real (el cual es considerablemente más costoso). Además minimizan los riesgos, porque los cambios que se deban realizar (por errores o cambios en los requerimientos), se pueden realizar más fácil y rápidamente sobre el modelo que sobre el sistema ya implementado.

Las herramientas de modelado permiten concentrarse en ciertas características importantes del sistema, prestando menos atención a otras. Los modelos resultantes son una buena forma de determinar si están representados todos los requerimientos del sistema, como también saber si el analista comprendió qué hará el sistema. En el caso de los Autómatas Cooperativos Extendidos se ha desarrollado una herramienta, la ECA-Tool, disponible online en [70] no sólo para la redacción y edición del modelo, sino también para efectuar automáticamente el análisis descrito en el Capítulo 4.

La herramienta permite al usuario ir paso a paso añadiendo las diferentes componentes de un sistema de forma fácil e intuitiva y avisa de posibles incongruencias así como corrige errores básicos. Además, en cualquier momento el diseñador puede efectuar el análisis y ver qué componentes están resultando críticas o qué subprocesos se han ido agregando. Cuando el programador considera que ha finalizado el análisis y se encuentra satisfecho con el resultado obtenido y ha solucionado los problemas como los cuellos de botella del sistema puede pasar a la fase de simulación. Como el análisis a priori ya ha facilitado al diseñador la información relevante que ha obtenido la detección de esos problemas o la simplificación se adelanta desde la fase de simulación a la fase de diseño, evitando tener que regresar a esta una vez se detecten los problemas durante aquella.

Aunque en este apartado no se entra en el detalle de la implementación de la

herramienta, esta ha sido desarrollada en Java [51] y un esquema de las clases que la componen puede verse en la Figura 5.1. Brevemente, la herramienta se compone, de una parte, de tratamiento de texto para la identificación de los modelos tanto desde fichero de texto como introducidos por el usuario, una interfaz gráfica donde se pueden introducir por separado cada una de las componentes del sistema y una parte de análisis del modelo introducido. Los resultados pueden verse y guardarse tanto en versión textual como en versión web.

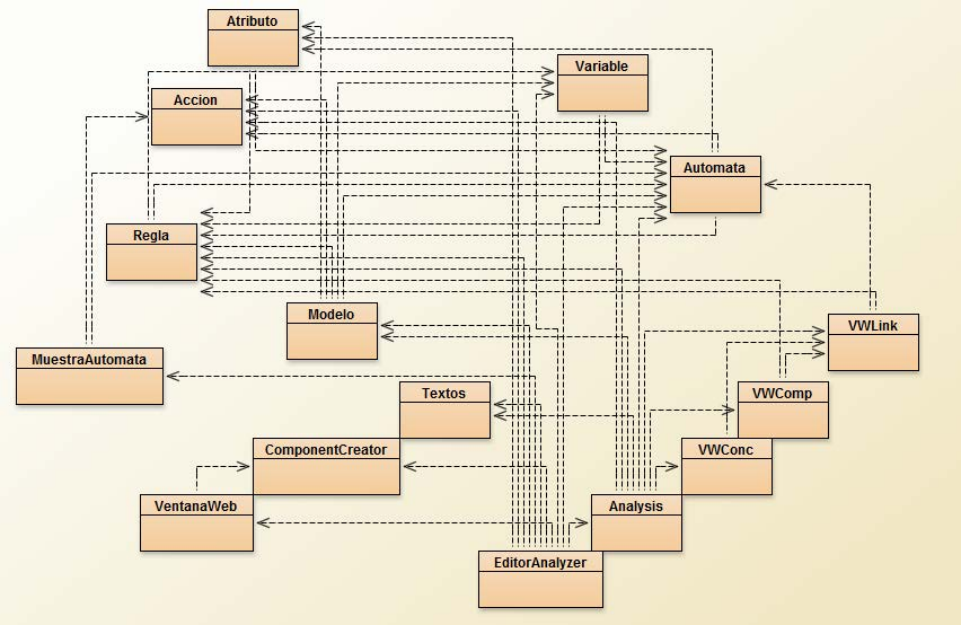


Figura 5.1: Esquema de las clases de la Herramienta ECA-Tool

La herramienta tiene un menú básico de carga y guardado de modelos en archivos de texto con extensión ".mdl" un menú de visualización del texto del modelo y del análisis y un menú de cambio de idioma y de información sobre la versión y ayuda. Cuando un usuario desea comenzar a definir un modelo debe empezar por declarar el nombre y descripción del mismo. A continuación puede comenzar con la definición de autómatas o reglas, si bien resulta preferible empezar por aquellos.

5.2. ECA-Tool: Edición y detección de errores

Los autómatas se describen de manera intuitiva, indicando:

- su nombre,
- una descripción opcional,
- las acciones contiene (con formato *estado_inicio -> estado_fin : acción*) no siendo necesaria la definición por separado de los estados y las acciones.

Si el usuario olvida la acción y el estado *dead* del cual parten obligatoriamente todos los autómatas, el editor lo añade automáticamente. La lista de autómatas está siempre accesible y es posible en cualquier momento añadir, modificar o eliminar autómatas. El editor muestra información, además, de los estados que han ido apareciendo así como de los atributos (que aparecerán en las reglas por primera vez) que corresponden al hacer matching entre las reglas y los autómatas, añadiéndolos automáticamente.

En la Figura 5.2 puede verse la edición en la herramienta ECA-Tool del autómata *SecurityOfficer* del ejemplo descrito en la Sección 4.3. En dicha figura pueden verse varios elementos, pero solo alguno de los cuales es directamente editable por el usuario. Por ejemplo, la lista de autómatas no se puede modificar directamente, pero sí es posible añadir, seleccionar, editar y borrar un autómata a través de los botones proporcionados. Solo el nombre del autómata, la descripción y la lista de acciones son directamente editables por el usuario (en caso de que no se cumpla con la sintaxis el sistema da un mensaje de error en el área de mensajes) mientras que la lista de estados del autómata y de atributos se extraen directamente de las listas de acciones y las reglas, respectivamente.

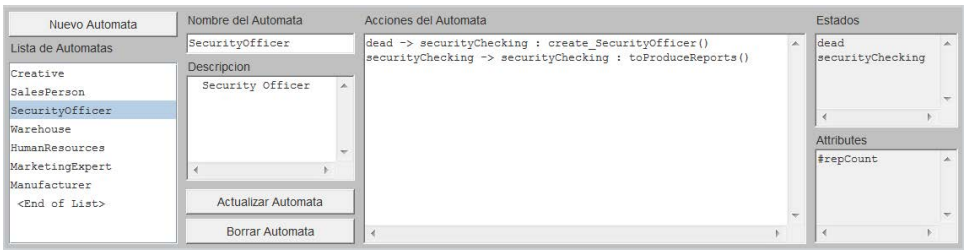


Figura 5.2: Edición del Autómata *SecurityOfficer* en ECA-Tool

En cuanto a las reglas, también se describen indicando:

- Su nombre.
- El vector de sincronización. Cada componente del vector se separa del resto con el signo + y sigue el formato *Var.acción*
- La Guarda. Consiste en un número entre 0 y n de expresiones donde, si se trata de atributos de tarea la parte izquierda será siempre una expresión con el formato $Var_i.atributo_i$ y el operador siempre será o bien == o bien ! = y la parte izquierda puede ser un valor u otra expresión $Var_j.atributo_j$. En el caso de los atributos de Numéricos la parte izquierda será siempre una expresión con el formato $Var_i.\#atributo_i$, el operador será uno de entre ==, ! =, <, ≤, > y ≥. La parte izquierda puede ser un valor u otra expresión $Var_j.\#atributo_j$. En el caso de que no haya restricciones se utiliza el valor *null*.
- La actualización. Que consiste en un número entre 0 y n de expresiones que pueden ser (para los atributos de tarea) asignaciones, donde la parte izquierda será siempre una expresión con el formato $Var_i.atributo_i$ y la parte derecha un valor,

otra expresión $Var_i.atributo_j$ o el operador *new*. En el caso de atributos numéricos además de asignaciones pueden ser incrementos o decrementos, bien sea directamente los operadores unarios ++ y −−, bien sea + = o − = en cuyo caso la parte izquierda será siempre una expresión con el formato $Var_i.\#atributo_i$ y la parte derecha podrá ser a su vez un valor u otra expresión $Var_j.\#atributo_j$. En el caso de que no haya actualizaciones se utiliza el valor *null*.

Al describir las reglas se realiza una comprobación de si las acciones del vector de sincronización están presentes en la definición de al menos uno de los autómatas descritos. En caso contrario, el editor avisará si no encuentra ninguna correspondencia e ignorará la componente del vector de sincronización en el análisis. En la Figura 5.2 puede verse la edición en la herramienta ECA-Tool de la regla de transacción *Security_Tasks* del ejemplo descrito en la Sección 4.3. De nuevo, la parte de la lista de reglas no es modificable directamente pero sí es posible añadir, seleccionar, editar y borrar una regla con los botones que se proporcionan. El nombre, el vector de sincronización, la guarda y la actualización son directamente editables, pero la lista de variables que aparecen y que servirán para ligar las reglas con los autómatas vinculados (por las acciones y los atributos) se obtiene de la información introducida en el resto de campos.

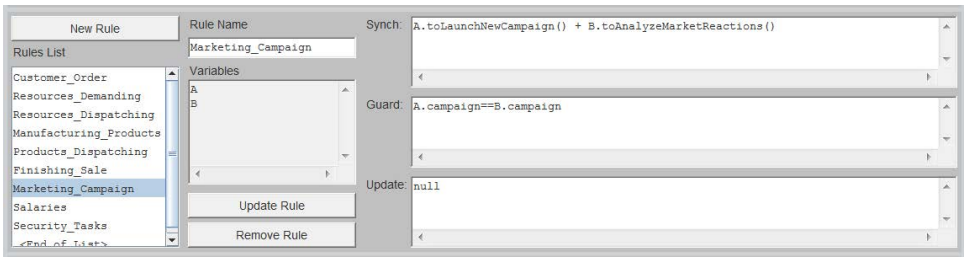


Figura 5.3: Edición de la Regla *Security_Tasks* en ECA-Tool

5.3. ECA-Tool: Análisis del modelo

Una vez se han introducido al menos el nombre y la descripción del modelo y al menos una regla y un autómata es posible realizar el análisis del mismo según lo visto en el Capítulo 4, simplemente seleccionando la opción de menú *Análisis a priori* o *Guardar Análisis* como en la Figura 5.4.

Dicho análisis puede generarse automáticamente en cualquier momento, si bien resulta adecuado tener en cuenta que si se realiza demasiado pronto es posible que haya reglas o autómatas que aún no han sido introducidos y que, por tanto, el resultado del análisis representa una foto fija sobre el sistema tal cual está definido. El hecho de añadir reglas nuevas que puedan competir con otras o autómatas con acciones presentes en reglas preexistentes puede producir que lo que inicialmente sea considerado un cuello de botella deje de serlo o lo que se ha identificado como una regla independiente o un subproceso separado acabe siendo parte integrante de otro proceso mayor.

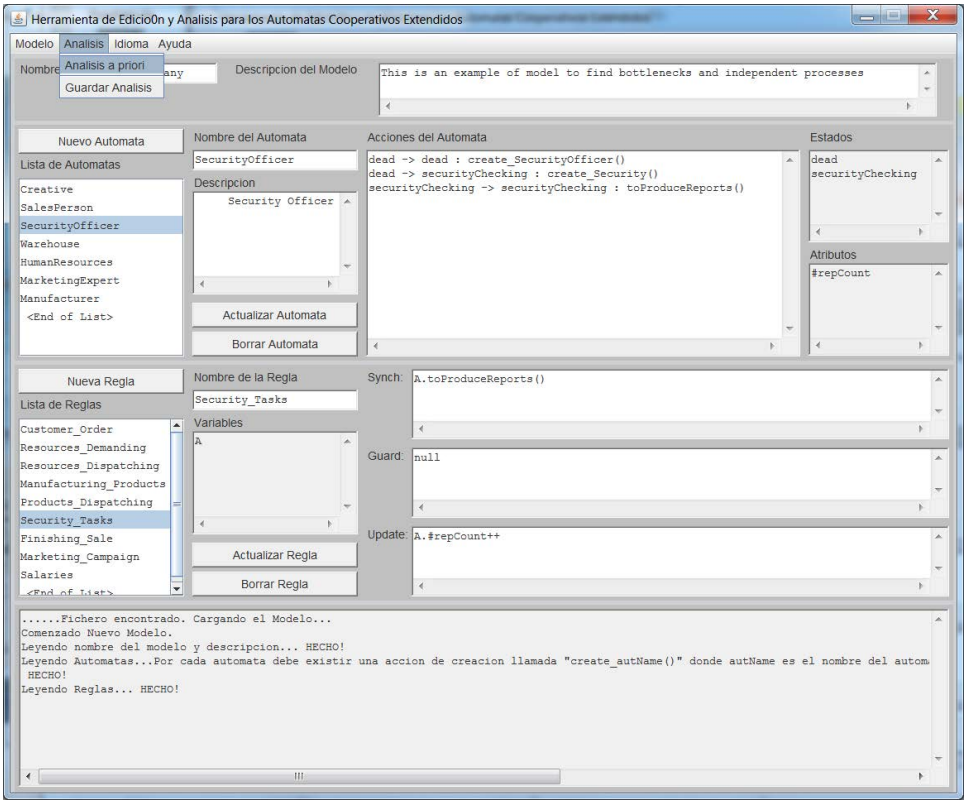


Figura 5.4: Seleccion del análisis en el menú del ECA-Tool

El resultado del análisis automático producido por la herramienta ECA-Tool se muestra en pantalla o se guarda en un fichero con formato de texto según la opción elegida en el menú y tiene la siguiente estructura:

- relaciones de vinculación muy débiles, mostrando el autómata, la regla y la acción a través de la que vinculan;
- relaciones de concurrencia muy débiles, mostrando los dos autómatas, la regla a la que se vinculan, las acciones de cada autómata y la posición que ocupan dichas reglas en el vector de sincronización de la regla;
- relaciones de competencia muy débil, mostrando las dos reglas que compiten y el autómata por el cual lo hacen.
- resultados del algoritmo, mostrando los conjuntos de reglas que conforman los procesos independientes, así como los autómatas vinculados a esos procesos;
- un resumen indicando con cuántas reglas está relacionado con cada autómata, con cuántos otros autómatas es concurrente y cada regla con cuántas reglas compite.

Aplicando la herramienta sobre el sistema descrito en la Sección 4.3 el resultado es un informe en modo texto que para su análisis se ha dividido en 4 figuras que se muestran a continuación. En primer lugar se obtienen las relaciones de vinculación entre autómatas y reglas de transacción como se muestra en la Figura 5.5. Sólo analizando esta parte ya se obtienen los primeros resultados:

- El autómata *Warehouse* está vinculado muy débilmente con las reglas *Customer_Order*, *Resources_Demanding*, *Resources_Dispatching*, *Products_Dispatching* y *Finishing_Sale*;
- simultáneamente, el autómata *Manufacturer* está vinculado muy débilmente con las reglas *Customer_Order*, *Resources_Dispatching*, *Products_Dispatching* y *Manufacturing_Products*.

```
Very Weak Linking Relations
  Automaton          Rule                          Action
  ---------------    ----------------------        ----------------------------
  Creative         >> Marketing_Campaign     through  toLaunchNewCampaign()()
  SalesPerson      >> Customer_Order         through  toSellGoods()()
  SalesPerson      >> Finishing_Sale         through  toServeGoods()()
  SecurityOfficer >> Security_Tasks          through  toProduceReports()()
  Warehouse        >> Customer_Order         through  toDemandProducts()()
  Warehouse        >> Resources_Demanding    through  toReceiveResources()()
  Warehouse        >> Resources_Dispatching  through  toServeResources()()
  Warehouse        >> Products_Dispatching   through  toReceiveProducts()()
  Warehouse        >> Finishing_Sale         through  toServeProducts()()
  HumanResources   >> Finishing_Sale         through  toAddCommission()()
  HumanResources   >> Salaries               through  toPaySalaries()()
  MarketingExpert >> Marketing_Campaign      through  toAnalyzeMarketReactions()()
  Manufacturer     >> Customer_Order         through  toObtainRequest()()
  Manufacturer     >> Resources_Dispatching  through  toReceiveComponents()()
  Manufacturer     >> Manufacturing_Products through  toManufacture()()
  Manufacturer     >> Products_Dispatching   through  toServeManufactures()()
```

Figura 5.5: Resultados del Análisis: Relaciones de Vinculación

A continuación se muestran las relaciones de concurrencia entre autómatas en la Figura 5.6 (Nótese que se ha reducido el tamaño de letra para mantener el formato de salida). En esta parte puede confirmarse parte de lo observado en la Figura 5.5 de tal manera que:

- El autómata *Warehouse* es concurrente con los autómatas *SalesPerson*, *Manufacturer* y *HumanResources*.

```
Very Weak Concurrency Relations
  Automaton1         Automaton2         Rule                    Action1            -pos1,Action2               -pos2
  ---------------    ---------------    ---------------------   ---------------------,---------------------------
  Creative         >< MarketingExpert in Marketing_Campaign     ( toLaunchNewCampaign()-0,toAnalyzeMarketReactions()-1)
  SalesPerson      >< Warehouse       in Customer_Order         (         toSellGoods()-0,        toDemandProducts()-1)
  SalesPerson      >< Manufacturer    in Customer_Order         (         toSellGoods()-0,         toObtainRequest()-2)
  SalesPerson      >< Warehouse       in Finishing_Sale         (        toServeGoods()-0,         toServeProducts()-1)
  SalesPerson      >< HumanResources  in Finishing_Sale         (        toServeGoods()-0,         toAddCommission()-2)
  Warehouse        >< Manufacturer    in Customer_Order         (    toDemandProducts()-1,         toObtainRequest()-2)
  Warehouse        >< Manufacturer    in Resources_Dispatching  (    toServeResources()-0,     toReceiveComponents()-1)
  Warehouse        >< Manufacturer    in Products_Dispatching   (   toReceiveProducts()-1,     toServeManufactures()-0)
  Warehouse        >< HumanResources  in Finishing_Sale         (     toServeProducts()-1,         toAddCommission()-2)
```

Figura 5.6: Resultados del Análisis: Relaciones de Concurrencia

En la Figura 5.7 se muestran las relaciones de competencia entre reglas obtenidad a continuación en el análisis. Con esto puede observarse que existen relaciones de competencia entre reglas de transacción principalmente por dos de los autómatas del sistema.

- más concretamente, hay dos autómatas que son, claramente, motivo de competencia entre reglas: *Warehouse* (por el que compiten hasta en 10 ocasiones) y *Manufacturer* (por el que compiten en 6 ocasiones).

```
Very Weak Competence Relations
  Rule1                    Rule2                      Automaton
  ----------------------   ----------------------     ---------------
  Customer_Order         <> Finishing_Sale         for SalesPerson
  Customer_Order         <> Resources_Demanding    for Warehouse
  Customer_Order         <> Resources_Dispatching  for Warehouse
  Customer_Order         <> Products_Dispatching   for Warehouse
  Customer_Order         <> Finishing_Sale         for Warehouse
  Resources_Demanding    <> Resources_Dispatching  for Warehouse
  Resources_Demanding    <> Products_Dispatching   for Warehouse
  Resources_Demanding    <> Finishing_Sale         for Warehouse
  Resources_Dispatching  <> Products_Dispatching   for Warehouse
  Resources_Dispatching  <> Finishing_Sale         for Warehouse
  Products_Dispatching   <> Finishing_Sale         for Warehouse
  Finishing_Sale         <> Salaries               for HumanResources
  Customer_Order         <> Resources_Dispatching  for Manufacturer
  Customer_Order         <> Manufacturing_Products for Manufacturer
  Customer_Order         <> Products_Dispatching   for Manufacturer
  Resources_Dispatching  <> Manufacturing_Products for Manufacturer
  Resources_Dispatching  <> Products_Dispatching   for Manufacturer
  Manufacturing_Products <> Products_Dispatching   for Manufacturer
```

Figura 5.7: Resultados del Análisis: Relaciones de Competencia

Finalmente en la Figura 5.8 se muestra la parte del análisis con el resultado del algoritmo y el resumen de resultados. La interpretación de este último texto de salida es muy sencilla ya que a primera vista se puede deducir que:

- aplicando el algoritmo se obtienen 3 subprocesos independientes, dos de los cuales son unarios;
- en el resumen final puede verse que, claramente, el autómata *Warehouse* está relacionado con 5 reglas del sistema;
- el autómata *Manufacturer* le sigue muy de cerca (está relacionado con 4 reglas) haciendo de estos dos autómatas serios candidatos a ser cuellos de botella del sistema.
- Analizando con más detalle cada autómata puede verse que, de los dos, el autómata *Manufacturer* tiene más posibilidades de provocar dichos cuellos de botella ya que tiene cuatro estados distintos por lo que sólo podrá efectuar una de las reglas en cada instante. Mientras que, el autómata *Warehouse* tiene un único estado y por tanto podría atender a cualquiera de las reglas en igualdad de condiciones. Es decir, que si en el sistema hubiese varios autómatas *Warehouse* no habría gran efecto en el funcionamiento del sistema, mientras que, si hubiese varios *Manufacturer* sí se podrían llevar varios procesos de fabricación y venta simultáneos.

```
Results of the Algorithm:
-----------------------------------------------------------------
  3 independent processes found:
 Set of Rules S_0  = { Security_Tasks }
 This process includes the set of automata:  { SecurityOfficer }

 Set of Rules S_1  = { Customer_Order, Resources_Demanding, Manufacturing_Products, Finishing_Sale,
                       Resources_Dispatching, Products_Dispatching, Salaries }
 This process includes the set of automata:  { HumanResources,
                                               Manufacturer,
                                               SalesPerson,
                                               Warehouse }

 Set of Rules S_2  = { Marketing_Campaign }
 This process includes the set of automata:  { Creative,
                                               MarketingExpert }


Summarizing:
-----------------------------------------------------------------
  Automaton Creative        is vw_linked with  1 over  9 rules.
  Automaton SalesPerson     is vw_linked with  2 over  9 rules.
  Automaton SecurityOfficer is vw_linked with  1 over  9 rules.
  Automaton Warehouse       is vw_linked with  5 over  9 rules.
  Automaton HumanResources  is vw_linked with  2 over  9 rules.
  Automaton MarketingExpert is vw_linked with  1 over  9 rules.
  Automaton Manufacturer    is vw_linked with  4 over  9 rules.
  Automaton Creative        is vw_concurrent with other  1 Automata.
  Automaton SalesPerson     is vw_concurrent with other  3 Automata.
  Automaton SecurityOfficer is vw_concurrent with other  0 Automata.
  Automaton Warehouse       is vw_concurrent with other  3 Automata.
  Automaton HumanResources  is vw_concurrent with other  2 Automata.
  Automaton MarketingExpert is vw_concurrent with other  1 Automata.
  Automaton Manufacturer    is vw_concurrent with other  2 Automata.
  Rule Customer_Order         is vw_competing with other  5 rules.
  Rule Resources_Demanding    is vw_competing with other  4 rules.
  Rule Resources_Dispatching  is vw_competing with other  5 rules.
  Rule Manufacturing_Products is vw_competing with other  3 rules.
  Rule Products_Dispatching   is vw_competing with other  5 rules.
  Rule Security_Tasks         is vw_competing with other  0 rules.
  Rule Finishing_Sale         is vw_competing with other  5 rules.
  Rule Marketing_Campaign     is vw_competing with other  0 rules.
  Rule Salaries               is vw_competing with other  1 rules.
```

Figura 5.8: Resultados del Análisis: Algoritmo y Resumen

Con esto finaliza este capítulo sobre la herramienta de edición y análisis automático. En el siguiente capítulo se muestra la herramienta desarrollada para la simulación de sistemas modelados con los Autómatas Cooperativos Extendidos: el *Extended Cooperating Automata Simulator*.

Capítulo 6

El Simulador

En su día construimos un simulador [55, 56] en Prolog [34] para el formalismo de la Máquina Química Abstracta [16]. En esta ocasión se ha construido un simulador en [51], destinado a la prueba de modelos escritos bajo el formalismo de los Autómatas Cooperativos [5] y bajo el formalismo de los Autómatas Cooperativos Extendidos [65], toda suerte que cualquier AC es lo mismo que un ACE pero sin hacer uso de los atributos numéricos.

El simulador construye, en base a un archivo de texto que contiene la definición de los autómatas y de las reglas de transacción, una estructura de objetos que va evolucionando según dichas reglas de transacción puedan o no dispararse (en base a sus guardas) y a las transformaciones que las actualizaciones de dichas guardas vayan generando.

La mejor manera de mostrar el funcionamiento del simulador, es mediante un ejemplo práctico. Vamos, en principio, a mostrar la ejecución de un sistema que resuelve un problema conocido de sistemas distribuidos, paralelismo y concurrencia. El problema de la comida de los filósofos.

Este problema, modelizado con autómatas cooperativos tiene los autómatas tipos de autómatas *fork* y *philosopher* cuyo diagrama de estados se puede ver en las Figuras 6.1 y 6.2

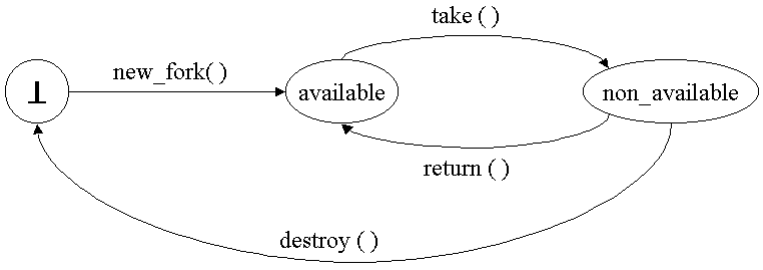


Figura 6.1: Diagrama de estados y transiciones del autómata *Fork*

De una manera informal el problema es el siguiente. Tenemos una mesa con un

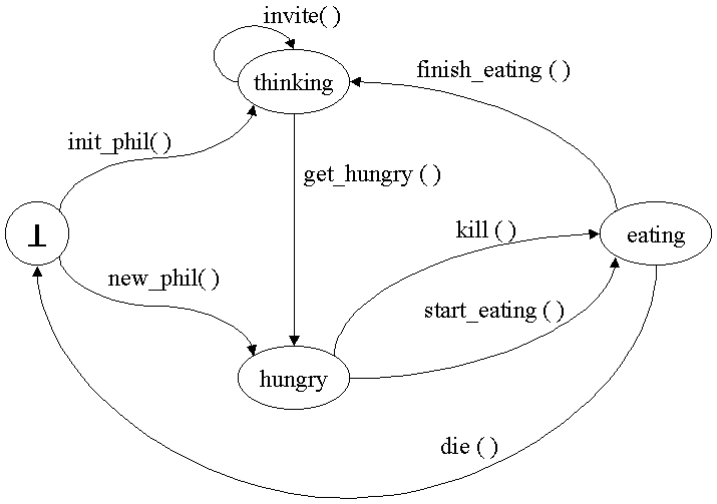


Figura 6.2: Diagrama de estados y transiciones del autómata *Philosopher*

plato de arroz y una serie de filósofos que pueden estar en tres estados diferentes (además del estado *dead*) pensando, hambrientos y comiendo. Entre cada par de filósofos hay un palillo (hay, por tanto, tantos palillos como filósofos) y cada filósofo, para poder comer, debe coger tanto el palillo situado a su izquierda como el palillo situado a su derecha. Si todos cogieran el palillo, por ejemplo, de su izquierda todos morirían de hambre pues ninguno podría comer y tampoco querría dejar su palillo. Así, la solución planteada consiste en que si un filósofo se encuentra hambriento y no puede coger el palillo que le falta, asesina a su dueño (con lo que el filósofo situado en ese lado y su palillo desaparecen del sistema) y de esa manera elimina el interbloqueo. Además, un filósofo puede invitar a otro a que se una a la comida.

El archivo de texto que conforma este modelo para que sea interpretable por el simulador es el siguiente (La declaración de variables es opcional):

```
Description
{
   This is a model of dynamic dining philosophers.
   Agents are: fork and Philosopher.
}

// Declare variables
Vars
{
   p,
   q,
   f1,
   f2
```

```
}

// Definition of the Philosopher automaton
Automaton Philosopher
{
   dead -> thinking : initPhil()
   dead -> hungry : newPhil()
   thinking -> thinking : invite()
   thinking -> hungry : becomeHungry()
   hungry -> eating : startEating()
   hungry -> eating : kill()
   eating -> thinking : finishEating()
   eating -> dead : die()
}

// Definition of the Fork automaton
Automaton Fork
{
   dead -> available : newFork()
   available -> notAvailable : take()
   notAvailable -> available : return()
   notAvailable -> dead : destroy()
}

Rules
{
    name:     Initialization
    synch:    p.initPhil() + f1.newFork()
    guard:    null
    update:   f1.id = new
              p.left = f1.id
              p.right = f1.id

    name:     creation
    synch:    p.invite() + q.newPhil() + f1.newFork()
    guard:    null
    update:   q.right = p.right
              p.right = f1.id
              q.left = f1.id
              f1.id = new

    name:     destruction
    synch:    p.kill() + q.die() + f1.destroy() + f2.take()
    guard:    q.right == f1.id
              p.left == f1.id
              p.right == f2.id
    update:   p.left = q.left
```

```
    name:     startEating
    synch:    p.startEating() + f1.take() + f2.take()
    guard:    p.right == f1.id
              p.left == f2.id
    update:   null

    name:     finishEating
    synch:    p.finishEating() + f1.return() + f2.return()
    guard:    p.right == f1.id
              p.left == f2.id
    update:   null

    name:     becomeHungry
    synch:    p.becomeHungry()
    guard:    null
    update:   null
}
```

Ahora vamos a ver cómo se interpreta esto en nuestro simulador.

En la Figura 6.3 tenemos el aspecto del simulador al cargar un modelo. La pantalla consta de cuatro ventanas principales y un pequeño panel de control en la parte superior para elegir el modo manual o el modo automático seleccionando el número de pasos. En la primera ventana se muestran los autómatas disponibles, (inicialmente sólo uno genérico llamado *Automaton Factory* que sirve como germen del resto si bien no pertenece realmente al sistema), en la segunda pantalla los vectores de sincronización de acciones presentes en el modelo y en la tercera los posibles conflictos a la hora de elegir entre qué autómatas pueden efectuar las reglas o qué acciones (en el caso de indeterminismo) se ven involucradas.

Como no tenemos seleccionado ningún autómata, la ventana de vectores de sincronización muestra todos los vectores. Si seleccionamos el autómata genérico *Autómaton Factory* sólo mostrará como vectores de sincronización de acciones posibles al de creación de objetos, como puede verse en la Figura 6.4.

Si ahora seleccionamos y ejecutamos esa acción, lo que tendremos será un nuevo autómata *Philosopher* y otro *Fork* como puede verse en la Figura 6.5.

Ahora sí que podemos invitar a otros dos filósofos a comer sin que se produzca error o conflicto en la elección, pues al invitar a un filósofo entrará directamente al estado *hungry* y por tanto no podrá realizar la accion *invite()* La secuencia de estas dos invitaciones puede verse en las Figuras 6.6 y 6.7.

Si, por el contrario, quisiéramos disparar la transacción cuyo vector de sincronización incluye la orden *start_eating()* hay dos posibles autómatas capaces de realizar esa acción, así que se producirá un conflicto y deberemos seleccionar uno de los dos en la ventana número 3. Un ejemplo de esto lo tenemos en la Figura 6.8.

Seleccionando, por ejemplo, el primero de los autómatas, éste pasaría al estado *eating* y los autómatas *fork* cuyo atributo coincida con los del autómata *philosopher* pasarían al estado *non_available*. Esto puede verse en la Figura 6.9.

En estas condiciones, podría pensarse que un *philosopher* puede tratar de asesinar a su competidor, pero esto en realidad no es posible, pues los intentos homicidas de

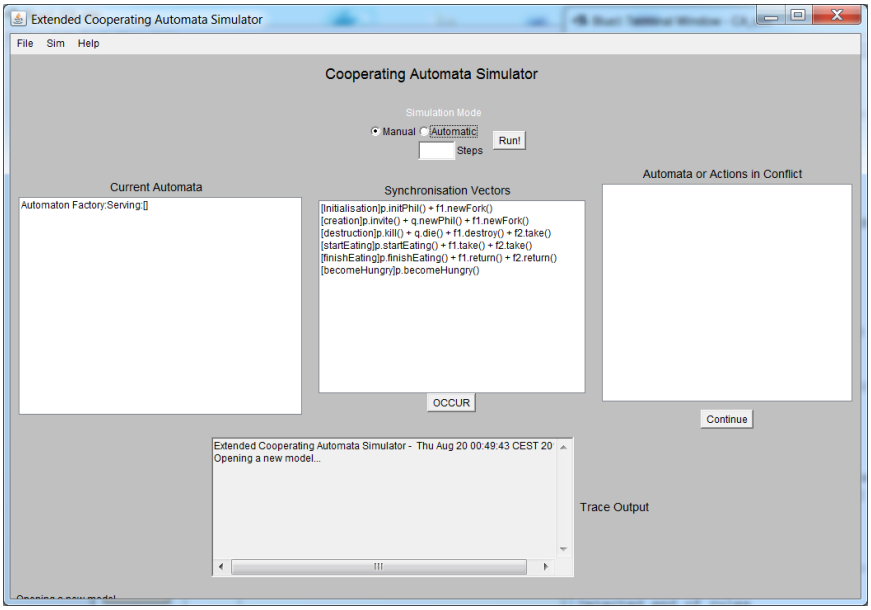


Figura 6.3: Entrada al simulador, tras haber cargado el modelo de los filosofos

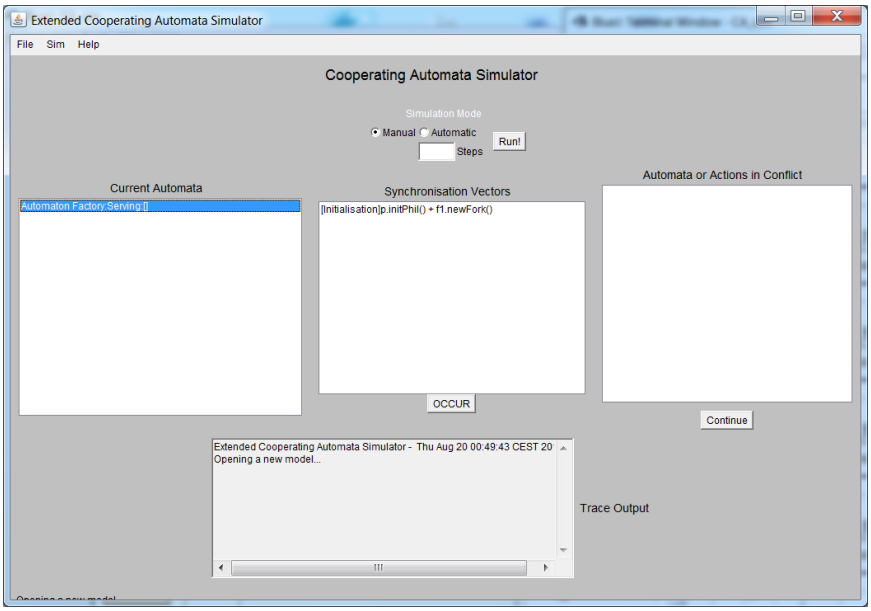


Figura 6.4: Vectores de sincronización disponibles al inicio

un filósofo en este modelo siempre van encaminados a su inmediato compañero a la derecha y no a su izquierda como es el caso. Por esto, si tratamos de efectuar el vector de sincronización que incluye la acción *kill()* el resultado es un mensaje de error en

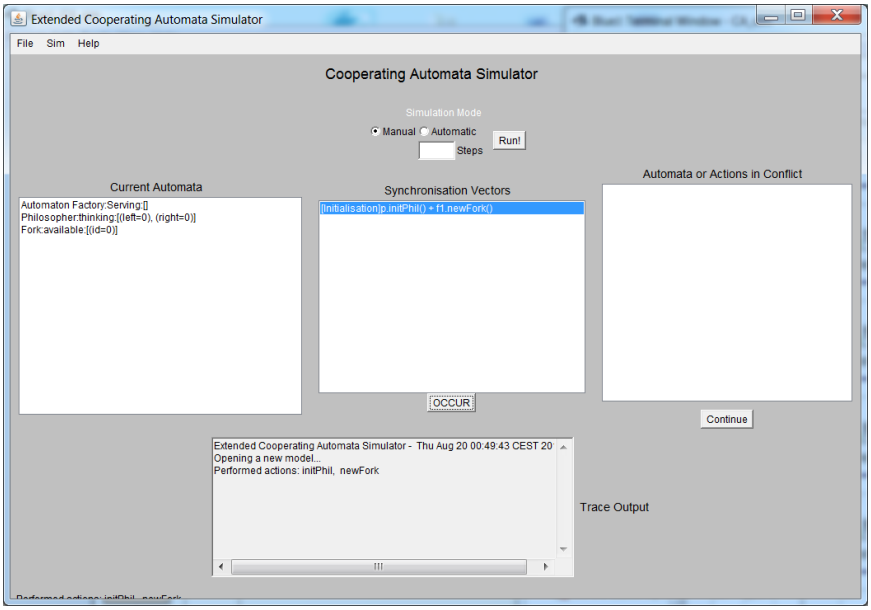


Figura 6.5: Creación del primer filósofo y su palillo

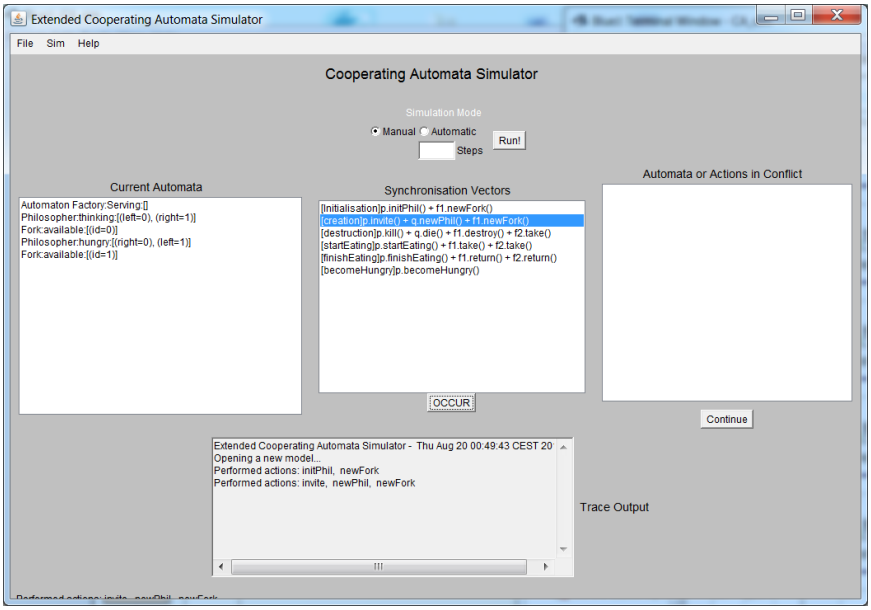


Figura 6.6: Invitación a un segundo filósofo directamente al estado hambriento

la ventana inferior como puede verse en la Figura 6.10.

Sin embargo, si tratamos de realizar la acción *finish_eating()* entonces no habrá conflicto y el estado de los autómatas *fork* pasará a *available* (véase Figura 6.11).

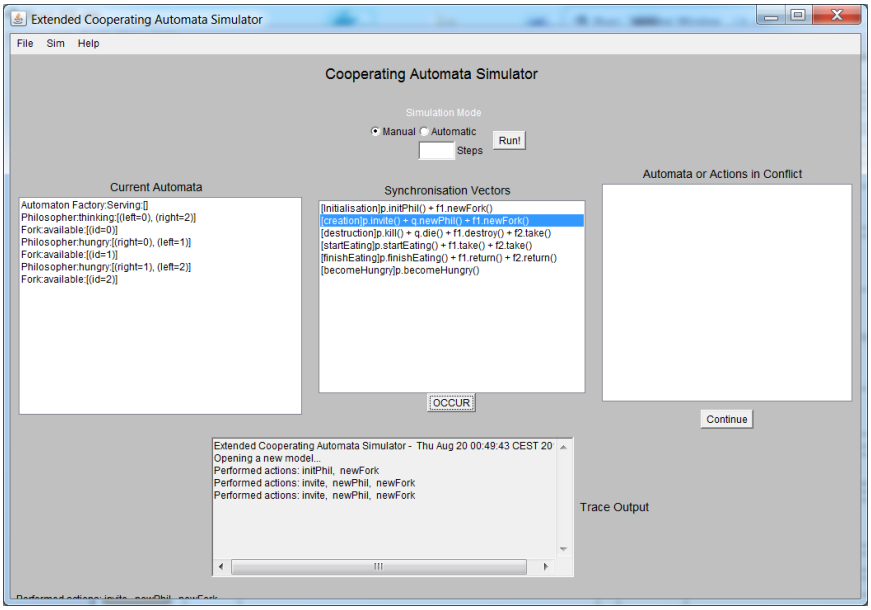


Figura 6.7: Sin conflictos en la elección de autómata, sólo uno puede sincronizar

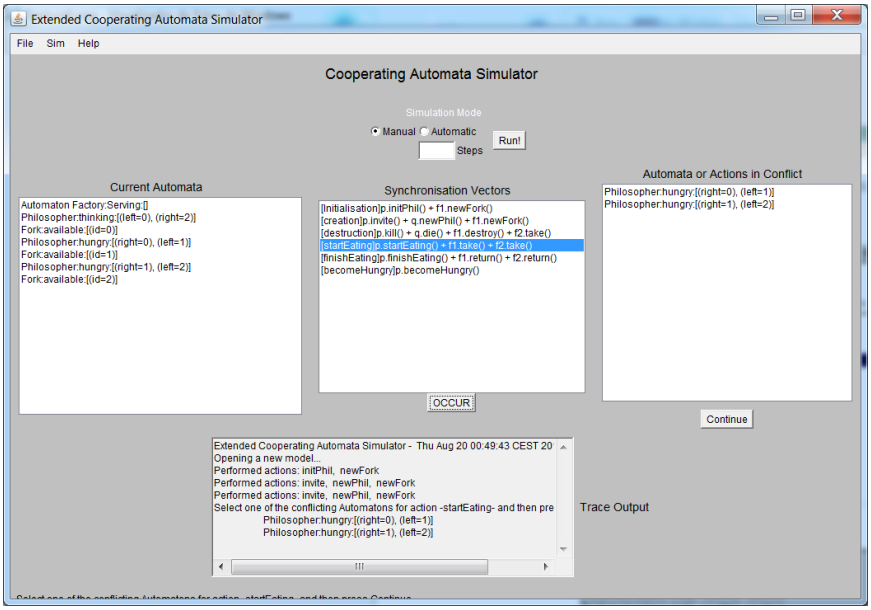


Figura 6.8: Conflicto: Dos autómatas coinciden en la misma componente de “synch”

Si, por el contrario, nos encontrásemos con que el que se hallase comiendo fuese el tercer autómata *philosopher* y el hambriento fuese el segundo (véase Figura 6.12), sí podría efectuarse la transacción *destruction()* y por tanto desaparecería el autómata

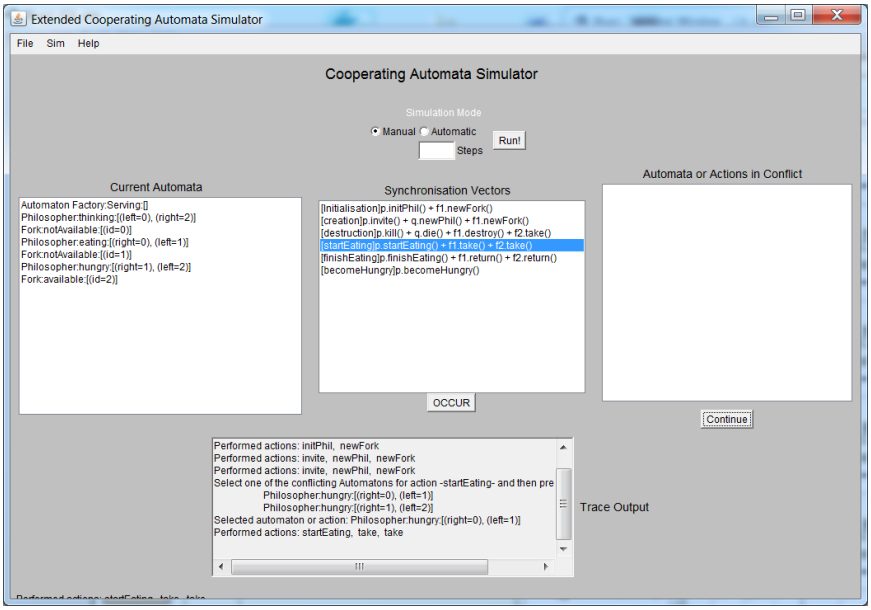


Figura 6.9: Uno de los autómatas pasa a *eating* y sus *fork* a *non_available*

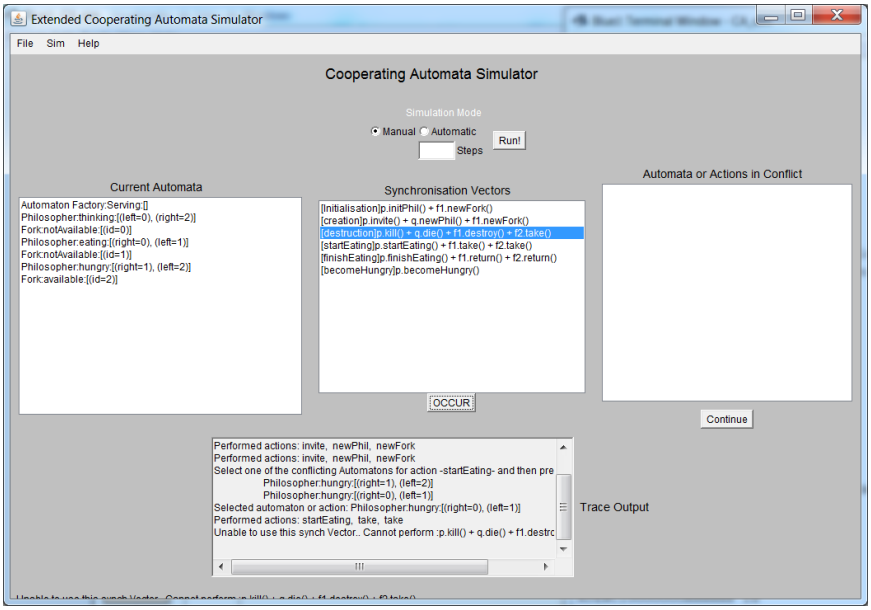


Figura 6.10: Intento fallido de disparar una regla

Philosopher asesinado y su palillo.(véase Figura 6.13)

Por último y como ejemplo de ejecución automática, es posible decirle al sistema que efectúe un número determinado de pasos automáticamente, resolviendo los

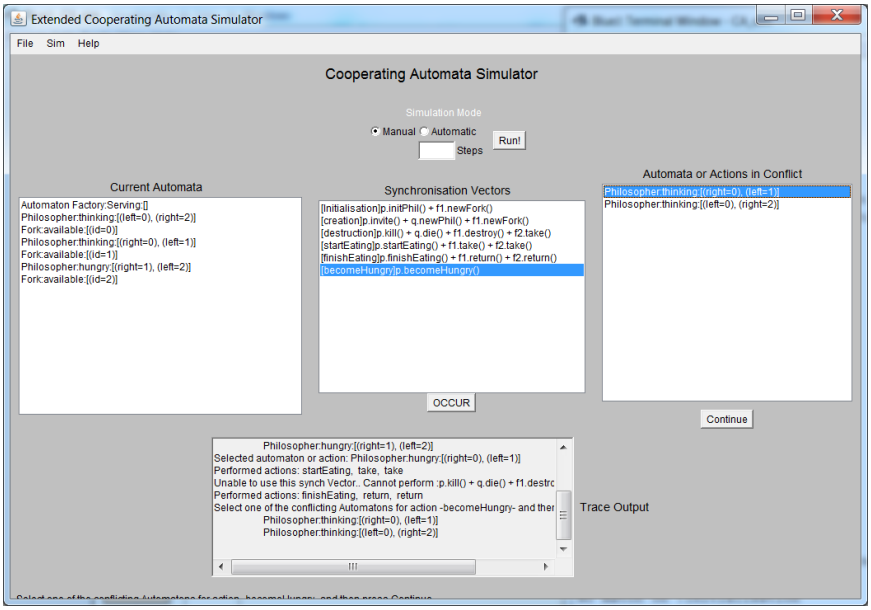


Figura 6.11: Ejecución de la transacción “finishEating”

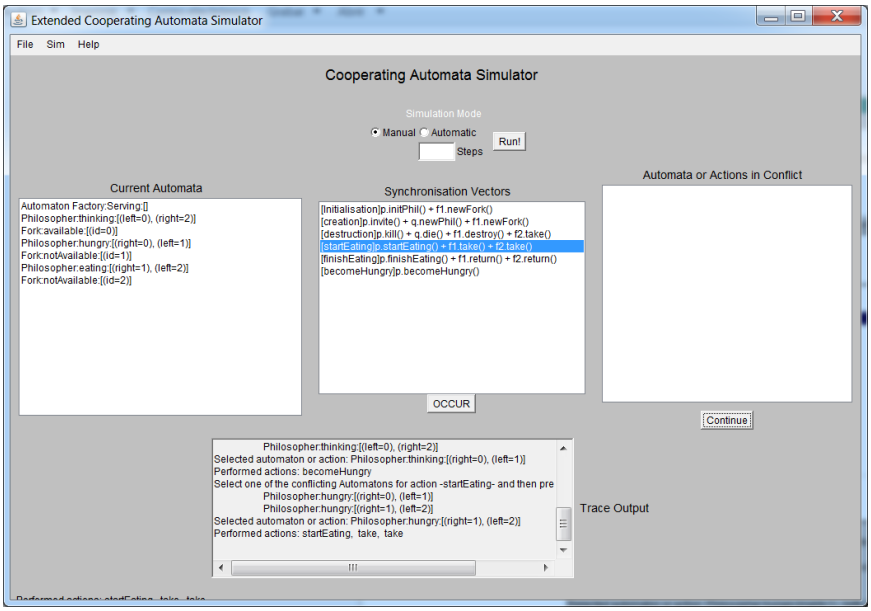


Figura 6.12: Un filósofo hambriento puede efectuar la acción “kill()”

conflictos de forma aleatoria. Las Figuras 6.14 y 6.15 muestran el estado del sistema antes de efectuar la simulación automática y después, en el que ha creado una serie de autómatas adicionales.

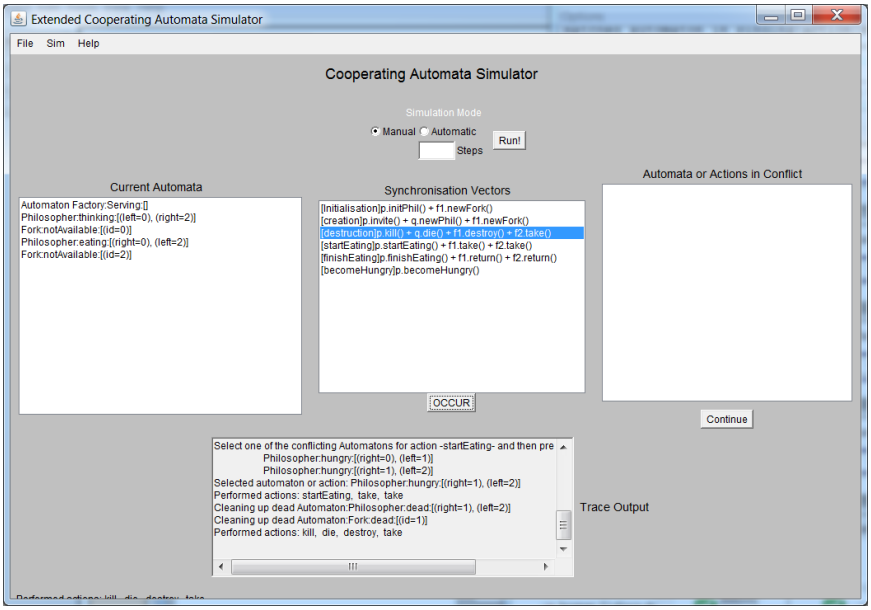


Figura 6.13: El filósofo asesinado desaparece y con él su palillo

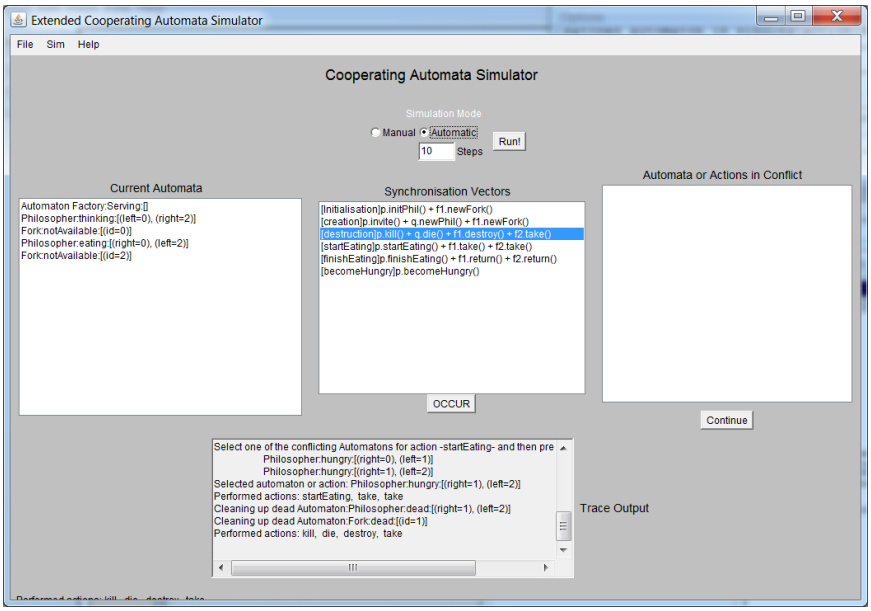


Figura 6.14: Antes de la ejecución de 10 pasos automáticamente

A continuación, para finalizar este capítulo vamos a presentar el ejemplo de la linea de montaje con los Autómatas Cooperativos Extendidos expuesto en el Capítulo 3.1.5 y el modelo analizado en el Capítulo 4.

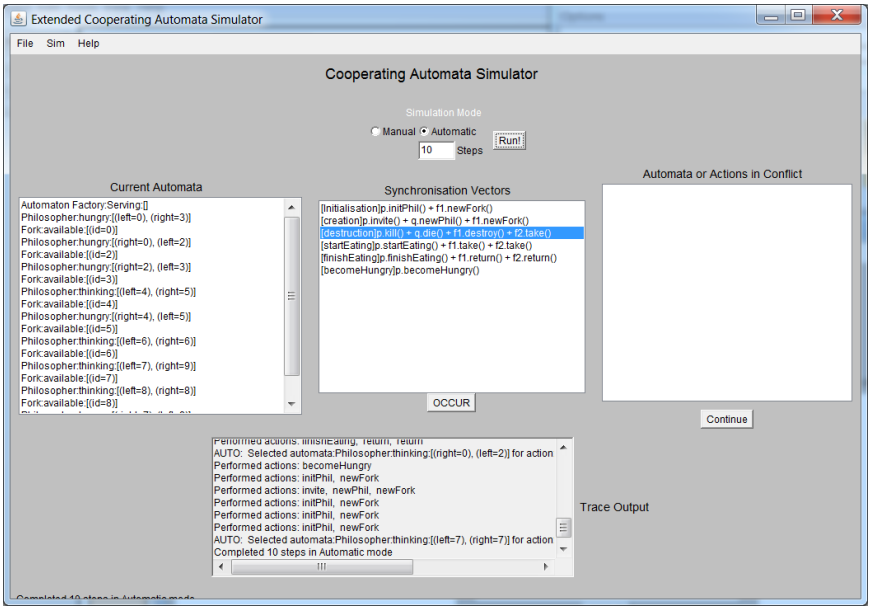


Figura 6.15: Tras la ejecución automática

Primero la versión para el simulador del sistema de la cadena de montaje y a continuación en la Figura 6.16, el sistema ya cargado en el simulador con los objetos creados a punto de iniciar el montaje del objeto complejo.

```
Description
{
   Este es un modelo de linea de montaje, con agentes
  objeto complejo, contenedor, rob y reponedor.
}

// Declare variables
Vars
{
   rob,
   rob1,
   rob2,
   rob3,
   rob4,
   con,
   con1,
   con2,
   con3,
   con4,
   oc,
   rep
```

```
}

// Definition of the Robot automaton
Automaton RobACE
{
   dead -> quieto : inicior()
   quieto -> preparado : cogerPieza()
   preparado -> quieto : ponerPieza()
}

// Definition of the Contenedor automaton
Automaton ConACE
{
   dead -> almacenando: inicioc()
   almacenando -> almacenando : recargar()
   almacenando -> almacenando : gastaPieza()
}
// Definition of the Reponedor Automaton
Automaton RepACE
{
   dead -> idle : inicio()
   idle -> idle : reponerPieza()
}
// Definition of the ObjectComplex Automaton
Automaton OcACE
{
   dead -> inicio : iniciar()
   inicio -> fase1 : ponerPieza1()
   fase1 -> fase2 : ponerPieza2()
   fase1 -> fase1 : movermas()
   fase1 -> fase1 : movermenos()
   fase2 -> fase3 : ponerPieza3()
   fase2 -> fase3 : ponerPieza4()
   fase2 -> fase2 : movermas()
   fase2 -> fase2 : movermenos()
   fase3 -> fase4 : ponerPieza3()
   fase3 -> fase4 : ponerPieza4()
   fase3 -> fase3 : movermas()
   fase3 -> fase3 : movermenos()
   fase4 -> fase5 : ponerPieza1()
   fase4 -> fase4 : movermas()
   fase4 -> fase4 : movermenos()
   fase5 -> dead : fin()
}

Rules
{
```

```
name:    New_ObjetoC
synch:   oc.iniciar()
guard:   null
update:  oc.#poso = 1
         oc.#pieza1 = 0
         oc.#pieza2 = 0
         oc.#pieza3 = 0
         oc.#pieza4 = 0

name:    New_Reponedor
synch:   rep.inicio()
guard:   null
update:  null

name:    NewRob_Con_pos1
synch:   rob1.inicior() + con1.inicioc()
guard:   null
update:  rob1.#posr = 1
         con1.#posc = 1
         con1.#cantidad = 0

name:    NewRob_Con_pos2
synch:   rob2.inicior() + con2.inicioc()
guard:   null
update:  rob2.#posr = 2
         con2.#posc = 2
         con2.#cantidad = 0

name:    NewRob_Con_pos3
synch:   rob3.inicior() + con3.inicioc()
guard:   null
update:  rob3.#posr = 3
         con3.#posc = 3
         con3.#cantidad = 0

name:    NewRob_Con_pos4
synch:   rob4.inicior() + con4.inicioc()
guard:   null
update:  rob4.#posr = 4
         con4.#posc = 4
         con4.#cantidad = 0

name:    MovimentoAdelante
synch:   oc.movermas()
guard:   oc.#poso < 4
update:  oc.#poso += 1
```

```
name:    MovimientoAtras
synch:   oc.movermenos()
guard:   oc.#poso > 1
update:  oc.#poso -= 1

name:    Acoplar_pieza_1
synch:   oc.ponerPieza1() + rob.ponerPieza()
guard:   oc.#poso == rob.#posr
         oc.#poso == 1
         oc.#pieza1 == oc.#pieza3
         oc.#pieza3 == oc.#pieza4
update:  oc.#pieza1 += 1

name:    Acoplar_pieza_2
synch:   oc.ponerPieza2() + rob.ponerPieza()
guard:   oc.#poso == rob.#posr
         oc.#poso == 2
         oc.#pieza2 == 0
         oc.#pieza1 == 1
update:  oc.#pieza2 += 1

name:    Acoplar_pieza_3
synch:   oc.ponerPieza3() + rob.ponerPieza()
guard:   oc.#poso == rob.#posr
         oc.#poso == 3
         oc.#pieza3 == 0
         oc.#pieza2 == 1
update:  oc.#pieza3 += 1

name:    Acoplar_pieza_4
synch:   oc.ponerPieza4() + rob.ponerPieza()
guard:   oc.#poso == rob.#posr
         oc.#poso == 4
         oc.#pieza4 == 0
         oc.#pieza2 == 1
update:  oc.#pieza4 += 1

name:    Finalizar
synch:   oc.fin()
guard:   null
update:  null

name:    Preparar
synch:   con.gastaPieza() + rob.cogerPieza()
guard:   con.#posc == rob.#posr
update:  con.#cantidad -= 1
```

```
    name:     Reponer
    synch:    con.recargar() + rep.reponerPieza()
    guard:    con.#cantidad == 0
    update:   con.#cantidad = 4
}
```

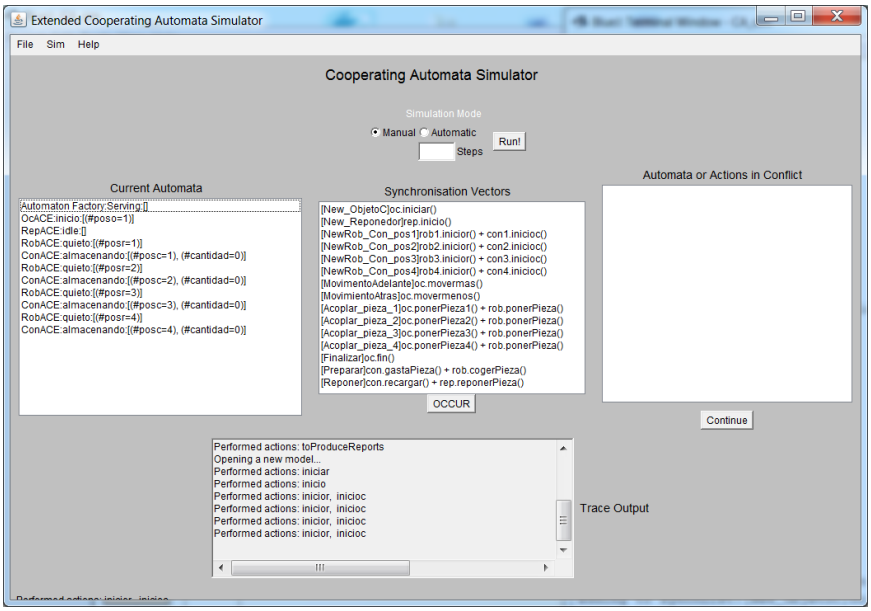


Figura 6.16: Inicio en simulador del sistema de la cadena de montaje

Para finalizar, a continuación mostramos el sistema construido y analizado en el Capítulo 4.3 donde ya se han identificado los tres procesos independientes y se han construido dos parejas de objetos *Creative* y *MarketingExpert* para realizar campañas de marketing y se resuelve el conflicto de su elección al disparar la regla de transacción correspondiente como puede verse en la Figura 6.17 y también justo después de invocar al proceso independiente de la regla *Security_Tasks* con el incremento del atributo numérico del autómata *SecurityOfficer* en la Figura 6.18.

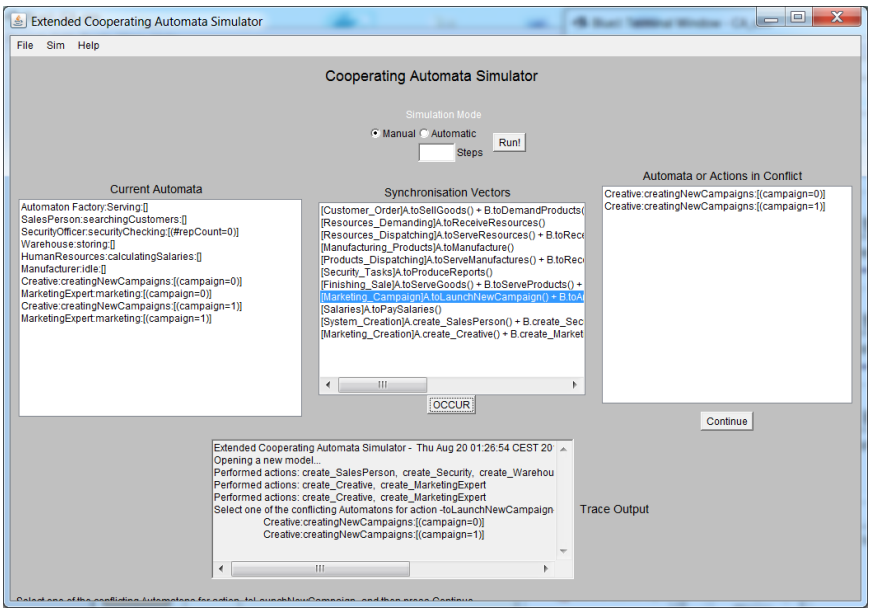


Figura 6.17: Modelo de la Compañía de la Sección 4.3

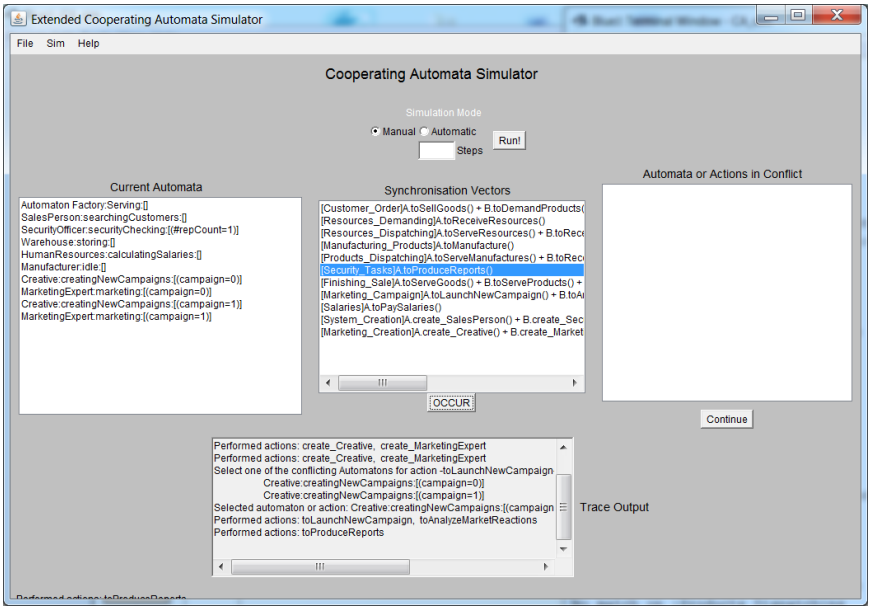


Figura 6.18: Tras la invocación de *Security_Tasks*

Capítulo 7

Conclusiones

En la presente Tesis Doctoral se han presentado una serie de formalismos basados en la Teoría de Autómatas y en las Redes de Petri destinados a la representación de sistemas distribuidos. Dentro de las propuestas que están basadas en autómatas cabe destacar algunas que han sido mostradas más en profundidad como la de los autómatas Team que presentan un entorno multiagente para la modelización de sistemas *groupware* y los Autómatas Finitos Paralelos que dan una notación específica para expresar la concurrencia y el paralelismo en sistemas distribuidos; en cuanto a los modelos basados en redes de Petri, presentamos las Redes de Referencias y su simulador, la herramienta Renew. Tanto unos como otros han sido comparados, imitados o transformados con los Autómatas Cooperativos Extendidos. Este formalismo permite mostrar de una forma gráfica y sencilla la problemática de representación de sistemas de este tipo aunando características de la Teoría de Autómatas y de las redes de Petri haciendo explícito el mecanismo de concurrencia en la forma de reglas de transacción.

Como resumen del funcionamiento del modelo, un sistema de Autómatas Cooperativos Extendidos consta de dos partes fundamentales. En primer lugar, de los autómatas en sí que, a su vez, además de estados y acciones tienen una porción de memoria privada y, en segundo lugar, unas reglas de transacción con vectores de sincronización de acciones, guardas y actualizaciones.

La memoria privada de los autómatas contiene hasta dos tipos de atributos. Estos atributos son: los atributos de tarea o identificadores y los atributos numéricos. Como su propio nombre indica, los atributos de tarea se utilizan para sincronizar autómatas en tareas y subtareas, mientras tanto, los atributos numéricos difieren de los anteriores en que su rango de valores es finito y corresponde a un subconjunto de los naturales representando una extensión del conjunto de estados del autómata.

El modelo original presenta tres niveles de expresividad:

- modelo básico (sin atributos de tarea) corresponde a la coordinación pura entre agentes. Consiste básicamente en relaciones cliente servidor, pero sin una ligadura fija entre autómatas. En este nivel las únicas limitaciones para la ejecución de las reglas de transacción corresponden a la disponibilidad de los autómatas para efectuar acciones en su actual estado, entendiendo en este contexto que el estado es tanto el estado explícito del autómata como el valor de sus atributos

numéricos. Sí es posible coordinar agentes a través de estos atributos pero tan solo porque el estado en el que se encuentran (el valor de los mismos) cumple una cierta restricción como puede ser la igualdad.

- El modelo extendido (con un atributo de tarea como máximo por agente) es adecuado para representar aplicaciones *workflow*, donde los agentes pertenecen a determinadas tareas (existe ligadura entre ellos) y estas tareas (y subtareas) pueden competir entre ellas y funcionar en paralelo, no obstante, no hay colaboración entre ellas al pertenecer como mucho cada agente sólo a una. En este nivel la propiedad de acotamiento es decidible.

- El modelo completo (con dos o más atributos de tarea) permite representar aplicaciones *groupware*, con tareas, subtareas, cooperación, concurrencia, paralelismo y competencia entre tareas. Este modelo tiene la misma potencia expresiva que la Máquina de Turing y, por tanto, no podemos comprobar la decibilidad de propiedades como el acotamiento. Nótese que de nuevo los atributos numéricos no son contemplados en el cómputo de atributos; esto es así porque los atributos de tarea son simbólicos (y virtualmente, de rango infinito) y, por tanto, representan relaciones fuera del estricto ámbito del autómata. Sin embargo los numéricos, al ser un conjunto finito de naturales, se limitan a impedir la explosión de estados.

El modelo original es muy similar exceptuando todo cuanto hace referencia a los atributos numéricos. La inclusión de los mismos así como de los mecanismos de consulta y actualización en las guardas y actualizaciones de las reglas de transacción son una de las aportaciones principales de esta tesis doctoral. La extensión presentada, si bien no añade grandes cambios en cuanto a la definición, sí que supone un cambio en el concepto del sistema sin modificar la base operacional del mismo, trasladando parte de la información de los estados y acciones de los autómatas a los atributos numéricos y a las restricciones y actualizaciones de las reglas. Con esto, se faculta al modelo de una forma mucho más precisa, sencilla y natural de representar sistemas cuya solución sólo podría darse con una explosión de estados en los autómatas así como de una explosión de reglas de transacción para incorporar dichas acciones y cambios de estado. Pese a la capacidad del modelo original para representar esos sistemas, en ocasiones los autómatas resultantes y el número de reglas acabansiendo demasiado complicados para problemas en apariencia bastante sencillos. Esto ocurre sobre todo cuando el estado del autómata (es decir, el nodo del cual parten las acciones en el diagrama) depende de cierto valor numérico, por ejemplo la posición geográfica. En este caso, es preciso distinguir el estado del autómata en todas y cada una de las posiciones que el autómata sea capaz de ocupar puesto que las acciones que pueda realizar estarán, probablemente, asociadas también a dicha posición.

En la extensión presentada existe un segundo tipo de atributos en los autómatas. La diferencia es que en este caso los atributos son numéricos y la explosión de reglas que se mencionaba anteriormente se reduce a añadir una simple guarda en una única regla. Esto no quita para que ambos formalismos sean del todo equivalentes. De hecho, se provee de un algoritmo de conversión de Autómatas Cooperativos Extendidos en Autómatas Cooperativos que garantiza el mantenimiento de todas las propiedades de aquellos y, dado que, cualquier sistema de Autómatas Cooperativos es, por definición,

también un sistema de Autómatas Cooperativos Extendidos, podemos concluir que ambos modelos son funcionalmente iguales. Por tanto, si bien los tres niveles de expresidad se mantienen al añadir los atributos numéricos existen sistemas cuya resolución sería antinatural pero posible en el modelo básico, pero que obtienen una solución mucho más elegante y precisa con el sistema extendido.

En otras palabras, se han propuesto los Autómatas Cooperativos Extendidos como una extensión de los Autómatas Cooperativos destinada a simplificar la notación en estos casos de dependencia de un valor numérico del diagrama de estados. La intención es lograr mantener la expresividad del modelo original y sus mismas propiedades, mejorando la legibilidad del mismo. Esta simplificación es particularmente clara en problemas como el representado en la Figura 3.4 del Capítulo 3 donde la posición física o cualquier otra magnitud numérica (como la representación de instantes del tiempo) es modelizada con un atributo numérico, dejando el autómata resultante así como las reglas de transacción mucho más claros y concisos.

La extensión evita complicadas y farragosas descripciones de autómatas para evitar el uso de atributos de tarea. Por ejemplo, si ciertos tipos de autómatas, pero no necesariamente siempre las mismas instancias de ellos, deben evolucionar juntos en determinadas fases del diseño la única manera de modelizarlo en el modelo AC es a través de la utilización de un atributo de tarea (usado para un fin distinto para el que fue ideado inicialmente) o añadir estados, reglas y acciones adicionales en cada autómata para permitir esa sincronizacion sin necesidad de usar un atributo de tarea. Sin embargo en los ACE, basta con usar un atributo numérico para aclarar que los autómatas están en la misma fase. En esta aproximación, cualquier autómata cuyo atributo numérico esté en el valor adecuado puede participar en la ejecución de una regla y no implica que sea el único que puede hacerlo ni hay una ligazón completa entre los autómatas involucrados. Y todo esto con la única condición que el número de valores posibles de esos atributos sea finito. El sistema de fabricación de la Figura 3.25 del Capítulo 3 es un claro ejemplo ya que el modelo resultante con atributos numéricos es muy sencillo, mientras que modelizarlo con un sistema básico sin atributos de tarea si bien es posible resulta si no ilegible sí muy complicado de entender y poco natural.

Ambos modelos son suficientemente expresivos para modelar la mayoría de sistemas. En concreto, existe un modo fácil de simular incluso el comportamiento de otros modelos complejos como los autómatas Team de manera sencilla y clara.

Una vez establecido el funcionamiento del modelo original y del modelo extendido y presentados algunos ejemplos de modelización de diferentes sistemas de uno y otro tipo se plantea el análisis a priori de las soluciones dadas a los diferentes sistemas como un medio de simplificación y de identificación de problemas, procesos y subprocesos en las etapas más iniciales del diseño. Para ello, y con la idea de automatizar el análisis para hacerlo útil en el proceso de diseño, se plantean tres tipos de relaciones diferentes:

- Relaciones de *vinculación* entre reglas y autómatas a través de acciones que representan las correlaciones entre los distintos vectores de sincronización y las acciones que definen a los propios autómatas.

- Relaciones de *concurrencia* entre autómatas en reglas que representan las sincronizaciones o cooperaciones entre autómatas en el disparo de una regla de

transacción al estar vinculados ambos autómatas a la misma regla pero por acciones diferentes del vector de sincronización (o componentes diferentes del mismo), con lo que implican que en el disparo de la regla de transacción ambos pueden participar y evolucionar simultáneamente.

- Relaciones de *competencia* entre reglas por un mismo autómata que representan la necesidad de elección entre reglas de transacción por la imposibilidad de un disparo simultáneo al compartir un mismo autómata vinculado a ambas reglas. En este caso, el disparo de una regla de transacción puede implicar el cambio de estado de un autómata que, por tanto, impida de hecho el disparo de aquella con la que competía la regla disparada.

De cada relación se plantean cuatro niveles de certidumbre, desde las relaciones muy débiles que sólo plantean la posibilidad de existencia de dichas relaciones hasta las muy fuertes que indican la absoluta certeza de la existencia de dicha relación, pasando por los estados intermedios de débil y fuerte, con crecientes niveles de seguridad.

De todas estas relaciones, las más débiles son aquellas en las que se centra el estudio del Capítulo 4 dado que son las que pueden realizarse a priori sin necesidad de simular el sistema. Por tanto, son las que permiten encontrar subprocesos, cuellos de botella y reglas independientes o críticas antes incluso de finalizar la modelización del sistema, permitiendo su simplificación y solución de problemas en el mismo proceso de modelización. El resto de niveles de certidumbre requieren una simulación y, por tanto, sólo reflejan el comportamiento del sistema en un instante dado de la simulación o ejecución, algo así como identificar los subprocesos, cuellos de botella, etc. durante una "foto fija" del sistema.

Del mismo modo, se plantea un algoritmo que permite, utilizando el cálculo de las relaciones muy débiles, identificar de forma automática los subprocesos existentes del sistema y, por tanto, las reglas de transacción independientes entre sí. Estos cálculos permiten, por un lado, la simplificación del sistema en subsistemas independientes y, por otro lado, la identificación de ejecuciones que lleven al mismo resultado a pesar de incluir secuencias diferentes de disparo de reglas.

También se ha desarrollado una aplicación que permite la modelización, edición y validación de sistemas bajo el modelo de los Autómatas Cooperativos Extendidos, realizando una doble tarea, primero, asegurar la corrección sintáctica del modelo construido y, segundo, en tiempo de diseño realizar el análisis anteriormente descrito de manera automática. Esto es posible al haberse integrado en la propia herramienta de edición la automatización del análisis. Se realiza el estudio sobre el propio modelo de las diferentes relaciones existenes y se obtiene un resumen de resultados mediante la aplicación del algoritmo y la interpretación del mismo. De este modo el diseñador puede, durante el proceso de diseño y durante la edición del modelo, realizar el análisis a priori del modelo para comprobar si hay autómatas o reglas que intervengan en la mayoría de transacciones o, por el contrario, la existencia de subprocesos independientes y utilizar estos resultados como guía para completar el diseño de forma que el sistema sea más eficiente y presente menos problemas más adelante.

Por último se ha presentado también una herramienta destinada a la simulación de sistemas implementados bajo el modelo de los Autómatas Cooperativos y también bajo la extensión propuesta que permiten, de manera sencilla, manual o automática,

comprobar el funcionamiento del sistema recién diseñado y probar la respuesta del mismo ante cada una de las reglas de disparo.

Dado que el modelo alcanza la expresividad de la Máquina de Turing con dos atributos de tarea no es posible la demostración automática de determinadas propiedades como el acotamiento pero, sin embargo, en los modelos sin o con un único atributo de tarea sí se tiene un árbol de alcanzabilidad de estados finito y sí es posible realizar determinados estudios sobre ellos. Es factible, además, (dada la estrecha relación de los ACE con las redes de Petri) obtener el árbol de cobertura de red de Petri equivalente o transformar el sistema en un sistema de Redes de Referencias aprovechando con ello las herramientas ya existentes de cálculo y validación de las mismas.

Como posibles trabajos futuros se plantean varias líneas de desarrollo.

- En primer lugar y dado que el modelo ya dispone de una herramienta de edición y análisis y otra de simulación el siguiente paso lógico es el de añadir la funcionalidad de prototipado. Si bien, al tratarse de un modelo a muy alto nivel dicho prototipado debería ser inicialmente sólo a nivel de definición de clases y agentes. Una posibilidad es aprovechar la infraestructura de alguna de las plataformas de modelado descritas en el capítulo introductorio para realizar una transformación del modelo bajo la perspectiva de los Autómatas Cooperativos Extendidos hacia alguno de los estándares que ya disponen de dichas herramientas de prototipado como *GAMA* [52] o *Agent Factory* [126] entre otras.

- Otra línea de investigación futura consiste en establecer un algoritmo eficiente de análisis de un sistema en tiempo de simulación para obtener resultados similares con el resto de grados de certidumbre detectando las situaciones de interbloqueo antes de que se produjeran y estableciendo mecanismos de control en los casos en los que pudieran producirse.

- Por último, llevando al máximo el concepto de simplificación del sistema, una tercera línea de investigación consiste en la búsqueda de un mecanismo para obtener la forma canónica de un sistema de manera (en lo posible) automática para permitir definir conceptos como la bisimulación y obtener esquemas básicos de funcionamiento reutilizables en posteriores modelos.

Bibliografía

[1] Van der Aalst, W.M.P., *Verification of WorkFlow Nets.* En Proc. of ICATPN'97, vol. 1248 de Lecture Notes in Computer Science, Springer Verlag, pags. 407-426, (1997).

[2] ACRONYMICS INC., *AGENTBUILDER: An Integrated Toolkit for Constructing Intelligent. Software Agents. Reference Manual.* http://www.agentbuilder.com (2004).

[3] Alonso, D., Sánchez-Ledesma, F. , Sánchez, P. Pastor, J.A., Álvarez, B., *Models and Frameworks: A Synergistic Association for Developing Component-Based Applications*, The Scientific World Journal, (2014).

[4] Arnold, A., Nivat, M., *Comportements de processus.* En Colloque AFCET "Les mathématiques de l'Informatique", pags. 35-68, (1982).

[5] Badouel, E., Darondeau, P., QUichaud, D., Tokmakoff, A., *Modelling Dynamic Agent Systems with Cooperating Automata.* International Conference on Parallel and distributed Processing Techniques and Applications (PDPTA'99), Las Vegas, Nevada, (1999).

[6] Badouel, E., Oliver, J., *Reconfigurable nets, a class of high level Petri nets supporting dynamic changes within workflow systems.* En Proc. of Workflow Managemente: Net-based concepts, models, techniques and Tools (W.Van der Aalst Ed.) Lisboa, (1998).

[7] Badouel, E., Oliver, J., *Dynamic Changes in Concurrent Systems: Modelling and Verification.* INRIA Research Report RR-3708. (1999).

[8] Banâtre, J.P., Le Métayer, D., *A new computational model and Its Discipline of programming.*, Technical Report, INRIA, 566 France, (1986).

[9] Banâtre, J.P., Le Métayer, D., *Gamma and the chemical reaction model: ten years after*, En Coordination programming: mechanisms, models and semantics, World Scientific Publishing, IC Press, (1996).

[10] Banâtre, J.P., Le Métayer, D., *Gamma and the chemical reaction model: Fifteen years after.* En Proc. of Multiset Processing, Mathematical, Computer Science, and Molecular Computing Points of View, Lecture Notes in Computer Science, vol. 2235, (2001).

[11] Battiston, E., De Cindio, F., Mauri, G., *Nets: a class of high-level nets having objects as domains.* En G. Rozengerg (Ed.) Advances in Petri Nets 88 vol. 240 of Lecture Notes in Computer Science, Springer-Verlag (1988).

[12] ter Beek, M.H., Ellis, C.A., Kleijn, J., Rozenberg G., *Team automata for CSCW.* En Proc. of 2nd Int. Coll. on Petri Net Technologies for Modelling Communication Based Systems, pags, 1-20, (2001).

[13] ter Beek, M.H., Ellis, C.A., Kleijn, J., Rozenberg G., *Team automata for Spatial Access Control.* En Proc. of ECSW2001, European Conference on Computer Supported Cooperative Work, (2001).

[14] Bellifemine, F., Caire, G., Poggi, A., Rimassa, G., *JADE: A white Paper.* En EXP in search of innovation vol. 3, pags. 6-19. (2003).

[15] van Benthem J., *Modal Correspondence Theory.* PhD thesis, Mathematisch Instituut and Instituut voor Grondslagenonderzoek, University of Amsterdam, (1976).

[16] Berry, G., Boudol, G., *The Chemical Abstract Machine.* En Proc. of the Principles of Programming Languages, pags. 81-93, ACM Press, (1993).

[17] Bordini, R., Hübner, J., Wooldridge, M., *Programming Multi-Agent Systems in AgentSpeak using Jason.* Wiley Series in Agent Technology. ISBN: 978-0-470-02900-8 (2007).

[18] Borshchev, A., *The Big Book of Simulation Modeling: Multimethod Modeling with Anylogic 6.* AnyLogic North America. ISBN: 978-0-9895731-7-7. (2013).

[19] Bray, T. Paoli, P., *Extensible Markup Language (XML) 1.0 (Fifth Edition).* W3C Recommendation 26 November 2008. C. M. Sperberg-McQueen, Eve Maler, François Yergeau eds. (2008).

[20] Braubach, L., Pokahr, A., *The Jadex Project: Simulation.* Multiagent Systems and Applications, vol. 45, pags. 107-128. (2013).

[21] Braun, P., Rossak, W., *Mobile Agents: Basic Concepts, Mobility Models, and the Tracy Toolkit.* Morgan Kaufmann Publishers Inc. (2004).

[22] Brooks R. A., *Intelligence Without Reason.* En Proc. of the 1991 International Joint Conference on Artificial Intelligence, pags. 569-595, (1991).

[23] Bryl, V., Mello, P., Montali, M., Torroni, P., Zannone, N., *B-Tropos: Agent-Oriented Requirements Engineering Meets Computational logic for Declarative Business Process Modeling and Verification.* En Proc8th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA VIII), LNCS, vol. 5056, pags. 157-176, Springer. (2008).

[24] Bullock, A., Benford, S., *Access Control in Virtual Environments.* En Proc. of VRST'p7, ACM symposium on Virtual Reality Software and Technology, pags. 29-35. (1997).

[25] Bullock, A., *Spatial Access Control in Collaborative Virtual Environments.* Tesis Doctoral, University of Nothingham. (1998).

[26] Bullock, A. and Benford, S., *An access control framework for multi-user collaborative environments.* of the GROUP'99 International ACM SIGGRUP Conference on Supporting Group Work, pags. 140-149. (1999).

[27] Castells, P., *La Web Semántica* C. Bravo, M.A. Redondo (Eds.), Sistemas Interactivos y Colaborativos en la Web, Ediciones de la Universidad de Castilla-La Mancha, pags. 195-212. (2003).

[28] Chandy, K.M., Misra, J., *Parallel Program Design.* Addison-Wesley Publishing, Reading, Mass, (1988).

[29] Chiang, S. Kuo, C., Meerkov, S.M., *Bottlenecks in Serial production Lines: Identification and Application.* Mathematical Problems in Engineering, (2002).

[30] Christensen, S., Damgaard Hansen, N., *Coloured Petri Nets Extended with Channels for Synchronous communication.* Technical Report DAIMI PB-390, Computer Science Department, Aarhus University, (1992).

[31] ACM, *conference on Computer-Supported Cooperative Work and Social Computing* (1985-2015).

[32] CybelePro: Distributed Intelligent Systems, Intelligent Automation Inc., *CybelePro Agent Infrastructure, User's Guide v3.0.* http://www.CybelePro.com.

[33] De Michelis, G., Ellis, C.A., *Computer supported Cooperative Work and Petri Nets.* Third Advenced Course on Petri Nets, Dagstuhl Castle, Germany. Lectures on Petri Nets II: Applications (W.Reisig and G.Rozenberg Eds.), vol. 1492 of Lecture Notes in Computer Science, Springer Verlag, pags. 125-153, (1998).

[34] Deransart, P., Cervoni, L., Ed-Dbali, A., *Prolog: The Standard: Reference Manual*, Springer-Verlag, (1996).

[35] Dewan, P., Shen, H., *Flexible Meta Access-Control for Collaborative Applications.* En Proc. of the CSCW'98 ACM Conference on Computer Supported Cooperative Work, Seattle, Washington, ACM Press, pags. 247-256, (1998).

[36] Dijkstra, E.W., *A discipline of programming.* Prentice hall, Englewood Cliff, J.J., (1976).

[37] Eftekhari, M. Hossein, Barzegar, Zeynab, Isaai, M. T., *Web 1.0 to Web 3.0 Evolution: Reviewing the Impacts on Tourism Development and Opportunities.* En Proc. of the First International Conference on Human-computer Interaction, Tourism and Cultural Heritage, pags. 184-193, Springer-Verlag. (2011).

[38] Ellis, C.A., Nutt, G.J., *Modeling Enactment of Workflow Systems.* En Proc. of ICATPN'93, vol. 691 of Lecture Notes in Computer Science, Springer Verlag pags. 1-16, (1993).

[39] Ellis, C.A., *Team Automata for Groupware Systems.* En Proc. of GROUP'97 International ACM SIGGROUP Conference on Supporting Group Work: The Integration Challenge, pags. 415-424,. J. Clifford, B. Lindsday, and D. Maier eds. ACM Press, (1997).

[40] Engelfriet, J., Leih, G., Rozenberg, G., *Net base description of parallel object-based systems, or POTs ad POPs.* En J.W. de Bakker, W.P. de Roever y G. Rozenberg (eds.) Rex. School/Workshop Foundations of Object-Oriented Languages, Noordwijkerhout, volume 489 of Lecture Notes in Computer Science, Springer Verlag, pags. 229-273, (1991).

[41] Fagernes, S., Couch, A.L., *Coordination and information exchange among resource management agents.* En Proc. of Integrated Network Management, (Agoulmine, N, Bartolini, C., Pfeifer, T., O'Sullivan, D. Eds.), pags. 422-429. IEEE. (2011).

[42] Fasli, Maria and Virginas, Botond, *BDI Agents: Flexibility, Personalization, and Adaptation for Web-Based Support Systems.* Cap. 9 de *Intelligent Agents in the Evolution of Web and Applications*, pags. 191-221. Springer Berlin Heidelberg. (2009).

[43] Findlow, G., Billington, J., *High-level nets for dynamic dining philosophers systems.* En Semantics for Concurrency, Leicester 1990 (M.Z. Kwiatkowska, M.W. Shields, and R.M.Thomas Eds.) Workshop in Computing, Berlin, pags. 185-203. Springer-Verlag. (1990).

[44] Finin, T., Fritzson, R., McKay, D., McEntire, R., *KQML as an agent communication language.* En Proc. of the third international conference on Information and knowledge management (CIKM). (1994).

[45] Friendly, M., *Advanced Logo: A Language for Learning.* Computer Science for the Behavioral Sciences Series. (1988).

[46] Fuentes R., Gómez, J.J., Pavón, J., *Verification and Validation Techniques for Multi-Agent Systems.* Upgrade, vol. 5, pags. 15-19. Council of European Informatics Societies. (2004).

[47] Frutos, D., Marroquin O., Rosa, F., *Ubiquitous Systems and Petri Nets.* Ubiquitous Web Systems and Intelligence, LNCS vol.3481, pags. 1156-1166. Springer-Verlag. (2005).

[48] Gallardo, M.M., Merino, P., Pimentel, E., *Debbuging UML Designs with Model Checking.* Journal of Object Technology, vol 1, no.2 (2002).

[49] Gibb, B.K., Damodaran, S., *ebXML: Concepts and Application.* John Wiley & Sons, Inc. (2002).

[50] Gómez-Sanz, J., Pavón, J., *INGENIAS Development Kit (IDK) Manual.* Facultad de Informática, Universidad Complutense de Madrid, Spain. http://ingenias.sourceforge.net (2004).

[51] Gosling, G., Joy, B., Steele, G., Bracha, G., Buckley, A., *The Java® Language Specification. (PDF doc.)* https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf, (2014).

[52] Grignard, A., Taillandier, P., Gaudou, B., An Vo, D., Huynh, N., Drogoul, A., *GAMA 1.6: Advancing the Art of Complex Agent-Based Modeling and Simulation.* 16th International Conference on Principles and Practices in Multi-Agent Systems (PRIMA), vol. 8291, pags. 242-258. (2013).

[53] Gutknecht, O., Ferber, J., *Madkit: a Generic Multi-Agent Platform.* Autonomous Agents. AGENTS. pags. 78-79, ACM Press. (2000).

[54] Helsinger, A., Wright, T., *Cougaar: A Robust Configurable Multi Agent Platform.* Aerospace Conference, pags. 1-10. IEEE (2005).

[55] Herrero, C., Oliver, J., *Intérprete Prolog para una Máquina Química Abstracta*, Technical Report, DSIC-II/3/95 Universitat Politècnica de València, (1995).

[56] Herrero, C., Oliver, J., *Construcción de una Máquina Química Abstracta.* En J.M. Troya y C. Rodriguez (Eds.) Actas de I Jornadas de Informática, (1995).

[57] Herrero, C., Oliver, J., *Objetos en el λ-cálculo Etiquetado Paralelo.* Technical Report, DSIC-II/39/97 Universitat Politècnica de València, (1997).

[58] Herrero, C., Oliver, J., *Objets in the Parallel Label-Selective λ-calculus*, A.Suárez Sarmiento (Ed.) Actas de IV Jornadas de Informática, (1997).

[59] Herrero, C., Oliver, J., *Object-Oriented Parallel Label-Selective λ-calculus.* H.Hüttel, U. Nestmann (Eds.) BRICS Notes Series, (1997).

[60] Herrero, C., Oliver, J., *The Parallel Multi-Label Selective λ-calculus: An Object Oriented proposal*, Technical Report, DSIC-II/10/99 Universitat Politècnica de València, (1999).

[61] Herrero, C., Oliver, J., *Object-Oriented Specification with the Parallel Label-Selective λ-calculus*, J. Pavelka (Ed.) Proc. of SOFSEM'99, Lecture Notes in Computer Science, Springer Verlag, (1999).

[62] Herrero, C., Oliver, J., *Autómatas Cooperativos para modelizar Sistemas Distribuidos.* En Proc. of IX Jornadas de Concurrencia, Edicions d'Enginyeria i Arquitectura La Salle, pags. 115-128, (2001).

[63] Herrero, C., Oliver, J., *Una extensión de los Autómatas Cooperativos.* En Concurrencia y Sistemas Distribuidos, Treballs d'Informática i Tecnología n. 16, Benicassim (Castellón), pags. 265-279, (2003).

[64] Herrero, C., Oliver, J., *Extended Cooperating Automata.* En Proc. of SMC'2003, 2003 IEEE International Conference On Systems, Man & Cybernetics, IEEE Press, Washington (USA), pags. 402-408, (2003).

[65] Herrero, C., Oliver, J., *Autómatas Cooperativos Extendidos: sistemas multiagente con dependencia geográfica*. Revista Iberoamericana de Inteligencia Artificial, vol. 8, n. 23, pags. 73-84, (2004).

[66] Herrero, C., Oliver, J., *SimECA: Un simulador para los Autómatas Cooperativos*. En Proc. CEDI 2005, JCSD 2005, pags. 205-217, (2005).

[67] Herrero, C., Oliver, J., *Concurrencia y Paralelismo en los ACE*. En Proc. XVI Jornadas de Concurrencia y Sistemas Distribuidos (JCSD 2008), pags. 171-186, (2008).

[68] Herrero, C., Oliver, J., *Autómatas Cooperativos Extendidos vs Redes de Referencias*. En Proc. XVII Jornadas de Concurrencia y Sistemas Distribuidos (JCSD 2009), pags. 51-64, (2009).

[69] Herrero, C., Oliver, J., *A-Priori analysis and search for structural problems in a Multiagent System model*. The Scientific World Journal, Septiembre de 2014. ISSN: 2356-6140. Hindawi Publishing Corporation (2014). Aceptado para publicación. Retirado por los autores.

[70] Herrero, C., Oliver, J., *ECA-Tool downloading page.* http://personales.upv.es/cherrero/ECA, (2015).

[71] Herrero, C., Oliver, J., *A-Priori analysis and search for structural problems in a Multiagent System model*. Applied Mathematics and Computation. ISSN: 0096-3003. Elsevier. (2015). En revisión.

[72] *High-Level Petri Nets Concepts, Definitions and Graphical Notation*. Final Draft International Standard ISO/IEC 15909 Version 4.7.1 October 28, (2000).

[73] Higman, G., *Ordering by divisibility in abstract algebras*. Proc. London Mathematical Society, vol. s3-2, pags. 326-336, (1952).

[74] Hirsch, B., Konnerth, T., Heßler, A., *Merging Agents and Services - the JIAC Agent Platform*. En Multi-Agent Programming: Languages, Tools and Applications, pags. 159-185. (2009).

[75] Himmelspach, J., Uhrmacher, A. M., *Plug'n simulate.* En Proc. of the 40th Annual Simulation Symposium, pags. 137-143. IEEE Computer Society. (2007).

[76] Hinchey, M., and Vassev, E., *Multi-Agent Systems - Theory, Approaches and NASA Applications*. Software Agents, Agent Systems and Their Applications, NATO Science for Peace and Security Series, vol. 32, pags. 181-202. IOS Press. (2012).

[77] Holz, T., Campbell, A.G., O'Hare, G.M.P, Stafford, J.W., Martin, A., Dragone, M., *MiRA - Mixed Reality Agents*. International Journal of Human-Computer Studies, vol. 69 pags. 251-268. (2011).

[78] Hopcroft, J.E., Motwani, R., Ullman J.D., *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Pearson Education, (2000).

[79] Hüttel, H., Kleist, J., Nestmann, U., Sangiorgi, D., *Surrogates in ∅geblik: Towards Migration in Obliq.* H.Hüttel, U. Nestmann (Eds.), BRICS Notes Series, (1997).

[80] Jamroga, W., Penczek, W., *Specification and Verification of Multi-Agent Systems.* Lectures on Logic and Computation, vol.7388, pags. 210-264. (2012).

[81] Jennings, N. R., *An agent-based approach for building complex software systems. Commun.* ACM, 44, 4 (Apr.), pags. 35-41, (2001).

[82] Jensen, K., *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use.* EATCS Monographs on Theoretical Computer Science, (1992,1994,1997).

[83] Jin-Shyan L., Pau-Lo H., *Design and implementation of the SNMP agents for remote monitoring and control via UML and Petri nets.* IEEE Transactions on Control Systems Technology, vol. 12. (2004).

[84] Joshi, A.,Ramakrishnan, N., Houstis, E. N., *Multiagent System Support of Networked Scientific Computing.* IEEE Internet Computing, vol. 2, pags. 69-83, (1998).

[85] Karp, R.M., Miller, R.E., *Parallel program schemata.* Journal of Computer and System Sciences, vol. 3, pags. 147-195, (1969).

[86] Kelly, D., *Teoria de Autómatas y lenguajes formales.* Prentice Hall. (1995).

[87] Keesmatt, N.W., *Vector Controlled Concurrent Systems.* KPN Research. (1996).

[88] Klügl, F., *SeSAm: Visual Programming and Participatory Simulation for Agent-Based Models.* En. D. Weyns and A. Uhrmacher (eds) Agents, Simulations and Applications, cap. 16. (2009).

[89] Kravari, K., Kontopoulos, E., Bassiliades, N., *EMERALD: A Multi-Agent System for Knowledge-based Reasoning Interoperability in the Semantic Web.* 6th Hellenic Conference on Artificial Intelligence (SETN 2010), LNCS, vol. 6040/2010, pags. 173-182. Springer Berlin / Heidelberg. (2010).

[90] Kravari, K., Bassiliades, N., *A Survey of Agent Platforms.* Journal of Artificial Societies and Social Simulation, vol. 18. (2015).

[91] Kripke, S. A., *Semantical Analysis of Modal Logic I Normal Modal Propositional Calculi.* Mathematical Logic Quarterly, vol.9 issue 5-6, pags. 67-96, (1963).

[92] Kummer, O., *Introduction to Petri Nets and Reference Nets.* Universitt Hamburg and Fachbereich Informatik, (2001).

[93] Kummer, O., *Referenznetze* Logos-Verlag, ISBN 3-8325-0035-9. Berlin, (2002).

[94] Kummer, O., F. Wienberg y M. Duvigneau., *The Reference Net Workshop.* web page at http://renew.de/ (2015).

[95] Leitão, P., *Agent-based Distributed Manufacturing Control: A State-of-the-art Survey*, Engineering Applications of Artificial Intelligence, vol. 22, pags. 979-991. Pergamon Press, Inc. (2009).

[96] Le Page, C., Bousquet, F., *The Cormas platform. Tutorial.* 4e Conférence de l'Association Européenne de Simulation Sociale. (ESSA). (2007).

[97] Lin, Hong., *Architectural Design of Multi-Agent Systems: Technologies and Techniques.* ISBN: 1599041081, 9781599041087. IGI Global. (2007).

[98] Lucas, S., Oliver, J., *Parallel label-Selective λ-calculus (LCEP).* En M. Alpuente, R. Barbuti y I. Ramos (Eds.), proc. of 1994 Joint Conference on Declarative Programming GULP-PRODE'94, pags. 125-139. (1994).

[99] Luck, M., McBurney, P., Preist, C., *Agent Technology: Enabling Next Generation Computing (A Roadmap for Agent Based Computing).* AgentLink. (2003).

[100] Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., Balan, G., *MASON: A Multi-Agent Simulation Environment.* En Simulation: Transactions of the Society for Modeling and Simulation International, vol. 82. pags. 517-527. (2005).

[101] Lynch, N.A., Tuttle, M.R., *An introduction to Input/Output Automata.* CWI Qarterly, vol. 2, pags. 219-246, (1989).

[102] Maalal, S., Addou, M., *A new approach of designing Multi-Agent Systems.* International Journal of Advanced Computer Science and Applications(IJACSA), vol. 2 núm. 11. (2011).

[103] Marcheroni, M., Wagner, T., Wüstenberg, L., *Verifying Reference Nets By Means of Hypernets: a Plugin for Renew.* En Proc. of ACSD/Petri Nets Workshops (CEUR Workshop), Donatelli, Susanna and Kleijn, Jetty and Machado, Ricardo Jorge and Fernandes, João M. editors, pags. 285-299, (2010).

[104] Markov, A.A., *Teoriya algorifmov i konstruktivnaya matematika, matematicheskaya logika, informatika i smezhnye voprosy* Izbrannye trudy, tomo 2, (1907).

[105] McCulloch, W., Pitt, W., *A logical calculus of the ideas immanent in nervous activity* Bulletin of Mathematical Biophysics, vol. 5, pags. 115-133, (1943).

[106] North, M.J., Howe, T.R., Collier, N.T., Vos, J.R., *A Declarative Model Assembly Infrastructure for Verification and Validation.* En Advancing Social Simulation: The First World Congress, pags. (129-140). Springer, Heidelberg. (2007).

[107] Milner, R., *Communication and Concurrency.* Prentice Hall, (1989).

[108] Merelli, E., Armano, G., Cannata, N.,Corradini, F., d'Inverno, M., Doms, A., Lord, P. W., Martin, A., Milanesi, L., Möller, S., Schroeder, M., Luck, M., *Agents in bioinformatics, computational and systems biology.*, Briefings in Bioinformatics, vol. 8, (2007).

[109] Murata, T., *Petri nets: Properties, analysis and applications.* En Proc. IEEE, vol. 77, pags. 541-580, (1989).

[110] Nuseibeh, B., Easterbrook, S., *Requirements Engineering: A Roadmap.* En Proceedings of the Conference on The Future of Software Engineering. ICSE '00. pags. 35-46, ACM. (2000).

[111] Oey, M.A., van Splunter, S., Ogston, E., Warnier, M., Brazier, F.M.T., *A Framework for Developing Agent-Based Distributed Applications.* En Proc. of the 2010 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT), vol. 2, pags. 470-474. IEEE/WIC/ACM, (2010).

[112] S. Okamoto, S., Hazon, N., Sycara, K., *Solving Non-Zero Sum Multiagent Network Flow Security Games with Attack Costs.* En Proc. of International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS), Valencia, Spain, (2012).

[113] Oliver, J., *Extensión del λ-cálculo para la modelización de procesos concurrentes.* Tesis Doctoral, Universitat Politècnica de València, (1996).

[114] Park, D., *Concurrency and Automata on Infinite Sequences.* En Proc. of the 5th GI-Conference Karlsruhe. Theoretical Computer Science, Springer-Verlag. vol. 104, pags. 167-183, (1981).

[115] Peterson, J.L., *Petri Net Theory and the Modeling of Systems. Prentice Hall, INC.*. Englewood Cliffs, N.J, (1981).

[116] Petri, C.A., *Fundamentals of a Theory of Asynchronous Information Flow.* En Proc. of IFIP Congress'62, pags. 386-390, (1962).

[117] Polhill, J.G, *Extracting OWL Ontologies from Agent-Based Models: A Netlogo Extension.* Journal of Artificial Societies and Social Simulation, vol. 18. (2015).

[118] Rabin, M. O.; Scott, D., *Finite Automata and Their Decision Problems.* IBM Journal of Research and Development, vol. 3, pags. 114-125, (1959).

[119] RDF Working Group, *RDF 1.1 XML Syntax* W3C Recommendation 25 February 2014 Fabien Gandon, INRIA Guus Schreiber, VU University Amsterdam, eds. (2014).

[120] Reisig, W., *Petri Nets: An Introduction.* Springer-Verlag New York, Inc. (1985).

[121] Ribino, P., Cossentino, M., Lodato, C., Lopes, S., Sabatucci, L., Seidita, V., *Ontology and Goal Model in Designing BDI Multi-Agent Systems.* En Journal WOA@ AI* IA, vol. 1099, pags. 66-72. (2013).

[122] Rogers, H., Jr., *Theory of Recursive Functions and Effective Computability.* McGraw Hill (1967) and MIT Press, Cambridge, Massachusetts. (1987).

[123] Roser, C., Nakano, M., Tanaka, M., *Shifting Bottleneck Detection.* En Proc. of the Winter Simulation Conference. pags. 1079-1086, E. Yucesan, C.-H. Chen, J.L. Snowdon, and J.M. Charnes eds. (2002).

[124] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorenzen, W., *Object-Oriented Modeling and Design.* Prentice-Hall, London. (1991).

[125] Rumbaugh, J. Jacobson, I. and Booch, G., *Unified Modeling Language Reference Manual, The (2nd Edition).* Pearson Higher Education, (2004).

[126] Russell, S., Jordan, H., O'Hare, G.M.P., Collier, R.W., *Agent Factory: A Framework for Prototyping Logic-Based AOP Languages.* En Proc. of the Ninth German Conference on Multi-Agent System Technologies (MATES), pags. 125-136. (2011).

[127] Sabatucci, L., Cossentino, M., Gaglio, S., *A semantic description for agent design patterns.* En Proc. of the Sixth International Workshop"From Agent Theory to Agent Implementation"(AT2AI-6) at The Seventh International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS). (2008).

[128] Sengupta, S., Das, K. and VanTil R.P., *A new method for bottleneck detection.* En Proc. of the Winter Simulation Conference. S. J. Mason, R. R. Hill, L. Münch, O. Rose, T. Jefferson, J. W. Fowler eds. (2008).

[129] Sislak, D., Rehak, M., Pechoucek, M., Pavlicek, D., *Deployment of A-globe Multi-Agent Platform.* En Proc. of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems, pags. 1447-1448. (2006).

[130] Sonnessa, M., *JAS: Java Agent-based Simulation library. User's Guide version 1.0.* http://jaslibrary.sourceforge.net/ (2004).

[131] Stotts, P.D., Pugh, W., *Parallel Finite Automata for Modeling Concurrent Software Systems.* Journal of Systems and Software vol. 27, pags. 27-43, (1994).

[132] Sun Microsystems), *Jini architecture Specification.* http://river.apache.org/doc/specs/html/jini-spec.html, (1998-2015).

[133] Sun Microsystems, *JavaSpaces Specifications.* http://www.oracle.com/technetwork/java/introduction-135941.html, (1998-2015).

[134] Suryanarayanana, V., Theodoropoulos, G., Leesc, M., *Distributed Simulation of Multi-agent Systems.* En Proc. of 2013 International Conference on Computational Science. Procedia Computer Science, vol. 18, pags. 671-681, (2013).

[135] Tisue, S., Wilensky, U., *NetLogo: Design and implementation of a multi-agent modeling environment.* En Proc. of the Agent 2004 Conference on Social Dynamics: Interaction, Reflexivity and Emergence, Chicago, Illinois. (2004 - actualizado en 2013).

[136] Tokmakoff, A., Farkas. A., Mosel, S., *Tailoring Systems Engineering Processes for Interation of Research and Prototyping Activities.* En Proc. de ISSESSs'99, Curitaba, CS Press, (1999).

[137] UPnP forum., *UPnP^{TM} Device Architecture 1.1* (2008).

[138] Valk, R., *Nets in Computer Organization.* Petri Nets: Applications and Rlationship to other Models of Concurrency, vol. 255 of Lecture Notes in Computer Science, W. Brauer, W. Reisig y G. Rozenberg (eds.) Springer Verlag, pags. 218-233, (1986).

[139] Valk, R., *Petri Nets as Token Object.* J. Desel y M. Silva (Eds.) Applications and Theory of Petri Nets, vol. 1420 of Lecture Notes in Computer Science, Springer Verlag, pags. 1-25, (1998).

[140] Valk, R., *Concurrency in Communicating Object Petri Nets.* Advances in Petri Nets: Concurrent Object-Oriented Programming and Petri Nets, pags. 164-195. Springer. (2001)

[141] Walker, D., *Objects in the π-calculus.* Information and Computation, vol. 116, pags. 253-271, (1995).

[142] Walsh, A.E., *Uddi, Soap, and Wsdl: The Web Services Specification Reference Book.* Prentice Hall Professional Technical Reference. (2002).

[143] Wang, J., Song, Y., *Architectures Supporting RosettaNet.* IEEE Computer Society (2006).

[144] Weiser, M., *Some Computer Science Issues in Ubiquitous Computing.* Communications of the ACM July 1993, pags. 74-84. ACM Press. (1993).

[145] Weiser, M., *The Computer for the 21st Century.* En Proc. of Human-Computer Interaction: Toward the Year 2000, pags. 933-940. Morgan Kaufmann Publishers Inc. (1995).

[146] Winikoff, M., *JACK intelligent agents: An industrial strength platform.* En Multi-Agent Programming, vol. 15, pags. 175-193. Springer. (2005).

[147] Wooldridge, M., Jennings, N.R., Kinny, D., *A Methodology for Agent-Oriented Analysis and Design.* En Proc. of the Third Annual Conference on Autonomous Agents, pags. 69-76. ACM. (1999).