Tesis de Posgrado

Modelos de transformación de Grafos para estilos de arquitectura de software

Hirsch, Dan Francisco

2003

Tesis presentada para obtener el grado de Doctor en Ciencias de la Computación de la Universidad de Buenos Aires

Doctorado en Ciencias de la Computación

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Modelos de Transformación de Grafos para Estilos de Arquitectura de Software

por

Dan Francisco Hirsch

Directores:

Prof. Ugo Montanari
Dipartimento di Informatica
Università di Pisa
Corso Italia 40
(56125), Pisa, Italia

Prof. Daniel Yankelevich
Departamento de Computación
Universidad de Buenos Aires
Pabellón 1 - Planta Baja - Ciudad Universitaria
(1428), Buenos Aires, Argentina

Facultad de Ciencias Exactas y Naturales

Para mayor información, por favor contactar

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Pabellón 1 - Planta Baja - Ciudad Universitaria
(1428), Buenos Aires, Argentina
phone/fax: +54-11-45763359
e-mail: compuba@dc.uba.ar

Departamento de Computación

Doctorado en Ciencias de la Computación

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Graph Transformation Models for Software Architecture Styles

by

Dan Francisco Hirsch

Advisors:

Prof. Ugo Montanari
Dipartimento di Informatica
Università di Pisa
Corso Italia 40
(56125), Pisa, Italia

Prof. Daniel Yankelevich
Departamento de Computación
Universidad de Buenos Aires
Pabellón 1 - Planta Baja - Ciudad Universitaria
(1428), Buenos Aires, Argentina

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

e-mail: dhirsch@dc.uba.ar
http://www.dc.uba.ar/people/exclusivos/dhirsch

Cover design by Dan Hirsch.

Contents

Abstract

Un *Estilo de Arquitectura de Software* es una clase de arquitecturas que exhiben un patrón común. Si la reusabilidad y la clasificación de arquitecturas son objetivos principales para permitir explotar trabajo previo, la identificación de una arquitectura dentro de un estilo específico, requiere lenguajes expresivos y bien fundados para representar estilos. Los requerimientos de los sistemas modernos incluyen distribución, concurrencia, reconfiguración y movilidad. Por lo tanto, es necesario desarrollar el lenguaje de estilos y por consiguiente su semántica formal. Esta tesis presenta un marco formal para describir estilos de arquitectura de software basado en los sistemas de transformación de grafos. En particular, para describir estilos elegimos utilizar gramáticas libres de contexto para *Reescritura de Hiperejes* (*Hyperedge Replacement (HR)* grammars) y gramáticas HR de Sincronización (*Synchronized HR (SHR)* grammars) para modelar la comunicación y la coordinación entre componentes. En la segunda parte de la tesis continuamos usando gramáticas HR (sin sincronización) para describir estilos e introducimos un enfoque específico, basado en teoría de tipos, para describir las reconfiguraciones (llamadas *transformaciones*). Por analogía, podemos pensar en estilos como tipos para las instancias de arquitectura y en derivaciones HR como pruebas de tipo. Las reconfiguraciones deben preservar los tipos: en terminología de la teoría de tipos esta propiedad se llama *subject reduction*. Nuestro método consiste en definir transformaciones sobre pruebas de tipo. Los diferentes temas en la tesis son acompañados por ejemplos específicos y por un caso de estudio que es utilizado a lo largo de la tesis.

Resumen

La evolución de los sistemas de software y la creciente complejidad de sus procesos de desarrollo han hecho necesario establecer pasos de diseño bien definidos que permitan reducir el gap entre requerimientos e implementaciones. En esta dirección, en años recientes, un área importante de la investigación ha tratado con las *Arquitecturas de Software*. Básicamente, una arquitectura de software es una descripción de alto nivel de un sistema complejo, con un nivel adecuado de abstracción que permite capturar del dominio del problema *los componentes*, a partir de los requerimientos, que serán diseñados detalladamente más adelante. Un *Estilo de Arquitectura de Software* es una clase de arquitecturas que exhiben un patrón común. Si la reusabilidad y la clasificación de arquitecturas son objetivos principales para permitir que los diseñadores exploten el trabajo previo, la identificación de una arquitectura de un sistema dentro de un estilo específico requiere lenguajes expresivos y bien fundados para representar estilos.

Los requerimientos de los sistemas modernos imponen nuevas características a la descripción de las arquitecturas de software. Estos incluyen la distribución, la concurrencia, la reconfiguración y la movilidad. Por lo tanto, es necesario desarrollar el lenguaje de estilos que soporte estas características y por consiguiente su semántica formal. Entonces, la descripción de un estilo de arquitectura de software debe incluir la estructura de los tipos de componentes y sus interacciones, los patrones de comunicación, y las leyes que gobiernan la reconfiguración y/o la movilidad en la arquitectura. Siguiendo esta línea de investigación, es nuestro objetivo contribuir a la formalización de modelos para la descripción de arquitecturas y estilos de arquitectura de software.

Esta tesis presenta un marco formal para la descripción de estilos de arquitectura de software basado en los sistemas de transformación de grafos. Una gramática de grafos caracteriza una clase de grafos que comparten características estructurales y está compuesta por un conjunto de reglas de reescritura llamadas *producciones*. Inicialmente, los grafos representan las configuraciones estáticas de las arquitecturas y las gramáticas representan los estilos. Luego, podremos incluir producciones que especifiquen las interacciones en *runtime* entre componentes, la reconfiguración y la movilidad.

Las arquitecturas de sistemas pueden ser controladas de manera centralizada a través de un coordinador o administrador, o pueden ser sistemas denominados *self organising* en los cuales la coordinación está distribuida localmente entre los componentes. Los

sistemas que tenemos en mente son distribuidos, heterogéneos y móviles, y por lo tanto elegimos una estrategia self organising. En particular, para describir estilos elegimos utilizar gramáticas libres de contexto para *Reescritura de Hiperejes* (*Hyperedge Replacement (HR)* grammars) y gramáticas HR de Sincronización (*Synchronized HR (SHR)* grammars) para modelar la comunicación y la coordinación entre componentes. Un *hipereje* es un elemento atómico con una etiqueta (de un alfabeto con rango) y con tantos tentáculos como el rango de su etiqueta. Un conjunto de nodos y un conjunto de hiperejes forman un *hipergrafo* si cada hipereje está conectado, por sus tentáculos, con sus nodos de enlace (*attachment nodes*). Los hiperejes corresponden a los componentes y sus nodos de enlace son sus puertos de comunicación con otros componentes. Una producción HR reescribe un solo hipereje en un hipergrafo cualquiera. Entonces, para modelar la coordinación de los componentes de la arquitectura combinamos reescritura de grafos con condiciones de sincronización, obteniendo los sistemas SHR. Las producciones sincronizadas se especifican agregando condiciones en los nodos que permiten coordinar varias reescrituras, determinando la manera en que los componentes interactúan y se reconfiguran. Las producciones para (la etiqueta de) un hipereje determinado representan las posibles evoluciones para un determinado tipo de componente del estilo. Las producciones de una gramática se agrupan en tres conjuntos: el primer conjunto contiene las producciones HR para la construcción de todas las posibles configuraciones iniciales del estilo. El segundo conjunto contiene las producciones SHR que modelan la evolución de la comunicación para cada tipo de componente, y el tercer conjunto contiene las producciones SHR para la reconfiguración de la estructura del estilo. Las producciones de la comunicación requieren sincronización pero no pueden cambiar la estructura topológica del grafo.

Con respecto a literatura anterior, ésta tesis presenta una extensión de los sistemas SHR con la adición de *movilidad de nombres* (*name mobility* como en π-calculus). Esta extensión permite aumentar substancialmente el poder expresivo del método para la representación de sistemas complejos móviles y reconfigurables, manteniendo al mismo tiempo la capacidad de describirlos de una manera descentralizada y distribuida.

Representamos a los hipergrafos y los sistemas SHR en forma textual usando *syntactic judgements*. Esto permite una clara separación entre reescritura y coordinación, y la introducción de varios mecanismos de sincronización como adecuadas álgebras (móviles) de sincronización. Específicamente, presentamos las reglas de inferencia al estilo SOS para las álgebras de sincronización *Hoare* (CSP) y *Milner* (CCS, π-calculus). Sin embargo, en nuestra propuesta extendemos las álgebras de proceso permitiendo la sincronización simultánea de cualquier número de participantes. Las condiciones de sincronización para movilidad se resuelven vía unificación.

Como resultado importante y evidencia formal del poder expresivo del método, presentamos un resultado de correspondencia que prueba que SHR con sincronización de Milner subsume al π-calculus. Para esto, definimos una traducción donde una transición en π-calculus se representa como una transición del correspondiente *syntactic judgement* traducido (es decir, un paso de reescritura). Puesto que π-calculus está equipado con una semántica *interleaving* y sólo sincronizaciones entre pares de elementos, la prueba es una correspondencia completa entre π-calculus y una versión restringida de los sistemas SHR.

En la segunda parte de la tesis continuamos usando gramáticas HR (sin sincronización) para describir estilos e introducimos un enfoque específico basado en teoría de tipos para describir las reconfiguraciones de estilos (llamadas *transformaciones*). Las nociones de reconfiguración y movilidad implican modificaciones a la estructura de la arquitectura cambiando componentes y conexiones. Estas modificaciones que un sistema puede sufrir conducen a la pregunta de cómo podemos asegurar que los cambios sean consistentes con el estilo al cual pertenece el sistema. Por analogía, podemos pensar en los estilos como tipos para las instancias de arquitectura y en derivaciones HR como pruebas de tipo. Las reconfiguraciones deben preservar los tipos: en terminología de la teoría de tipos esta propiedad se llama *subject reduction*. Nuestro método consiste en definir transformaciones sobre pruebas de tipo, en lugar que sobre grafos: mientras que cortar y pegar pruebas de tipo resulten nuevamente en pruebas de tipo, la propiedad de subjet reduction está garantizada. La formalización se efectúa representando grafos y producciones como términos de un cálculo λ tipado, donde un paso de derivación HR corresponde a *aplicación* seguido por reducción β. Entonces, las transformaciones se especifican como reescritura de términos (de alto orden): si todas las reglas de reescritura transforman pruebas de tipo en pruebas de tipo, entonces todas las posibles reescrituras satisfacen subject reduction. El uso de cálculo λ introduce la idea de *reescritura de grafos de alto orden*, permitiendo la parametrización del proceso de diseño con características de componentes y conectores que podrían ser especificados más adelante, manteniendo la garantía de consistencia.

La principal diferencia del método para reconfiguración usando SHR con respecto al que utiliza transformaciones, es que SHR es más dinámico, ya que es aplicable a sistemas abiertos en ejecución sin control global, a excepción de la sincronización. Por el otro lado, el método con transformaciones puede ser útil para trabajar al nivel de diseño estático, es decir, cambiar los pasos del diseño del sistema para producir un sistema diverso pero consistente. De todas maneras, el último método puede ser aplicado para especificar clases muy generales de reconfiguraciones y movilidad (como se demuestra en los ejemplos de la tesis), pero requiere un conocimiento global de la estructura del sistema.

Los diferentes temas introducidos en la tesis son acompañados por ejemplos específicos que permiten clarificar las diversas construcciones y demostrar su poder expresivo, y por un caso de estudio de un *sistema remoto de asistencia médica* que es utilizado a lo largo de toda la tesis.

Extended Abstract

The evolution of software systems and the increased complexity of their developing processes have led to the necessity of establishing well defined design steps to close the gap between requirements and implementations. In this direction, in recent years a main research area has concerned *Software Architectures*. Basically, a software architecture is a high-level description of a complex system, with an adequate level of abstraction that enables capturing, from its requirements, the *components* of the problem domain to be later designed in more detail. A *Software Architecture Style* is a class of architectures exhibiting a common pattern. If reusability and classification of architectures are main goals to allow designers to exploit previous work, the identification of a system architecture within a specific style requires expressive and well founded languages to represent styles.

The requirements of modern systems impose new characteristics to the description of software architectures. These include distribution, concurrency, reconfiguration and mobility. Therefore, it is necessary to develop the style language and its formal semantics accordingly. Thus, the description of a software architecture style must include the structure of component types and of their interactions, the communication patterns, and the laws governing reconfiguration and/or mobility changes in the architecture. In this line of research our goal is to contribute to the formalization of models for the description of software architectures and of software architecture styles.

This thesis presents a formal framework based on graph transformation systems for the description of software architecture styles. A graph grammar characterizes a class of graphs that share structural characteristics and it is composed of a set of rewriting rules called *productions*. At first, graphs represent the static configurations of architectures and grammars represent styles. Later we will be able to include productions specifying runtime interactions among components, reconfiguration and mobility.

System architectures can be managed in a centralized manner by an explicit coordinator or administrator, or can be *self organising* indicating that coordination management is distributed locally among components. The systems we have in mind are distributed, heterogeneous and mobile, thus we choose a self-organising approach. In particular, we choose context-free *Hyperedge Replacement* (HR) grammars to describe styles and *Synchronized* HR (SHR) grammars to model communication and coordination among

components. A *hyperedge* is an atomic item with a label (from a ranked alphabet) and with as many tentacles as the rank of its label. A set of nodes together with a set of hyperedges form a *hypergraph* if each hyperedge is connected, by its tentacles, to its attachment nodes. Hyperedges correspond to components and their attachment nodes are their communication ports with other components. A HR production rewrites a single hyperedge into an arbitrary hypergraph. Then, to model the coordination of the architecture components we combine graph rewriting with synchronizing conditions obtaining SHR systems. We specify synchronized productions by adding conditions on nodes which allow to coordinate several rewritings, thus determining how components interact and are reconfigured. The productions for a given hyperedge (label) represent the possible evolutions for a given component type of the style. The productions of a grammar are grouped in three sets: the first set contains the HR productions for the construction of all possible initial configurations of the style. The second set contains the SHR productions that model the communication evolution for each component type and the third set contains the SHR productions for the reconfiguration of the style structure. The communication productions require synchronization but cannot change the topological structure of the graph.

With respect to previous literature, this thesis presents an extension of SHR with the addition of *name mobility* (as in π-calculus). This extension allows us to substantially increase the expressive power of the approach for representing complex mobile and reconfigurable systems, still maintaining the ability of describing them in a decentralized, distributed way.

We represent hypergraphs and SHR systems in textual form using syntactic judgements. This allows the clear separation of rewriting and coordination and the introduction of various synchronization mechanisms as suitable (mobile) synchronization algebras. Specifically, we present the inference rules in the SOS style for *Hoare* (CSP) and *Milner* (CCS, π-calculus) synchronization algebras. However, we extend process algebras in that we allow synchronizations of any number of partners at the same time. Constraint conditions for mobility are solved via unification.

As an important outcome and a formal evidence of the expressive power of the approach, we present a correspondence result proving that SHR with Milner synchronization subsumes π-calculus. We define a translation where a transition in the π-calculus is represented as a transition of the corresponding translated judgement (i.e. a rewriting step). Since π-calculus is equipped with an interleaving semantics and only with two-partner synchronizations, we prove a complete correspondence between π-calculus and a restricted version of SHR systems.

In the second part of the thesis we continue using HR grammars (without synchronization) for describing styles and we introduce a specific, type-based approach for describing style reconfigurations (called *transformations*). The notions of reconfiguration and mobility imply changes to the architecture structure by changing components and connections. These modifications that a system may suffer lead to the question of how we assure that changes are consistent with the style the system belongs to. By analogy, we can think of styles as types for the architecture instances and of HR derivations as typing proofs. Reconfigurations must preserve types: in type theory terminology, this property is called subject reduction. Our approach is to define transformations on typ-

ing proofs rather than on graphs: as long as cutting and pasting typing proofs still yields typing proofs, subject reduction is guaranteed. The formalization is done by representing both graphs and productions as terms of a typed λ-calculus, where a HR derivation step corresponds to application followed by β-reduction. Then, transformations are specified as (higher order) term rewritings: if all the term rewriting rules transform typing proofs into typing proofs, then all the possible rewritings satisfy subject reduction. The use of λ-calculus introduces the idea of *higher order graph rewriting*, allowing to parameterize the design process with component and connector features which could be specified later, still guaranteeing consistency.

The main difference of the approach for reconfiguration using SHR with respect to the one using transformations, is that SHR is more dynamic in the sense that it applies to running open-ended systems without global control except for synchronization, whereas the approach with transformations may be useful for working at the level of blueprints, i.e. it rearranges the design steps of the system to produce a different but consistent system. Thus, the latter method can be applied to specify very general kinds of reconfigurations and mobility (as it is shown in the thesis examples), but it requires a global knowledge of system structure.

The different topics introduced in the thesis are accompanied by specific examples to clarify the various constructions and to show their expressive power, and by a case study of a *remote medical care system* that is used all along the thesis.

Part I

Introduction

Chapter 1

Modeling Software Architectures and Software Architectures Styles

1.1 Software Architectures and Software Architecture Styles

The evolution of software systems and the increased complexity of their developing process (including the maturity of the field as an engineering discipline) have led to the necessity of establishing well defined design steps to close the gap between requirements (related to the domain of a problem to solve) and implementations. In this direction, in recent years a main research area has concerned *Software Architectures*. Basically, a software architecture is a high-level description of a complex system, with an adequate level of abstraction that enables capturing, from its requirements, the *components* of the problem domain to be later designed in more detail. A software architecture description is intended to give all participants of a project a general knowledge of the structure of the system to be implemented.

As some of the well known definitions of software architectures we can quote:
From [Perry, D. and Wolf, A., 1992],

> *Architecture is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for the design.*
>
> *Some of the benefits we expect to gain from the emergence of software architecture as a major discipline are: 1) architecture as the framework for satisfying requirements; 2) architecture as the technical basis for design and as the managerial basis for cost estimation and process management; 3) architecture as an effective basis for reuse; and 4) architecture as the basis for dependency and consistency analysis.*

From [Garlan, D. and Shaw, M., 1996],

> *As the size and complexity of software systems increase, the design and specification of overall systems structure become more significant issues than the choice of algorithms and data structures of computation. Structural issues include the organization of a system as a composition of components; global control structures; the protocols for communication, synchronization, and data access; the assignment of functionality to design elements; the composition of design elements; physical distribution; scaling and performance; dimensions of evolution; and selection among design alternatives. This is the* **Software Architecture** *level of design.*
>
> *Abstractly, software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns.*

From [Hofmeister, C. *et al.*, 1999a],

> *The two main aspects of a Software Architecture are that it provides a design plan - a blue print - of a system, and that it is an abstraction to help manage the complexity of a system.*
>
> *This design plan isn't a project plan that describes activities and staffing for designing the architecture or developing the product. Instead, it is a structural plan that describes the elements of the system, how they fit together, and how they work together to fulfill the system's requirements. It is used as a blueprint during the development process, and it is also used to negotiate system requirements, and to set expectations with customers, and marketing and management personnel. The project manager uses the design plan as input to the project plan.*
>
> *The other main aspect of software architecture is that it is an abstraction that helps manage complexity. The software architecture is not a comprehensive decomposition or refinement of the system: Many of the details needed to implement the system are abstracted and encapsulated within an element of the architecture.*
>
> *The software architecture should define and describe the elements of a system at a relatively coarse granularity. It should describe how the elements fulfill the system requirements, including which elements are responsible for which functionality, how they interact with each other, how they interact with the outside world, and their dependencies on the execution platform.*

A *Software Architecture Style* is a class of architectures exhibiting a common pattern. This allows to abstract details of particular components giving a way to categorize paradigms of architectures and to obtain new instances of systems from a specific style. A style can be seen as a type for a given architectural instance.

If reusability and classification of architectures are main goals to allow designers to exploit previous work at an early stage of the development process, the identification of a system architecture within a specific style requires expressive and well founded languages to represent styles.

1.1. Software Architectures and Software Architecture Styles

To support architecture-based development, formal modeling notations and analysis and development tools that operate on architectural specifications are needed. *Architecture Description Languages (ADLs)* and their accompanying toolsets have been proposed as the answer. Loosely defined, an ADL for software applications focuses on the high-level structure of the overall application rather than the implementation details of any specific source module [Medvidovic, N. and Taylor, R., 1997].

An ADL provides a way of specifying the elements used in the architecture, generally both as types and instances. It also provides support for interconnecting element instances to form a configuration [Hofmeister, C. *et al.*, 1999a]. As a model of a system at a high level of abstraction, an ADL is intended (and can only be expected) to provide a *partial* depiction of the system.

For the formal description of software architectures many methods have been used. Research included revision of semantic models used in other areas. We can mention work done with languages and models as Z [Allen, R. and Garlan, D., 1992], CHAM [Inverardi, P. and Wolf, A. L., 1995], Darwin [Magee, J. and Kramer, J., 1996a], Wright [Allen, R. and Garlan, D., 1994]. Other projects for the development of ADLs are C2 [Medvidovic, N. *et al.*, 1996], SADL [Moriconi, M. *et al.*, 1995], Rapide [Lukham, D. *et al.*, 1995] and others. For a comparison of the ADLs produced in the last years, Medvidovic and Taylor present in [Medvidovic, N. and Taylor, R., 1997] a classification and comparison framework for a deep analysis of their characteristics.

The requirements of modern systems impose new characteristics to the description of software architectures. These include distribution, concurrency, reconfiguration and mobility. The evolution of models for the description of software architectures first talked about the description of configurations, components (and connectors) and interactions among components. Then, expansion, distribution and heterogeneity of systems (for example Internet and the World Wide Web) impose the possibility of changing the configuration of system networks, upgrade of components and subsystems; and addition and removal of components without a prior knowledge of how the system will evolve (for example adhoc networks and peer-to-peer systems). And finally, as a last step in the evolution, the integration of (logical or physical) mobility of systems (for example mobile agents and wireless technology). All these issues talk about system reconfiguration, which means that there is an evolution of the architecture reflected in changes over the structure of configurations. Therefore, it is necessary to develop the style language and its formal semantics accordingly.

Even though the relevance of addressing reconfiguration is widely recognized at the architectural level, the ability of ADLs to express reconfiguration differs among the different languages [Medvidovic, N. and Taylor, R., 1997]. In this thesis we will distinguish between two types of reconfiguration from the architectural point of view:

- **Static Reconfiguration:** Changes take place "off-line" at the architecture level and may be related with new or revision of requirements, new design decisions or evolution of the design process during the generation and/or maintenance of a software architecture.

- **Dynamic Reconfiguration:** Changes take place at run-time, i.e. during execution of the corresponding system. The simplest kind of dynamic reconfiguration

is the addition and removal of components. More complex reconfigurations are changes in the topology of the architecture and can include also mobility of components.

Moreover, we are convinced that reconfiguration has to be treated as a main issue at the level of style descriptions. Thus, the description of a software architecture style must include the structure of component types and of their interactions, the communication patterns, and the laws governing reconfiguration and/or mobility changes in the architecture. In this line of research our goal is to contribute to the formalization of models for the description of software architectures and of software architecture styles.

1.2 Why Graphs and Graph Transformations?

The description of software architectures may include various types of diagrams that model different aspects of a system. These diagrams can have a formal basis with well-defined semantics [Medvidovic, N. and Taylor, R., 1997] or can be used in a more informal way simply as a notational standard [Booch, G. *et al.*, 1999]. Most of the informal approaches to represent software architectures have been based in box-and-line drawing with the intention to communicate, among a project team, the intuition of the system structure under development. But also, this lack of formality prevents to clearly define the semantics of components and connections and the specification of relevant system properties. In any case, all these diagrams can be abstracted as some types of graphs. Therefore, we propose using graph models as a suitable abstraction for the architectural level with the benefits of a formal foundation together with its intuitive graphical approach.

More specifically, this thesis presents a formal framework based on graph transformation systems for the description of software architecture styles. A graph grammar characterizes a class of graphs that share structural characteristics and it is composed of an initial graph and set of rewriting rules called *productions*. Applying rewriting rules to graphs yields *graph rewritings* or *graph transformations*. The rewriting of a graph applying a grammar production corresponds to a *direct derivation step* and a sequence of direct derivation steps starting from an initial graph is a *derivation*. At first, graphs represent the static configurations of architectures and grammars represent styles. Later we will be able to include productions specifying runtime interactions among components, reconfiguration and mobility, allowing their explicit description at the level of style language. Then, grammar derivations will show evolution of system configurations. The use of graph transformations to specify systems is not new and the amount of work in this area shows its relevance in computer science. Besides the extensive work on foundations [Rozenberg, 1997], we can mention its application in the areas of concurrency, parallelism and distribution [Ehrig, H. *et al.*, 1999b]; for functional languages, visual and object-oriented languages and software engineering [Ehrig, H. *et al.*, 1999a]; and the proposal of general formal frameworks for system modeling [Engels, G. and Heckel, R., 2000; Mens, T., 2000].

System architectures can be managed in a centralized manner by an explicit coordinator or administrator, or can be *self organising* [Magee, J. and Kramer, J., 1996b]

indicating that coordination management is distributed locally among components. The systems we have in mind are distributed, heterogeneous and mobile, thus we choose a self-organising approach. In particular, we choose context-free *Hyperedge Replacement* (HR) grammars [Drewes, F. *et al.*, 1997] to describe styles and *Synchronized* HR (SHR) grammars to model communication and coordination among components (SHR was first used to represent distributed systems [Degano, P. and Montanari, U., 1987; Montanari, U. *et al.*, 1999]). A *hyperedge* is an atomic item with a label (from a ranked alphabet) and with as many tentacles as the rank of its label. A set of nodes together with a set of hyperedges form a *hypergraph* if each hyperedge is connected, by its tentacles, to its attachment nodes. Hyperedges correspond to components and their attachment nodes are their communication ports with other components. A HR production rewrites a single hyperedge into an arbitrary hypergraph. HR systems allow the application of more than one production at the same time but the specification of components may require that various components need to synchronize. Then, to model the coordination of the architecture components we combine graph rewriting with synchronizing conditions obtaining SHR systems. We specify synchronized productions by adding conditions on nodes which allow to coordinate several rewritings (called *synchronized rewriting*), thus determining how components interact and are reconfigured. The productions for a given hyperedge (label) represent the possible evolutions for a given component type of the style. Using SHR let each component locally define the coordination with other components, supporting a self organising approach. Then, synchronizing two or more productions correspond to the application of a rewriting rule obtained (resolving the synchronizing conditions) from the combination of context-free productions of the respective components. The productions of a grammar are grouped in three sets: the first set contains the HR productions for the construction of all possible initial configurations of the style. The second set contains the SHR productions that model the communication evolution for each component type and the third set contains the SHR productions for the reconfiguration of the style structure. The communication productions require synchronization but cannot change the topological structure of the graph. We first presented these ideas in [Hirsch, D. *et al.*, 1998] and [Hirsch, D. *et al.*, 1999]. The problem of finding the set of productions to use in a synchronized rewriting step is called *the rule-matching problem* [Degano, P. and Montanari, U., 1987]. The solution of the rule-matching problem is implemented considering it as a finite domain constraint problem [Mackworth, A.K., 1998]. An analysis of some techniques to solve this problem in a distributed and efficient way can be found in [Montanari, U. and Rossi, F., 1999; Montanari, U. *et al.*, 1999]. The description of these techniques is out of the scope of this thesis.

With respect to previous literature, this thesis presents an extension of SHR with the addition of *name mobility* (as in π-calculus [Milner, R., 1999; Sangiorgi, D. and Walker, D., 2001]). This extension allows us to substantially increase the expressive power of the approach for representing complex mobile and reconfigurable systems, still maintaining the ability of describing them in a decentralized, distributed way. The extension is obtained by adding the capability of creation and sharing of names to the definition of SHR productions. Now, a condition on a node is accompanied by a tuple of node names that the edge to be replaced wants to share during synchronization. This means

that a component is sharing the name of some of its ports with other components of the architecture. In this way, we have the synchronization of multiple components using SHR, and at the same time, the possibility of rearranging the topology of the graph by name passing. The locality of synchronizations together with context free productions allow us to achieve complex coordination of components along a graph without fixing *a priori* the number of participants of the resulting synchronized rewriting step. This feature constitutes a powerful support of the model for the description of distributed systems. This work was first introduced in [Hirsch, D. *et al.*, 2000] and continued in [Hirsch, D. and Montanari, U., 2001a; Hirsch, D. and Montanari, U., 2001b].

In the second part of the thesis we continue using HR grammars (without synchronization) for describing styles and we introduce a specific, type-based approach for describing style reconfigurations (called *transformations*). The notions of reconfiguration and mobility imply changes to the architecture structure by changing components and connections. These modifications that a system may suffer lead to the question of how we assure that changes are consistent with the style the system belongs to. By analogy, we can think of styles as types for the architecture instances and of HR derivations as typing proofs. Reconfigurations must preserve types: in type theory terminology, this property is called subject reduction. Our approach is to define transformations on typing proofs rather than on graphs: as long as cutting and pasting typing proofs still yields typing proofs, subject reduction is guaranteed. In this way, once a transformation is obtained it is assured that it is a consistent reconfiguration with respect to the style. Thus, a transformation is applied over a derivation segment returning a new derivation segment. After applying the transformation over a derivation, its result corresponds to the reconfigured segment of a new derivation. It is important to mention that because transformations are over derivation segments they can be composed and applied to several parts of a graph. Also, after a transformation is obtained, you can start from any of the derivations of the new resulting graph allowing the application of other transformations. Our aim is to give architects a tool to specify in a consistent way complex reconfigurations over the architectures they are working with. This is fundamental for software architecture modeling because once a transformation is obtained and its correctness checked (that it starts and ends with valid derivations), it can be included in a library of transformations for its future use. This work was introduced in [Hirsch, D. and Montanari, U., 2000] and [Hirsch, D. and Montanari, U., 1999].

The main difference of the approach for reconfiguration using SHR with respect to the one using transformations, is that SHR is more dynamic in the sense that it applies to running open-ended systems without global control except for synchronization, whereas the approach with transformations may be useful for working at the level of blueprints, i.e. it rearranges the design steps of the system to produce a different but consistent system. Thus, the latter method can be applied to specify very general kinds of reconfigurations and mobility (as it is shown in the thesis examples), but it requires a global knowledge of system structure. Some possible steps for combining both methods are described in the future work in Chapter 10. In [Medvidovic, N. and Taylor, R., 1997] the author differentiate these concepts as evolvability and dynamism, where, evolution, refers to "off-line" changes to an architecture (what we called static reconfiguration), and dynamism, on the other hand, refers to modifying the architecture while the system

is executing (what we called dynamic reconfiguration).

1.3 Thesis Results

Resuming what we have introduced above, the main goal of this thesis is motivated in the generation of supporting formal models for the description of software architecture styles. On one side, we choose software architecture as the focus of our research based on its recognized importance in the development of a mature software engineering process. On the other side, we use graph transformation systems as supporting model based on its, already stated, usefulness for the high level description of systems and their evolution. But also, we realized that the results we have obtained go beyond the domain of software architectures. Besides the general goal of the thesis we can summarized its main contributions:

First of all, we present an extension of SHR systems with the addition of *name mobility*. This extension allows us to increase the expressive power of HR obtaining the good characteristics of a graphical calculus and the possibility for describing complex mobile and reconfigurable systems. In this way we are able to propose an alternative modelling approach following a self organising philosophy (using local information without the necessity of a global control) that gives a useful solution for more real problems of distributed systems. This approach contrasts with push-out graph transformation models [Rozenberg, 1997] where graph rewriting is specified by context-sensitive rules implying centralized control.

We represent hypergraphs and SHR systems in textual form using syntactic judgements. This allows the clear separation of rewriting and coordination and the introduction of various synchronization mechanisms as suitable (mobile) synchronization algebras. Specifically, we present the inference rules in the SOS style for *Hoare* (CSP) and *Milner* (CCS, π-calculus) synchronization algebras. However, we extend process algebras in that we allow synchronizations of any number of partners at the same time. Synchronizing conditions for mobility are solved via unification. We have to mention that the initial work of SHR [Degano, P. and Montanari, U., 1987; Montanari, U. and Rossi, F., 1999; Montanari, U. *et al.*, 1999] only uses Hoare synchronization (without mobility).

As an important outcome and a formal evidence of the expressive power of the approach, we present a correspondence result proving that SHR with Milner synchronization subsumes π-calculus. We define a translation where a transition in the π-calculus is represented as a transition of the corresponding translated judgement (i.e. a rewriting step). At this point, it is necessary to comment on the differences between π-calculus and synchronized rewriting. On the one hand, π-calculus is equipped with an interleaving semantics and only with two-partner synchronizations. On the other, we are using graph transformations with synchronized rewriting which is a distributed concurrent model allowing for multiple, simultaneous synchronizations and rewriting. It is clear that there are graph transitions (the concurrent ones) that cannot be obtained in π-calculus. Thus, we prove a complete correspondence between π-calculus and a restricted version of SHR systems. For this we restrict the synchronization mechanism to what we call the Milner$_\pi$ transition system. Proofs constructed with Milner$_\pi$ transition

system are transitions corresponding to sequential steps of π-calculus.

We consider this correspondence result as fundamental for the thesis given the relevance of π-calculus for process calculi specially for mobility languages. The fact that we obtain a graphical calculus that subsumes π-calculus is a strong support for graphical formal languages as a next step in the high-level description of distributed, concurrent and mobile systems.

The second part of the thesis introduce a type-based approach for describing reconfiguration rules (transformations), where styles are seen as types and HR derivations as typing proofs. The emphasis of this approach is on maintaining style consistency after reconfiguration, where consistency is obtained by the subject reduction property. The formalization is done by representing both graphs and productions as terms of a typed λ-calculus, where a HR derivation step corresponds to application followed by β-reduction. Then, transformations are specified as (higher order) term rewritings: if all the term rewriting rules transform typing proofs into typing proofs, then all the possible rewritings satisfy subject reduction. Furthermore, the use of λ-calculus introduces the idea of *higher order graph rewriting*, allowing to parameterize the design process with component and connector features which could be specified later, still guaranteeing consistency.

Finally, the different topics introduced in the thesis are accompanied by specific examples to clarify the various constructions and to show their expressive power, and by a case study of a *remote medical care system* that is used all along the thesis.

1.4 Related Work

As related work using graph transformation for describing software architectures we have [Le Métayer, D., 1998] which has been the first to propose the use of graph grammars for describing software architecture styles. In [Le Métayer, D., 1998], Le Métayer presents a dual approach with context-free productions where nodes represent components and edges their communication links. Graphs are used to represent architectural instances where a graph is formally defined as a set of relation tuples. Also reconfiguration is treated but using a centralized approach. Together with the context-free grammar a coordinator is defined that is in charge of managing the architecture reconfiguration. The coordinator is expressed using conditional graph rewriting with rules (not context-free) on entities, links and conditions on public variables of entities textual specification. These variables are the only interactions among entities and the coordinator and are set by entities (in the textual language specification) indicating when a rule can be applied. In [Le Métayer, D., 1998] only binary and unary relations are used.

Le Metayer complements the graph grammar for the style with a small language with a **CSP** like notation used for describing entity (i.e. components) behavior including the pattern of interactions among them. The difference with our approach is that Le Metayer only uses grammars to generate the static configuration of the style. He also uses context-free productions but they are applied over unary relations that identify components. In spite of the fact that both approaches highlights the separation of static configuration, coordination and computation, the graph representation using relations mixes the static configuration with the dynamics of the communications (see Section 4.4) by connecting

components with communication links, instead of using ports (or connectors generally speaking). In our case, synchronized context-free productions can do the job while keeping the self organising philosophy we propose.

Again, as related work on specifying consistent reconfigurations we can mention [Le Métayer, D., 1998]. In his paper Le Métayer, together with the context free grammar for the static structure of the style and the coordinator conditional rules for reconfiguration, proposes a semi-decidable algorithm for "type checking" of styles to ensure that the coordinator does not break the style structure (which defines the type). The algorithm corresponds to a proof of convergence of graph rewriting rules. On the contrary, in our approach using λ-calculus the consistency of the reconfiguration does not need to be verified given that it is assured by the typing rules of the grammar. Also, we have to mention that part of the checking that is done in [Le Métayer, D., 1998] for the coordinator rules, is due to the set representation that was chosen. For example, when a component is removed it means that the coordinator rule has to take into account the deletion of the node and also all the edges corresponding to its communication links (this is not needed in our representation).

Other work on graph transformation for reconfiguration and description of software architectures and styles are [Wermelinger, M., 1999] and [Wermelinger, M. and Fiadeiro, J., 2002]. These works use a program design language (*COMMUNITY*) to represent program states and computations, and an algebraic framework based on category theory to represent architectures and their reconfigurations. In these case architectures are diagrams of a specific category with designs as nodes and morphisms among designs as arrows. Designs are graphs whose nodes are programs (written in *COMMUNITY*) and arcs denote superposition relationships. Then, reconfiguration is specified using conditional graph rewriting rules that depend on the state of the involved components (designs). Reconfiguration rules are based on the double-pushout approach to graph transformation. The inclusion of a given architecture in a style is determined (instead of a graph grammar) by typed graphs that define the ways that architectures are allowed to be constructed. This is done by equipping every architecture instance with a morphism to the corresponding type graph for the style.

The work in [Wermelinger, M., 1999; Wermelinger, M. and Fiadeiro, J., 2002], presents an alternative approach for software architecture reconfiguration using graph transformation, but we considered that in relation to our work it is at a different level. Our work is more concrete, in the sense that graphs are specific configurations of the architecture and reconfigurations are applied (and specified) over the explicit component network (in contrast to the morphisms among designs and the double-pushout rules). Also, the work in [Wermelinger, M., 1999; Wermelinger, M. and Fiadeiro, J., 2002] is limited with respect to the self organising approach presented in this thesis and the possible treatment of open systems. These is due to the fact that it requires context sensitive rules not allowing the possibility of unbound synchronizations (i.e. unbound number of participants in a reconfiguration) and the movement of components along the complete configuration graph. The use of typed graphs is a good typing solution for style description assuring consistency of the style but is limited for the specification of complex reconfigurations including the specification of hybrid styles (i.e. combination of different kind of styles) and the possibility of constructing consistent reconfigurations

from smaller inconsistent steps (see Part VI). In [Taentzer, G. *et al.*, 1998] distributed graph transformation for dynamic change management is also applied. This approach also handle reconfiguration via transformation rules but they are based on general graph rewriting rules and thus assume a global, centralized control driving reconfigurations.

Even if in this section we are commenting on related work using graph transformation for software architectures, based on the already stated relevance of π-calculus as a foundational calculi for mobility and its relation with our work, we have to mention the language called Darwin [Magee, J. *et al.*, 1995]. Darwin is an ADL specially design for distributed systems with its semantics expressed in the π-calculus [Magee, J. and Kramer, J., 1996a]. Darwin supports only constrained dynamic manipulation of architectures where the initial architecture may depend on some parameters and during runtime components may be replicated and deleted. Darwin has a supporting tool that allows graphical and textual representations. In spite of this and considering that we are presenting a formal model and not a specific ADL, it is clear that languages like Darwin could benefit from our proposal of a calculus that directly supports a graphical approach, concurrency and increased expressive power for dynamic reconfigurations.

Finally, we consider that it is important to mention the work (in other areas than software architecture) that have been derived from our research. In [König, B. and Montanari, U., 2001], the authors present a bisimilarity for synchronized graph rewriting with name mobility (only for Hoare synchronization), based on the work of [Hirsch, D. *et al.*, 2000], proving it to be a congruence. Also they introduce a so-called *format* which is a syntactic condition on productions ensuring that bisimilarity is a congruence. This last result is original not only for graph rewriting, but also for mobility in general. Also, triggered by [Hirsch, D. and Montanari, U., 2001b], we can mention the work of [De Nicola, R. *et al.*, 2003] in the area of global computing applications for wide area networks, including the introduction of an extension of the approach to cope with quantitative *Quality of Services* (QoS) requirements. The work in [Ferrari, G.L. *et al.*, 2001] introduces a semantics of ambient calculus based on synchronize rewriting with Milner synchronization; and the work in [Lanese, I. and Montanari, U., 2002] introduces a translation for synchronize rewriting with Hoare synchronization to logic programming.

1.5 Thesis Organization

In this part of the thesis we have introduced the notions of software architectures and software architecture styles. Also, we have presented the motivation of our approach for using graph transformations for modeling systems and their reconfigurations, comment on related work and summarize the main results of the thesis. It is in the rest of this thesis that we work out these ideas.

In Part II we present the basic definitions for HR systems and π-calculus.

In Part III we give an introduction to our approach without much formal details (i.e. only the graphical side), with the goal of introducing the reader to the idea of using graph transformation models to represent software architecture styles. This part is intended as a intuitive first step to the next parts of the thesis. In Chapter 3 we introduce how to represent architecture static configurations using hypergraphs and we present a case study from the telemedicine area that will be used to exemplified the new

notions. The use of graph grammars for describing styles is presented in Chapter 4. In this part we use SHR systems without mobility, separating productions in three sets: the *Style Static Configuration Set*, the *Communication Pattern Set* and the *Dynamic Reconfiguration Set*. The level of reconfiguration that can be achieved here is limited to simple creation and removal of components.

Part IV presents the formalization of SHR with the addition of *name mobility*. In Chapter 5 we formalize the notion of hypergraphs as well formed *syntactic judgements* generated from a set of axioms and inference rules. In Chapter 6 we also formalize SHR with name mobility as syntactic judgements generated from a set of axioms and inference rules, and present the inference systems for *Hoare* and *Milner* synchronization algebras.

In Part V, with the goal of studying the expressive power of the formalism introduced in Part IV, we give in Chapter 7 a translation for π-calculus using SHR Systems with Milner synchronization. We present a translation function and state the correspondence theorems.

Part VI presents an alternative approach for specifying reconfigurations (called *transformations*). The method warranties that if the transformation can be specified, then its application over system instances will be consistent with respect to the expected architecture style configuration. In Chapter 8 we show how to specify reconfigurations by *transformation rules* over grammar derivations, and in Chapter 9 we give a first idea of how this approach can be used by a designer that may want to specify a reconfiguration in a more constructive (and maybe more intuitive) way with intermediate steps that may not end in valid configurations of the style. In relation with this last topic, [Medvidovic, N. and Taylor, R., 1997] and [Hofmeister, C. *et al.*, 1999a] comment about the necessity of ADL's tolerance and/or support for incomplete architectural descriptions, how common they are during design and the advantages of allowing incomplete descriptions. However [Medvidovic, N. and Taylor, R., 1997], also mentioned that most existing ADLs and their supporting toolsets have been built around the notion that precisely these kinds of situations must be prevented.

Finally, in Part VII we present our conclusions and ideas for future work.

Part II

Background

Chapter 2

Basic Notions

2.1 Hypergraphs

A *hyperedge*, or simply an edge, is an atomic item with a label (from a ranked alphabet $LE = \{LE_n\}_{n=0,1,...}$) and with as many (ordered) tentacles as the rank of its label. A set of *nodes* together with a set of such edges form a *hypergraph* (or simply a graph) if each edge is connected, by its tentacles, to its *attachment* nodes. Similarly as in [Drewes, F. *et al.*, 1997], a graph is equipped with a set of external nodes identified by distinct names. The difference with the definition in [Drewes, F. *et al.*, 1997] is that instead of using names, they use a sequence of external nodes. External nodes can be seen as the connecting points of a graph with its environment (i.e. the context). Graphs are considered in this paper up to isomorphism.

DEFINITION 2.1. [Hypergraphs] Let $\mathcal{N}$ be a fixed infinite set of names and LE a ranked alphabet of labels. An *edge-labelled hypergraph*, or simply a graph, is defined as a tuple $G = \langle N, E, att, ext, lab_{LE}, lab_{LN} \rangle$ where,

1. N is a set of nodes.
2. E is a set of edges.
3. $att : E \mapsto N^*$ is the connection function (each edge can be connected to a list of nodes).
4. $ext \subseteq N$ is a set of external nodes.
5. $lab_{LE} : E \mapsto LE$ is the labeling function of edges, where $rank(lab_{LE}(e)) = |att(e)|$ for all $e \in E$.
6. $lab_{LN} : ext \mapsto \mathcal{N}$ is the labeling injective function of external nodes.

2.2 Hyperedge Replacement (HR) Systems

A *HR production* rewrites a single edge into an arbitrary graph. Productions will be written as $L \rightarrow R$. A production $p = (L \rightarrow R)$ can be applied to a graph G yielding H ($G \Rightarrow_p H$) if there is an occurrence of an edge labelled by L in G. A result of applying p to G is a graph H which is obtained from G by removing an edge with label L, and embedding a fresh copy of R in G by coalescing the external nodes of R with the corresponding attachment nodes of the replaced hyperedge. This notion of edge replacement yields the basic steps in the derivation process of an HR grammar.

DEFINITION 2.2. [HR Production] Given a set of external nodes *ext*, a HR production p is a pair $\langle L, R, ext \rangle$ (noted as $L \rightarrow R$), where:

1. L is a label of hyperedge.
2. R is a graph.
3. The attachment nodes of the edge with label L and the external nodes of graph R are exactly those in *ext*.
4. The nodes in *ext* cannot be deleted by p.

DEFINITION 2.3. [HR Grammar] A HR Grammar is a pair $HRG = \langle G_0, P \rangle$, where:

1. G_0 is a graph.
2. P is a set of HR productions.
3. The rewriting step resulting from the application of a production p_i to graph G_{i-1} ($G_{i-1} \Rightarrow_{pi} G_i$) is a *direct derivation step*. A HR *derivation* is a finite sequence of direct derivation steps of the form $G_0 \Rightarrow_{p1} G_1 \Rightarrow_{p2} \ldots \Rightarrow_{pn} G_n = H$, where $p_1, \ldots, p_n$ are in P.

2.3 Synchronized HR (SHR) Systems

DEFINITION 2.4. [SHR Production] Given a set *Act* of actions, a SHR production p is a tuple $\langle L, R, ext, f \rangle$, where:

1. $\langle L, R, ext \rangle$ is a HR production.
2. $f : ext \rightarrow Act^*$, assign tuples of actions to the attachment nodes of an edge labelled with L.

A *SHR grammar* consists of an initial graph and a set of SHR productions. A *SHR derivation* is obtained by starting with the initial graph and applying a sequence of rewriting rules, where each rewriting rule is obtained by synchronizing possibly several SHR productions. How many productions will synchronize depends on the *synchronization mechanism*. Note that HR productions are special cases of SHR productions where $Act = \emptyset$.

DEFINITION 2.5. [SHR Grammar] Given a set *Act* of actions, a SHR Grammar is a pair $SHRG = \langle G_0, P \rangle$, where:

1. G_0 is a graph.
2. P is a set of SHR productions.
3. The rewriting step resulting from the application of a rule P_i to graph G_{i-1} ($G_{i-1} \Rightarrow_{pi} G_i$) is a *direct derivation step.* A SHR *derivation* is a finite sequence of direct derivation steps of the form $G_0 \Rightarrow_{P1} G_1 \Rightarrow_{P2} \ldots \Rightarrow_{Pn} G_n = H$, where $P_1, \ldots, P_n$ are sets of SHR productions of P.

2.4 The π-Calculus

The π-calculus [Milner, R., 1999; Sangiorgi, D. and Walker, D., 2001] is a name passing process algebra. Many different versions of the π-calculus have appeared in the literature. The π-calculus we present here is synchronous, *monadic*, with guarded recursion and guarded sum.

DEFINITION 2.6. [π-Calculus Syntax] Let $\mathcal{N}$ be the countable set of names. The syntax of π-calculus *agent terms*, ranged over by $P, Q, \ldots$, are defined by the grammar:

$$P ::= nil \mid \sum_{i=1}^{n} \pi_i.P_i \mid P|P \mid \nu x.P \mid rec\, X.P$$

In order we have, inaction, guarded sum, parallel composition, restriction and recursion.

Prefixes, ranged over by π, are defined as:

$$\pi ::= \bar{x}y \mid x(y)$$

They correspond to the output action and input action. The occurrences of y in $x(y).P$ and $\nu y.P$ are bound; *free names* of agent P are defined as usual and we denote them with $\mathrm{fn}(P)$. Also, we denote with $\mathrm{n}(P)$ and $\mathrm{n}(\pi)$ the sets of (free and bound) names of agent P and prefix π, respectively. The τ symbol will be considered as an action given that we are interested in the synchronization of components but not in their internal computations.

Also, we require that any free occurrence of X in $rec\, X.P$ must be in the scope of a prefix (guarded recursion).

If σ is a name substitution, we denote with $P\sigma$ the agent P whose free names have been replaced according to substitution σ, in a capture-free way.

DEFINITION 2.7. [π-Calculus Structural Congruence] We define π-calculus agents up to a *structural congruence* $\equiv$; it is the smallest congruence that satisfies axioms in table 2.1.

(alpha) $P \equiv Q$ if P and Q are alpha equivalent with respect to bounded names

(par) $P|nil \equiv P \qquad P|Q \equiv Q|P \qquad P|(Q|R) \equiv (P|Q)|R$

(res) $\nu x.\, nil \equiv nil \qquad \nu x.\nu y.\, P \equiv \nu y.\nu x.\, P$

$\nu x.\,(P|Q) \equiv P|\nu x.\, Q \quad$ if $x \notin \mathrm{fn}(P)$

(rec) $rec\, X.\, P \equiv P\,[rec\, X.\, P/X]$

Table 2.1: π-Calculus Structural Axioms

We remark that $P \equiv Q$ implies $P\sigma \equiv Q\sigma$ and $\mathrm{fn}(P) = \mathrm{fn}(Q)$: so, it is possible to define the effect of a substitution and to define the free names also for agents up to structural equivalence.

DEFINITION 2.8. [π-Calculus Operational Semantics]
The operational semantics of the π-calculus is defined via labelled transitions $P \xrightarrow{\alpha} P'$, where P is the starting agent, P' is the target one and α is an action. There are many versions for the operational semantics of π-calculus but we will not describe them in this thesis. We refer to [Milner, R., 1999; Sangiorgi, D. and Walker, D., 2001] for further explanations of the different semantics.

For the translation presented in Chapter 7 we will use the *late operational semantics* of π-calculus and its corresponding transition system.

The transitions for the *operational semantics* are defined by the rules of Table 2.2. The *actions* an agent can perform are defined by the following syntax:

$$\alpha ::= \tau \mid x(z) \mid \bar{x}y \mid \bar{x}(z)$$

and are called respectively *synchronization*, *bound input*, *free output* and *bound output* actions; x and y are free names of α ($\mathrm{fn}(\alpha)$), whereas z is a bound name ($\mathrm{bn}(\alpha)$); moreover $\mathrm{n}(\alpha) = \mathrm{fn}(\alpha) \cup \mathrm{bn}(\alpha)$.

(Sum) $\sum_{i=1}^{n} \alpha_i.P_i \xrightarrow{\alpha_i} P_i$ with $\alpha_i = x(y), \bar{x}y$

(Par) $\dfrac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q}$ if $\mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \emptyset$ *(Com)* $\dfrac{P \xrightarrow{\bar{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{P|Q \xrightarrow{\tau} P'|Q'\{y/z\}}$

(Close) $\dfrac{P \xrightarrow{\bar{x}(y)} P' \quad Q \xrightarrow{x(y)} Q'}{P|Q \xrightarrow{\tau} \nu y.\,(P'|Q')}$ *(Open)* $\dfrac{P \xrightarrow{\bar{x}y} P'}{\nu y.\, P \xrightarrow{\bar{x}(y)} P'}$ if $x \neq y$

(Res) $\dfrac{P \xrightarrow{\alpha} P'}{\nu x.\, P \xrightarrow{\alpha} \nu x.\, P'}$ if $x \notin \mathrm{n}(\alpha)$ *(Cong)* $\dfrac{P \equiv P' \quad P \xrightarrow{\alpha} Q \quad Q \equiv Q'}{P' \xrightarrow{\alpha} Q'}$

Table 2.2: π-Calculus Operational Semantics

The prefix $x(z)$ for bound input means *input some name along link named x and call it y.* A free output $\bar{x}y$ means *output the name y along the link named x.* A bound output $\bar{x}(z)$ is not available at the syntactic level and corresponds to the emission of a private name of an agent to the environment, i.e., z is a name that was previously restricted: in this way, the channel becomes public and can be used for further communications between the agent and the environment. Rule *(Open)* allows to share with the environment a node that was originally bounded. This rule may be used for sharing a port of communication that was local among some processes and that now they want to allow others to communicate with them by that port. and together with rule *(Close)* causes an extrusion. Extrusion allows to export and share bounded nodes. But once synchronization is completed, it hides away those private names that were synchronized meaning that the names are still bound, but their scope has grown.

DEFINITION 2.9. [Agents]

- **Agen t Term:** An agent term is any term generated by the grammar in Definition 2.6.
- **Agen t:** An agent is an agent term that represents a class of terms up to structural equivalence.
- **Se quential Agent:** Some of the agent terms equivalent to it have guarded sum as top operator.

The following definition for standard decomposition of sequential agents will be needed for the translation of π-calculus to SHR systems in chapter 7.

DEFINITION 2.10. [Standard Decomposition] The *Standard Decomposition* of a sequential agent P is defined as:

$$P = \sigma_P \widehat{P}$$

where σ_P and $\widehat{P}$ are the standard substitution and standard agent of P, respectively, and with $\sigma P = (\sigma\sigma_P)\widehat{P}$ also standard, for any name permutation σ.
The standard agent $\widehat{P}$ is obtained by:

1. Find the first of the equivalent terms of P with respect to structural equivalence.
2. Distinguish and order *all free variable occurrences.* We assume $\mathrm{fn}(\widehat{P}) = \{v_1, \ldots, v_n\}$.

We assume that there is a procedure that, by a given order over the equivalence class of P, returns the corresponding standard substitution and standard agent for P. Also, we have that $\forall\gamma.(\sigma_{\gamma P} = \gamma\sigma_P) \wedge (\widehat{\gamma P} = \widehat{P})$ with γ any substitution.

For example, assuming that sequential process $P = \bar{x}y.z(w).nil + x(y).nil$ is the first (by a given order) of its class, we have:

$$\sigma_P = [x/v_1, y/v_2, z/v_3, x/v_4] \text{ and } \widehat{P} = \bar{v}_1 v_2.v_3(w).nil + v_4(y)$$

Part III
Software Architecture Styles and Grammars

In Part III we give an introduction to our approach without much formal details (i.e. only the graphical side), with the goal of introducing the reader to the idea of using graph transformation models to represent software architecture styles. This part is intended as a intuitive first step to the next parts of the thesis.

In this part of the thesis we present to the reader the idea of using graph transformation models to represent software architecture styles. The presentation in this part is intended as an introduction to our approach without much formal details (i.e. only the graphical side), with the goal of introducing the reader to the idea of using graph transformation models to represent software architecture styles. This part is intended as a intuitive first step to the following parts of the thesis.

In Chapter 3 we introduce how to represent architecture static configurations using hypergraphs and we present a case study from the telemedicine area that will be used to exemplified the new notions. The use of graph grammars for describing styles is presented in Chapter 4. In this chapter we use SHR systems without mobility. The approach separates productions in three sets: the *Style Static Configuration Set*, the *Communication Pattern Set* and the *Dynamic Reconfiguration Set*. The level of reconfiguration that can be achieved here is limited to simple creation and removal of components. As we already mentioned in the introduction of this thesis we are interested in a self organising approach and consequently focused on graph transformation models supporting this idea, i.e. the use of context-free grammars.

Chapter 3

Software Architectures and Graphs

One of the basic steps in the construction of a software system is the identification of its software architecture ([Garlan, D. and Shaw, M., 1996; Perry, D. and Wolf, A., 1992]), that means that the different participants of the project must agree on a system configuration of its components and the definition of the interactions among them.

In this chapter we will focus only on the representation of the static configuration of software architectures. We present graphs as a formal model for describing software architectures, more specifically, graphs are proposed for describing the static configurations of systems. At the end of this chapter we also comment about related work using graphs based approaches to support multiple view architectures.

3.1 Case Study: Remote Medical Care System

This section presents a case study from the telemedicine area, which will be used in some of the different chapters of this thesis. This case study is motivated by a real system developed as part of a project carried out by University of L'Aquila and Parco Scientifico e Tecnologico d'Abruzzo, a regional consortium of public and private research institutions and manufacturing industries in Italy. Also, it was proposed as a working case study for the Tenth International Workshop on Software Specification and Design (IWSSD-10) [Inverardi, P. and Muccini, H., 2000].

The current trend in healthcare is to transition patients from hospital care to home care as quickly as feasible. The Teleservices and Remote Medical Care System (TRMCS) is intended to provide and guarantee assistance services to at home or mobile users. This type of patient does not need continuous assistance but may need prioritized assistance when urgencies happen, in which case the patient would call a help center for assistance. The system must handle help request to a help center from patients with a medical emergency. Also, patients may have internet-based medical monitors that give continuous readouts. A help center may be contracted to read these monitors over the net and raise alerts when dangerous values are detected.

The case study can be examined from many perspectives like requirements, safety critical aspects, security, design and user interface. Some of the requirements of the system related to dynamic changes and reconfigurations are as follow.:

- Be open to new service installations.
- Handle users that are geographically distributed and heterogeneously connected, offering homogeneous service costs.
- Handle dynamic changes of users and their locations.
- Support mobile users.

Others requirements not directly related with dynamics but which can affect the possible solutions include:

- Guarantee continuous service of the system.
- Guarantee the delivery of help service in response to a help request in a specific critical time range.
- Handle several help request in parallel that compete for service by overlapping in time and space.
- Support conflict resolution according to resolution policies that minimize user damage.

For clarity, the operations of the different components have been simplified. The three types of units operate as follows.

User sends either alarm (i.e. help requests) or check signals (i.e. control messages) on the user subsystem state or on the user health state, respectively.

Router accepts signals (control or alarm) from the Users. It forwards the alarm requests upward to the Server and checks the behavior of the user subsystem through the control messages.

Server receives alarms from Routers and dispatches the help requests.

The software architecture of the system follows a hierarchical style. There is only one server in the system. A variable number of routers are connected to the server and a variable number of users are connected to each router.

3.2 Graphical Notations: Semiformal and Formal Languages

To fulfill the task of obtaining the software architecture of a given system, usually, practitioners rely on box-and-line drawings with the intention to communicate, among a project team, the intuition of the system structure under development. Many of the graphical notations that practitioners have used or are using do not have a formal semantics that allows a clear understanding for the meaning of these kind of diagrams. An example of these informal notations that is being used widely, almost imposed in many organizations as a standard, is UML [Booch, G. *et al.*, 1999]. In spite of being a notation that comes from the object-oriented universe, and not being initially thought with that goal, it is being used for system architecture description. The problem (among others) with these informal notations is that it is very difficult to reconcile the different interpretations that can be given to the same diagram by different participants of a project.

On the other side, we have the development of *Architecture Description Languages (ADLs)*, which are languages specially developed to cope with the architectural aspects of a system [Medvidovic, N. and Taylor, R., 1997]. In general, most of these formalisms support graphical descriptions with defined semantics that allow some type of analysis, although many of them are restricted to specific domains or the use of one type of style. It is in the direction of understanding which is the best way of describing software architectures that many researchers are investigating the real suitability (or not) of using languages like UML for architectural description and their relationship with ADLs [Garlan, D. and Kompanek, A., 2000; Hofmeister, C. *et al.*, 1999a; Hofmeister, C. *et al.*, 1999b; Medvidovic, N. and Rosemblum, D., 1999].

But above all the mentioned approaches we can see a common denominator. They use graphical notations for supporting communication of ideas among people. It is in this context, that we see graphs as the nearest (and most general) formal model to the box-and-line drawing representation. And with this in mind, one of our goals is to introduce the reader to a common formal ground among the many languages, notations and tools that he/she may find, trying to identify which one is the best suited for his/her needs, and also if it is necessary, to be able to develop them formally.

3.3 Graphs for SA Static Configurations

A graph represents a state of the static configuration of the architecture. The configuration evolution from one state to the other will be modeled as graph rewriting from one graph to another. At this point, you should imagine that there are various possibilities of representations for architectures as graphs, depending on how edges and nodes are interpreted with respect to components and communication links or ports. In this thesis, we will follow the representation first introduced for software architectures in [Hirsch, D. *et al.*, 1998] and [Hirsch, D. *et al.*, 1999], where a software architecture structural topology is described as a hypergraph where hyperedges are components and nodes are communication ports. Hyperedges sharing a node mean that there is a communication link among

the components. Names and other attributes of components can be seen as labels of the hypergraph. Then we can give a definition of a hyperedge (or simply an edge) as an atomic item with a label (from a ranked alphabet $LE = \{LE_n\}\{n = 0, 1, \ldots\}$) and with as many (ordered) tentacles as the rank of its label. A set of nodes together with a set of such edges forms a hypergraph (or simply a graph) if each edge is connected, by its tentacles, to its attachment nodes. Note that the usual notion of graph is a special case of hypergraphs where all hyperedges have rank 2. You can find the formal definitions in Section 2.1 and more information about hypergraphs in [Drewes, F. *et al.*, 1997].

3.4 Example

As an example, you can see in Figure 3.1a. a hypergraph representing an instance of the style used for the TRMCS system. Hyperedges (components) are drawn as boxes with a label for the component name and they are connected together by nodes (communication ports). In this case, we have one **Server** component and two **Routers**, the first one with one **User** and the second with two of them. The communication port **AlarmRSp** connects the **Server** and **Routers** and it is used to forward **User** alarms to the **Server**. The communication port **Signalp** connects a **Router** with its **Users** and it is used to send **User** alarms to the **Router** and to send check signals from the **Router** to a **User**.

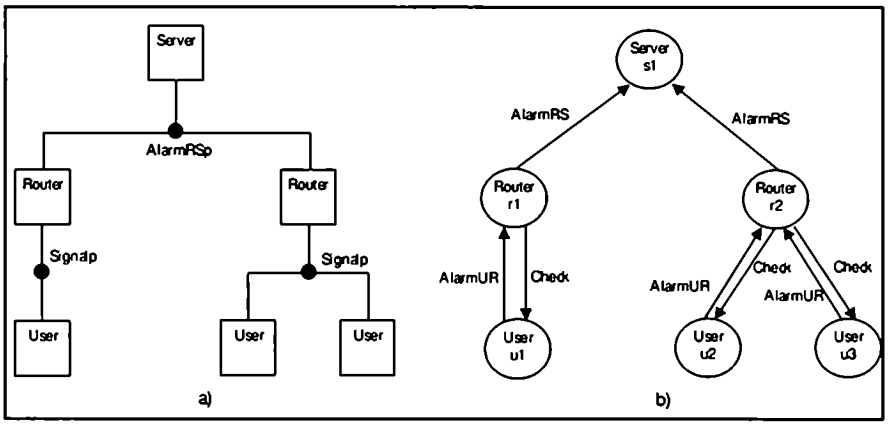

Figure 3.1: Representations of a software architecture configuration as a graph. a) Components as Hyperedges. b) Components as Nodes.

Another possible representation of architectures as graphs and the first work in which (context-free) graph grammars are used to represent styles is [Le Métayer, D., 1998]. Le Metayer presents a dual approach where nodes represent components and edges their communication links. Graphs are used to represent architectural instances where a graph is formally defined as a set of relation tuples of the form $R(e_1, \ldots, e_n)$ where R is a n-ary relation and e_i are entity names (note that this definition also allows the representation of hypergraphs). In [Le Métayer, D., 1998] only binary and unary relations are used,

a binary relation $L(e_1, e_2)$ represents a directed link of name L between e_1 and e_2 (a communication link from component e_1 to component e_2); and a unary relation $L(e)$ characterizes the role of entity e (e is a component L) in the architecture. As an example, Figure 3.1b. shows the same TRMCS instance of Figure 3.1a. with this notation, where component types **Server**, **Router** and **User** are unary relations, and communication links (not ports) **AlarmRS** ,**AlarmUR** and **Check** are directed edges (binary relations). Nodes $s_1, r_1, r_2, u_1, u_2, u_3$ are the components of the system. Again, it

is worth noticing that these two approaches are not the only ones, and that other possible ways of interpreting architecture structures can be applied. For example, the dual ones of these two are options. A second point to comment is about another important notion of software architecture, which are connectors. Connectors represent interactions among components. They may represent simple interactions or can be entities that have to deal with complex communication mechanisms and protocols. In our context of graph representation, connectors can be seen as simple communication ports (as in the hypergraph approach). But also, if they must specify complex interactions they can be treated as first class entities like components (but as a distinguished class) with their corresponding behavior to coordinate interactions (also known as roles [Allen, R. and Garlan, D., 1994]). In this setting, nodes are the connections between components and connectors.

3.5 Multiple Views

Another important topic that is of relevance in the area of Software Design is the notion of view. A view is the representation of a system from a given set of requirements or properties of interest. The goal of using views is to reduce complexity, separation of concerns and help in analysis. Usually, views are represented with different types of diagrams, for example, in UML you have class diagrams, statecharts, deployment diagrams, etc., that are used to describe different aspects of a system. In the area of software architecture the use of views has been proposed and studied too. For example we have [Kruchten, P.B., 1995], where the 4 + 1 View Model is introduced, and [Bass, L. *et al.*, 1998] where the use of views is proposed to analyze architectures. A third work that applies a multiple view approach and relates UML with software architecture is [Hofmeister, C. *et al.*, 1999a].

But one of the main problems with views is how to keep and/or check consistency among views. Another problem is how to be able (if possible and desired) to obtain an integrated view from the basic ones. As far as we know, the existing formal ADLs do not support a multiple view approach, but some work has been done on its formalization and the treatment of the problems mentioned above. We can mention [Fradet, P. *et al.*, 1999] and [Périn, M., 2000], where a graph notation is used to represent different views of a system and diagrams with multiplicity (in the spirit of UML class diagrams) define a class of graphs for each type of view. Multiplicity is used to constraint the number of connections on nodes for graphs in a class. Also, an algorithm is proposed to check the consistency among views and a language of constraints is introduced to express more complex consistency requirements together with a decision procedure to

check if diagrams satisfy constraints expressed in that language.

Another work in this line is [Heckel R., 1998], where an object-oriented approach is introduced to specify different views of a system using open graph transformation systems based on a loose semantics. In [Heckel R., 1998], a view is an incomplete specification of a system focusing on a particular aspect or subsystem, and then a view specifies only what, at least, has to happen on a system's state. Views are obtained from a common reference model and evolve concurrently with corresponding extensions to the original reference model to keep consistency when new dependencies appear among views. At the end, views are compositionally integrated to obtain the overall system specification. For temporal specification and verification of safety and lifeness properties temporal arrow logic [Marx, M. *et al.*, 1996] is used.

Then, following the idea of graphs as a general model for system architectures, and as it was already mentioned that it is used in other works, views can be seen as a family of graphs where each view may have a different interpretation of the meaning of nodes and edges.

In this thesis we will not work with different views leaving this topic for future work (see chapter 10).

Chapter 4

Software Architecture Styles and Graph Grammars

Now that we have introduced graphs as a representation of the static configuration of architecture instances, we want to go to a higher level and talk formally about classes of architectures sharing similar characteristics, i.e. software architecture styles. To achieve this goal, in this chapter we present graph grammars. At the same time, the use of grammars comes with the notion of *graph rewriting* or *graph transformation* which together will let us describe the evolution of a system architecture not only as transformations over system instances, but at the level of style.

4.1 Graph Grammars and Graph Transformations for Styles

The motivation to introduce the use of styles is very important. It allows reuse of architectures by the identification of related structures on supposedly different systems [Perry, D. and Wolf, A., 1992; Garlan, D. and Shaw, M., 1996; Bass, L. *et al.*, 1998]. And also, it will make possible to compare different styles (i.e. classify styles in a taxonomy) being able to choose which is the best one to fulfill the requirements for a system to be implemented. It is clear that the use of informal notations without a defined semantic does not allow a clear notion of what is the meaning of the different diagrams that are used to describe the architecture. As a consequence it is very difficult to talk about styles and reusability.

The description of a software architecture style must contain:

- The structure model of component types and their interactions or connections, i.e. the structural topology.
- The communication pattern, i.e. the interactions among component types.
- The laws governing the dynamic changes in the architecture configuration, i.e. reconfiguration and/or mobility.

A graph grammar characterizes a class of graphs with similar characteristics and is composed of a set of rewriting rules called productions (which are the ones that define the general structure of the graphs of the class). A production rewrites a graph into another graph, deleting some elements (nodes and edges), generating new ones, and preserving others. Also, as we already mentioned, the application of rewriting rules to graphs drives to the notion of graph rewriting or graph transformation, which itself, it in is very useful and another motivation for the approach. Using graph grammars and graph rewriting not only let us specify the construction of the static structure of systems for a given style. Later we will be able to include productions specifying runtime interactions among components, reconfiguration and mobility, allowing their explicit description at the level of style language. In conclusion, a graph gives a simple view of the structure of an architectural instance at a given state, separated from the application of the rewriting rules that show the evolution between states.

The use of graph transformation to specify systems is not new and the amount of work in this area shows its relevance in computer science. As we already mentioned in the introduction (see Section 1.2) and chapter 3, [Le Métayer, D., 1998] has been the first to propose the use of graph grammars for describing software architecture styles. Also, we can mention its application in the areas of concurrency, parallelism and distribution [Ehrig, H. *et al.*, 1999b]; for functional languages, visual and object-oriented languages and software engineering [Ehrig, H. *et al.*, 1999a]; and the proposal of general formal frameworks for system modeling [Engels, G. and Heckel, R., 2000; Mens, T., 2000].

There are many formalisms for rule based graph specification and transformation. In this thesis we will just consider context-free graph grammars. Context-free productions can be specified in terms of node replacement or edge replacement. As we introduced in Section 1.2, thinking about the new generation of systems and their requirements for distribution, heterogeneity and mobility, is that we are interested in modeling *self organising* systems [Magee, J. and Kramer, J., 1996b], i.e., management is distributed among components without a central coordinator. Then, continuing with the approach proposed in [Hirsch, D. *et al.*, 1998] and [Hirsch, D. *et al.*, 1999], we describe a software architecture style using context-free *Hyperedge Replacement* (HR) grammars [Drewes, F. *et al.*, 1997] and *Synchronized* HR (SHR) grammars to model communication and coordination among components (SHR was first used to represent distributed systems [Degano, P. and Montanari, U., 1987; Montanari, U. *et al.*, 1999]).

The productions of a grammar are grouped in three sets:

- St atic Configuration Set: This set of productions represents the construction of all possible initial configurations of the class of architectures modeled by the style, i.e. the static structure of architectures.

- Dyna mic Reconfiguration Set: This set contains the productions for the dynamic evolution of the style configuration, this means create and remove components. In this case SHR productions are used to coordinate the reconfiguration of architectures.

- Co mmunication Pattern Set: This set contains the productions that model the communication evolution for each type of component. These productions are SHR

productions that during rewriting will synchronized for the evolution of the system. Productions in this set cannot change the topological structure of the graph.

4.2 Style Static Configuration

A HR production (or simply a production) rewrites a single (hyper)edge into an arbitrary graph. Productions will be written as $L \rightarrow R$. A production $p : L \rightarrow R$ can be applied to a graph G yielding H ($G \Rightarrow_p H$) if there is an occurrence of an edge labelled by L in G. A result of applying p to G is a graph H which is obtained from G by removing an edge with label L, and embedding a fresh copy of R in G by coalescing the corresponding attachment nodes. In the standard definition for HR systems, no nodes are deleted. This notion of edge replacement yields the basic steps in the derivation process of an HR grammar. See the formal definitions in Section 2.2.

In Figure 4.1 we have a graph H containing an edge with label L_1an d another with label L_2. Then, we have two productions, p_1t hat rewrites an edge with label $L_1$with a graph R_1 and p_2 that rewrites an edge with label L_2 with a graph R_2. The right hand side of the figure shows the result of applying p_1 and p_2 to the edges in graph H. The result of rewriting graph H is a new graph where the two edges with labels L_1 and L_2 were replaced by graphs R_1 and R_2. Note that R_1 and R_2 can be any graphs, containing new edges and nodes, but that L_1 and L_2 attachment nodes cannot be deleted (maybe other edges in graph $H - B$ are attached to that nodes too). Here, nothing is said about the order in which productions are applied, and this implies some properties of HR systems (and other rule-based formalisms).

These properties are sequentialization and parallelization, confluence and associativity. Sequentialization and parallelization say that it does not matter to replace edges of a hypergraph one after another, or simultaneously. Confluence says that edges of a hypergraph can be replaced in any order without affecting the result. Finally, associativity says that if an edge is replaced and afterwards an edge of the new part is replaced again by a new hypergraph, the same result is obtained by first replacing the last edge and then replace the first with the result. These properties show the suitability of hyperedge replacement for specifying distributed and concurrent systems.

The application of a production p to a graph G yielding H ($G \Rightarrow_p H$) is called a direct derivation step. Given a set of productions P, a sequence of direct derivation steps starting from graph G_0, of the form $G_0 \Rightarrow_{p1} G_1 \Rightarrow_{p2} \ldots \Rightarrow_{pn} G_n = H$, where $p_1, \ldots, p_n$ are in P, is called a derivation of length n. From the above properties it can be seen that there can be more than one possible derivation from G_0 to G_n.

A HR grammar consists of an initial graph G_0 and a set of productions P. A derivation in the grammar is obtained by starting with the initial graph G_0 and applying a sequence of rewriting rules, each obtained by productions in P.

For the definition of the static configuration set of a style, a set of productions is specified which represents the construction of all possible initial configurations of the class of architectures modeled by the style. After obtaining an initial configuration for a desired system instance, the evolution of the system begins.

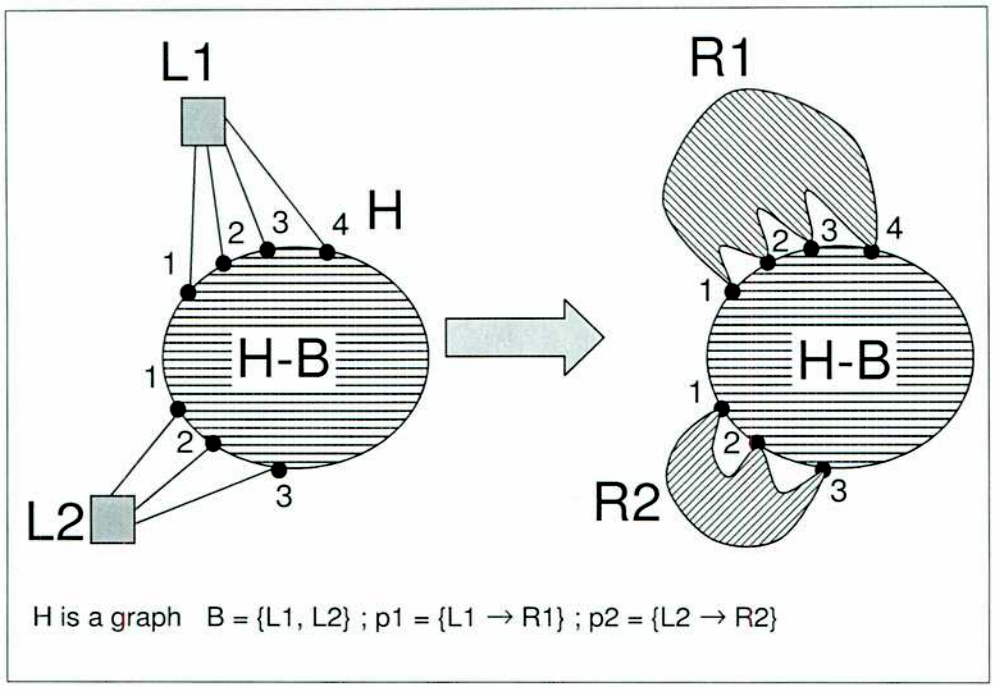

Figure 4.1: Applications of context-free graph productions.

4.2.1 Example

Using the TRMCS case study, Figure 4.2a. shows the initial graph and the static configuration set of productions for the TRMCS hierarchical style. Recalling, components are edges (boxes) and communication ports are nodes. Edge labels have two parts, one is the component name and the other is the status of the component that represents its different states during evolution. For the static productions the status is always **Init**, meaning that they are in a kind of initialization phase. After the initialization phase, components change to an **Idle** status applying the *Idle Productions* (not shown) which means that the architecture instance can start using its dynamic evolution productions from the other sets.

The initial graph corresponds to the **Server** type component. Production *Create Router (CR)* adds a **Router** type component attached to the **Server** (using port **AlarmRSp**) and creates a new port **Signalp** (new nodes are depicted as blank nodes in productions). Production *Create User (CU)* adds a **User** type component attached to one of the **Routers** (using port **Signalp**). These two productions together with the initial graph generate all the static configurations of the style. Figure 4.2b. shows one possible derivation to generate the graph configuration in Figure 3.1a. Double lined components indicate edges over which productions are applied.

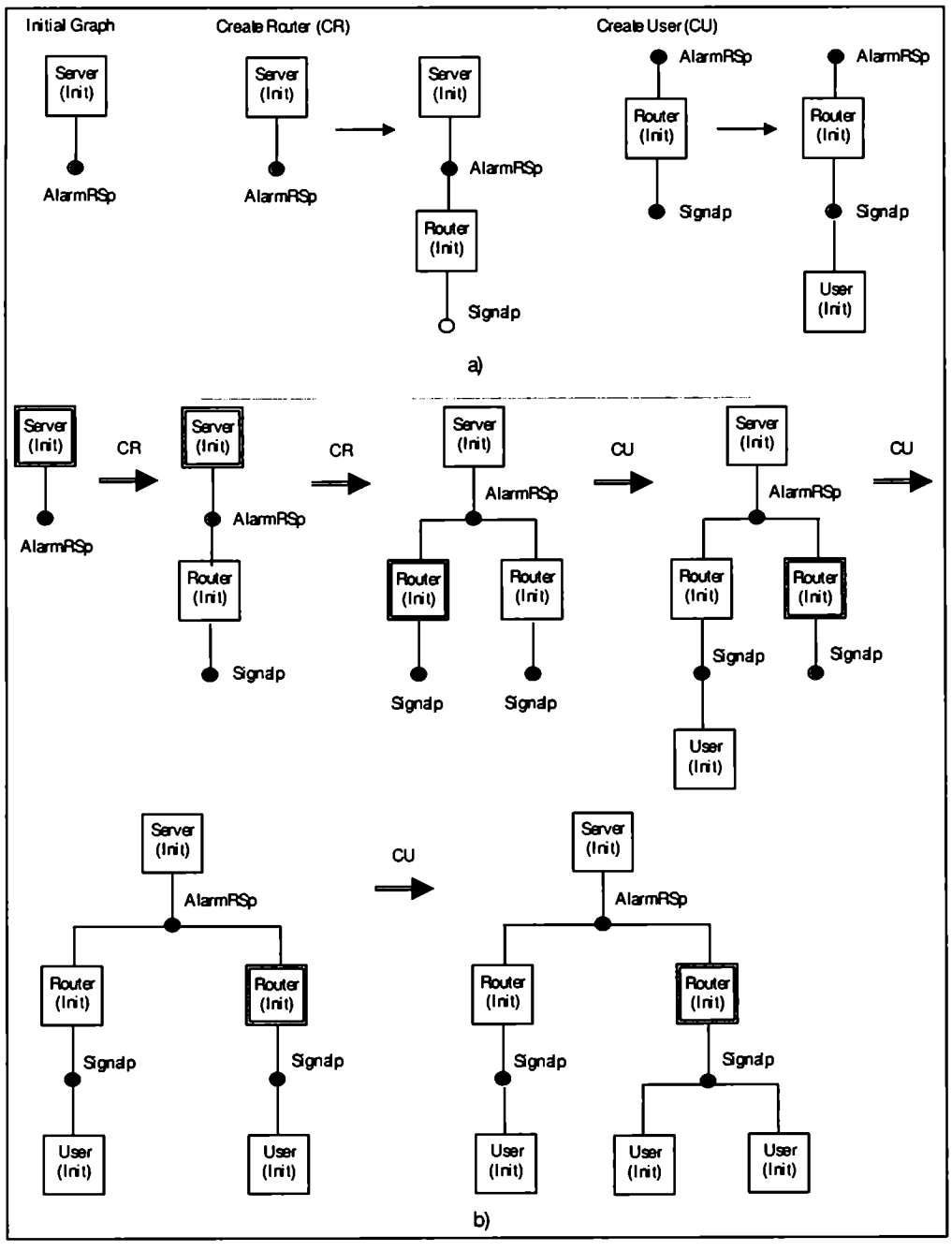

Figure 4.2: a) Initial Graph and Static Configuration Set. b) An Architecture Derivation.

4.3 Style Dynamic Reconfiguration

To model coordination of the evolution of a software architecture configuration, we need to choose a way of selecting which components will evolve and communicate. Using

graph rewriting we may need to specify that more than one production has to be applied at the same time (i.e. various components need to synchronize). For this, we combine graph rewriting with synchronizing conditions obtaining SHR systems. We specify synchronized productions by adding conditions (or generally actions) on nodes which allow to coordinate several rewritings (called *synchronized rewriting*), thus determining how components interact and are reconfigured.

For this, assuming to have an alphabet of actions Act, we associate some labels (actions) to the (attachment) nodes in the left member of productions. In this way, each rewrite of an edge must match actions with its adjacent edges and they have to move as well. This technique was already applied in [Degano, P. and Montanari, U., 1987; Montanari, U. *et al.*, 1999] to represent distributed systems.

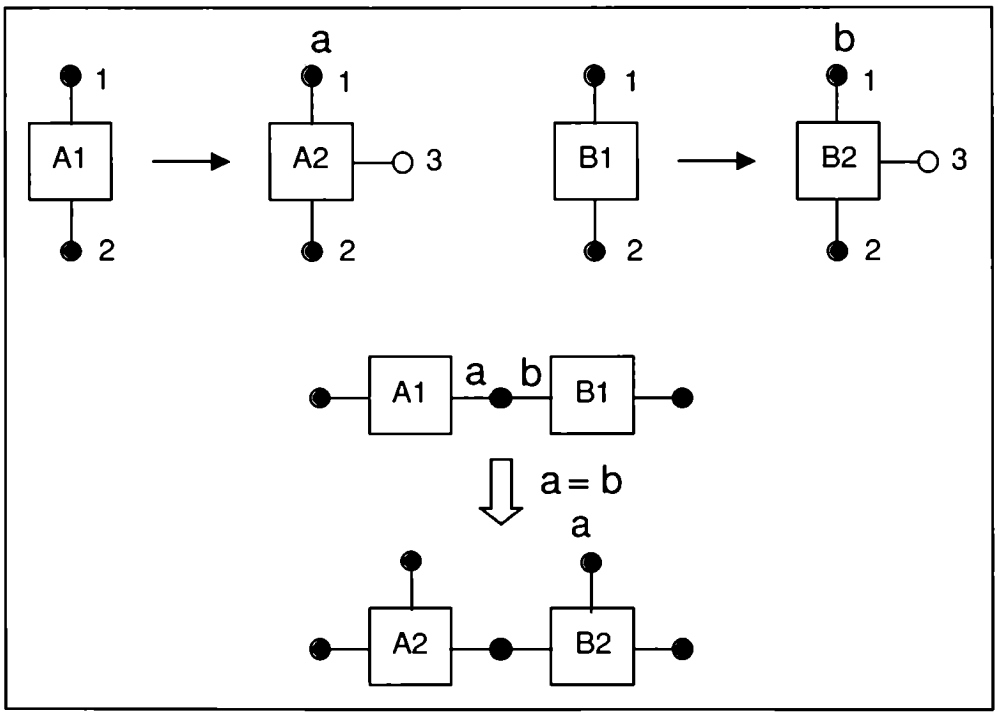

Figure 4.3: Synchronized Rewriting.

For example, consider the two productions in Figure 4.3. The first production rewrites an edge with label A_1 into an edge A_2 with rank three, creating a new node. The other production does the same for edges with labels B_1 and B_2. Now, lets take, like in the figure, a graph that contains edges A_1 and B_1 which share one node, such that no other edge in the graph is attached to that node. Each of the productions have a action on that node (a and b), which means that *the production can be applied only if actions on nodes are satisfied.* If $a \neq b$, then the edges cannot rewrite together (using that productions). If $a = b$, then they can move, via each of its corresponding production. This is called a synchronized rewriting step. This type of synchronization mechanism,

called *Hoare synchronization*, is the basic one, but other types of coordination policies can be used.

The problem of finding the set of productions to use in a synchronized rewriting step is called *the rule-matching problem* [Degano, P. and Montanari, U., 1987]. The solution of the rule-matching problem is implemented considering it as a finite domain constraint problem [Mackworth, A.K., 1998]. An analysis of some techniques to solve this problem in a distributed and efficient way can be found in [Montanari, U. and Rossi, F., 1999; Montanari, U. *et al.*, 1999]. The description of these techniques is out of the scope of this thesis.

Now that the notions of production, graph rewriting and synchronized rewriting have been introduced, we can easily define SHR grammars (or simply a grammars). A grammar consists of an initial graph G_0 and a set of SHR productions P. A SHR derivation is obtained by starting with the initial graph G_0 and applying a sequence of rewriting rules, each obtained by synchronizing possibly several SHR productions (see Section 2.3). How many productions will synchronize depends on the *synchronization mechanism*. It is worth noticing that HR productions are special cases of SHR productions where $Act = \emptyset$, which allow to include cases where components may not need to synchronize.

The dynamic reconfiguration set of a style defines a set of productions which represent allowed structural changes that architectural instances can suffer during their evolution, like for example the creation of a new component or the removal of an existing one. This set contains SHR productions to model the coordination of a reconfiguration where many components are involved. SHR systems support our intention to propose a self-organising approach ([Magee, J. and Kramer, J., 1996b]) where there is no central coordinator controlling the system evolution and each component defines its own evolution. The application of these productions (and the ones for communication) is done after an initial configuration of a system is achieved using the static configuration set.

4.3.1 Example

Figure 4.4 shows some possible reconfiguration productions for the case study. In this example we use the Hoare synchronization mechanism for productions, so actions on a port must be matched by all components sharing it. Actions are within brackets allowing synchronization on tuples. All productions are applied in the **Idle** status specifying that reconfiguration can only occur when there are no communication interactions among the involved components, which otherwise can provoke erroneous states [Kramer, J. and Magee, J., 1990].

The first two productions have no actions and represent the creation of a new **User** and the removal of an existing one. Of course, this is a design choice that lets **Users** appear and disappear at any moment, and it can be specified in other ways. In the case of a removal of a **Router** it is not so simple, because a **Router** can have a number of **Users** attached to him. The *Remove Router* production is an example of the use of coordinated rewriting to model negative actions. The action $< noUSER >$ is imposed on the **signalp** port where the **Router** and its **Users** are connected. A action in this port means that for this production to be applied it must coordinate with all other

components connected to that port (i. e. the **Users**). So, if all neighbors agree on the action, then everybody can rewrite. But in this case, the only one with this action is the **Router** and it cannot leave the system while **Users** are attached to it. When all **Users** connected to the **Router** leave the system then the production with action $< noUSER >$ is satisfied and then it can be applied to the **Router**. The production can be applied because there are no neighbors, so the **Router** is the only one that has to agree on the action.

The rest of the productions represent a scenario where there is a forced removal of a **Router** and its **Users**. To avoid ending in an inconsistent state they have to synchronize ($< remove >$) to be removed at the same time. It is worth noticing that using SHR productions allow the specification of reconfigurations with an unbound number on participants and also that computation is not stopped while a system is reconfigured.

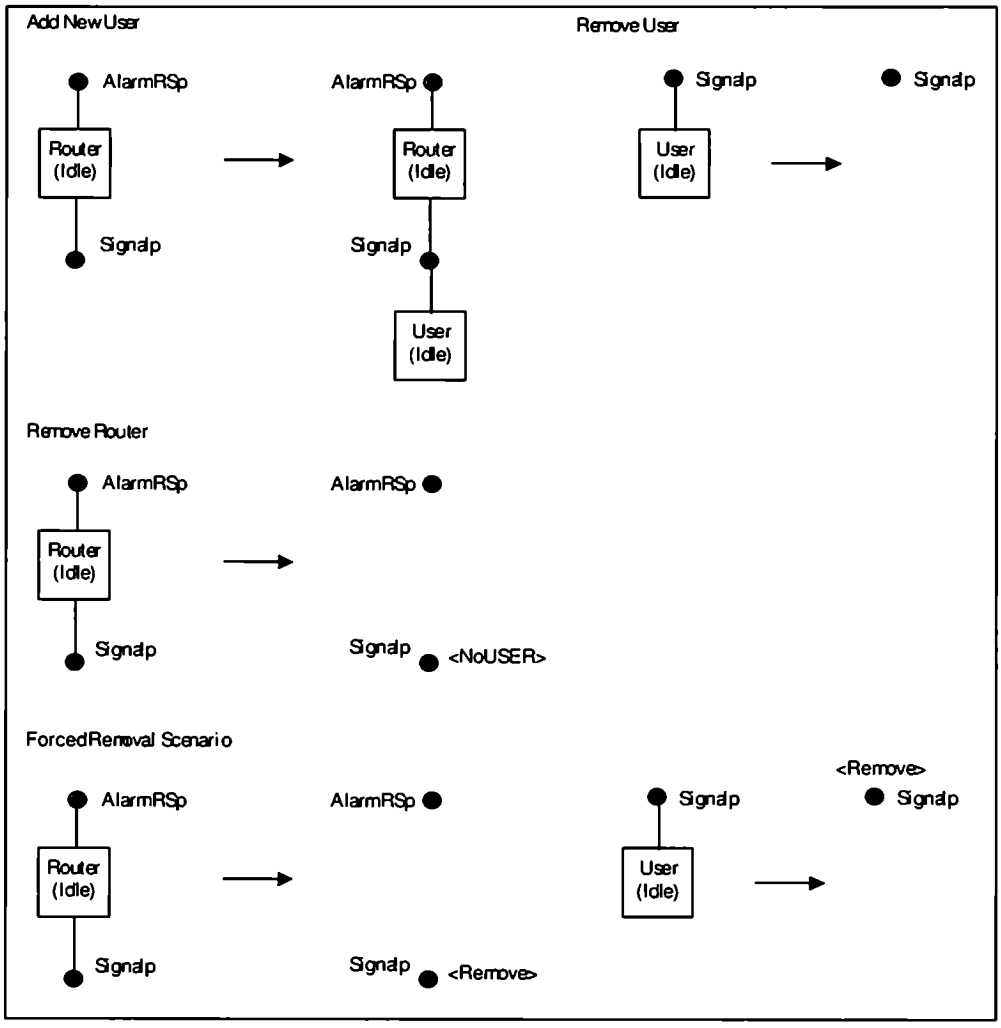

Figure 4.4: Dynamic Reconfiguration.

As we already introduced in Section 1.4, in the area of software architecture reconfiguration other work has been done using graph grammars and graph transformation. In [Le Métayer, D., 1998] also reconfiguration is treated, using a centralized approach. Together with the context-free grammar a coordinator is defined that is in charge of managing the architecture reconfiguration. The coordinator is expressed using conditional graph rewriting with rules (not context-free) on entities, links and conditions on public variables of entities textual specification. These variables are the only interactions among entities and the coordinator and are set by entities (in the textual language specification) indicating when a rule can be applied.

Other work on graph transformation for reconfiguration on software architectures are [Wermelinger, M., 1999] and [Wermelinger, M. and Fiadeiro, J., 2002] where a program design language (*COMMUNITY*) is used to represent program states and computations, and algebraic graph rewriting is employed for architecture reconfigurations. In [Taentzer, G. *et al.*, 1998] distributed graph transformation for dynamic change management is applied. These approaches also handle reconfiguration via transformation rules but they are based on general graph rewriting rules and thus assume a global, centralized control driving reconfigurations.

4.4 Style Communication Pattern

The communication pattern set can be represented as SHR productions where the communication pattern is modeled by synchronized rewriting. In this way, the communication evolution can be specified independently for each component type. Productions in this set do not change the configuration of the architecture, they only model possible interactions among components that may change, at most, component status. This representation gives an abstract representation of interactions compatible with the rest of the style description, but it is not our intention to make compulsory use of the graph modeling. Maybe it is more comfortable for a designer to specify the communication pattern in some textual formal language. What we want to show is that there is a graphical notion (compatible with the rest of the style description) that if desired can be used instead or as a complement of a lower level description. One thing to point out is the flexibility of using SHR productions that can be used for scenario-based specification of interactions and also with the possibility of easily adding new scenarios (just by introducing more productions).

As we pointed out, using productions for the communication pattern offers an abstraction compatible with the rest of the style specification which is good for analysis and general understanding of the system, but a designer may choose to use a textual language to model communications. For example in [Le Métayer, D., 1998], Le Metayer complements the graph grammar for the style with a small language with a **CSP** like notation used for describing entity (i.e. components) behavior. The difference of both approaches is that Le Metayer only uses grammars to generate the static configuration of the style obtaining graphs of the form presented in Figure 1b. He also uses context-free productions but they are applied over unary relations that identify components.

In spite of the fact that both approaches highlights the separation of static configuration, coordination and computation, the graph representation in Figure3.1b. mixes the

static configuration with the dynamics of the communications by connecting components with communication links, instead of using ports (or connectors generally speaking). For example, an instance of the case study style with fifty users attached to a router would still be connected by one port in our approach, while in the relation approach there would be fifty edges from the Router to the User and fifty from the user to the router. But still, in this second representation nothing else is said about how these links are related to each other completing the specification of the communication pattern (these is what we would identify as the role of a connector). In our case, SHR productions can do the job. In [Le Métayer, D., 1998], the language of entities is used to model components behavior including the pattern of interactions among them.

But, as an example of how both ideas can be complemented we can use the language of entities in [Le Métayer, D., 1998] as a textual representation of the communication, which offers the designer an alternative abstraction to productions.

4.4.1 Example

Following with the case study, we can show how the textual and graphical representation can support each other. Figure 4.5 and Figure 4.6a. show the textual and graphical specification of one possible communication scenario, the *User Sends Alarm* Scenario. This scenario describes a **User** (patient) sending an alarm for help to a **Router**, whom forwards it to the **Server**. After receiving the alarm the **Server** returns an acknowledgement to the **Router** and from him to the **User**, indicating that the help request was received and it is being sent to the **User**.

We will not fully describe the textual language and only use enough of it to exemplify the ideas mentioned above. The language has a **CSP** like notation, with $*[]$ as the repetitive command and $G \rightarrow C$ a guarded command. There are two possibilities for input and output communication commands:

- $\bullet a \in L?v$ and $a \in L!v$: correspond to the establishment of a rendezvous with any component linked to the current entity through a port of name L. This allows a component to communicate with an unbounded number of entities without knowing their name (This is not possible in **CSP**).
- $\bullet a : L?v$ and $a : L!v$: correspond to the establishment of a rendezvous with the explicit component linked to the current entity through a port of name L. This is necessary for a component to complete a series of communications with the same partner.

In this example we introduce another synchronization mechanism for the graphical representation, the point-to-point communication, where the synchronization is done between one sender and one receiver. The sender is represented with a production containing a tuple and the ! symbol on the shared node and the receiver is represented with a production containing the same tuple and the ? symbol on the corresponding node. A synchronized rewriting step represents the interaction of the two components.

The notation of the form $A \rightarrow B \rightarrow C$ is a shortcut for two productions of the form $A \rightarrow B$ and $B \rightarrow C$. Also, with this notation each column of productions corresponds,

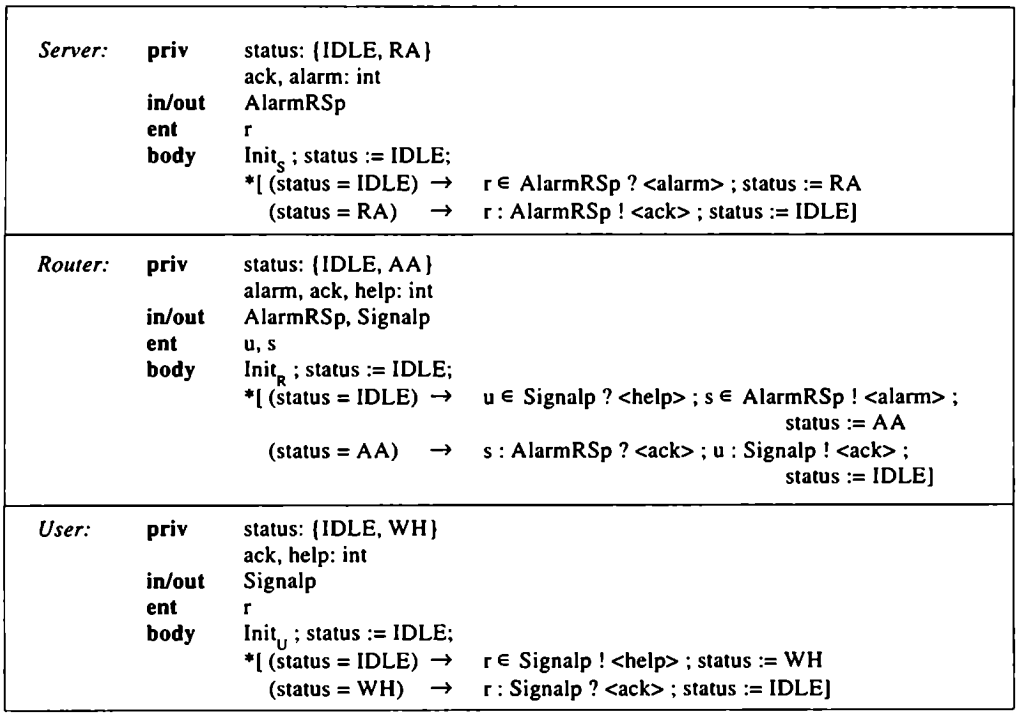

Figure 4.5: Textual representation for the Communication Pattern: User Sends Alarm Scenario.

for each component type, to the productions that participate on a synchronized rewriting step. The **Idle** status indicates that the component is not participating in any interaction or reconfiguration and it is ready to participate in a new one.

We have to note that in this part of the thesis we use for the point-to-point communication the **CSP** notation (! and ?) with the goal of pointing out the relation of our graph grammar approach and the textual representation introduced in [Le Métayer, D., 1998]. The formalization of point-to-point communication (also named *Milner* synchronization) is done in Part IV following a π-calculus notation.

Productions in Figure 4.6a. describes a **User** (patient) sending an alarm for help to a **Router**, whom forwards it to the **Server**. This is done with the first column of productions where the **User** sends the alarm < *help* >! going to a **Waiting for Help (WH)** status, the **Router** attached to that **User** receives it (< *help* >?) and forwards it to the **Server** (< *alarm* >!) going to **Receiving Alarm** status. The scenario is completed by the return of an acknowledgement (< *ack* >) from the **Server** to the **Router** and from him to the **User**. After all these the components return to the **Idle** status. Figure 4.6b. shows the application of the productions to the architecture instance where the **User** of the first **Router** sends an alarm. Note that a mapping from the textual representation to productions is very simple and can be use to complement both abstractions.

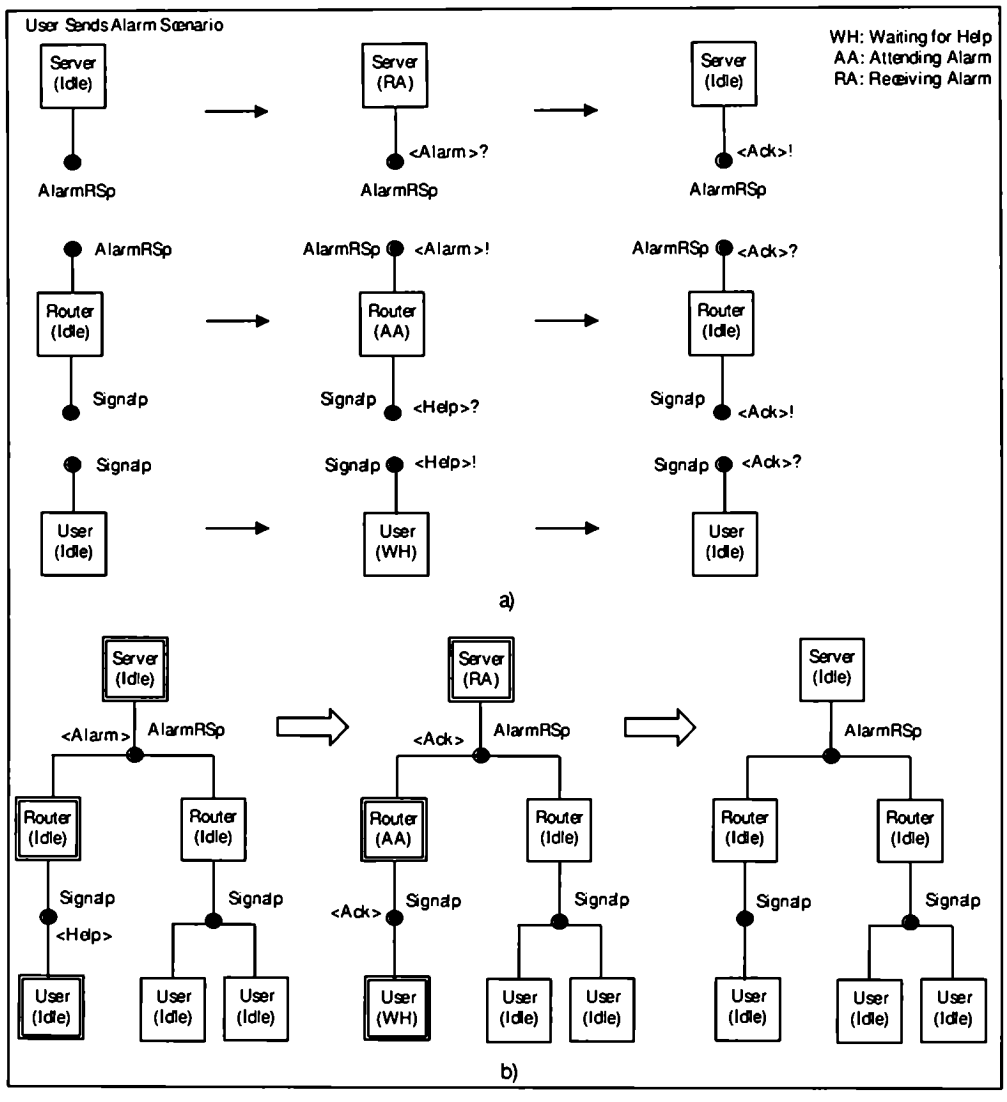

Figure 4.6: Communication Pattern Set. a) User Sends Alarm Scenario. b) Derivation showing the synchronized rewriting for the scenario.

Another possible scenario (the graphical view) is the one in Figure 4.7 where a **Router** checks a **User** subsystem. In this case there are two alternatives. The first one, where the **User** answers that everything is ok, is represented by the synchronization of productions in Figure 4.7a. And the second in Figure 4.7b., where the **User** answers with an error signal that is forwarded to the **Server** for its attention. Note that in the second alternative the acknowledgement is between **Server** and **Router** that afterwards return to **Idle** to continue attending other **Users**. The **User** with the error returns to the **Idle** without the need of any synchronization, modeling non-deterministically that in some moment the error is going to be fixed and it will start to work ok again. This

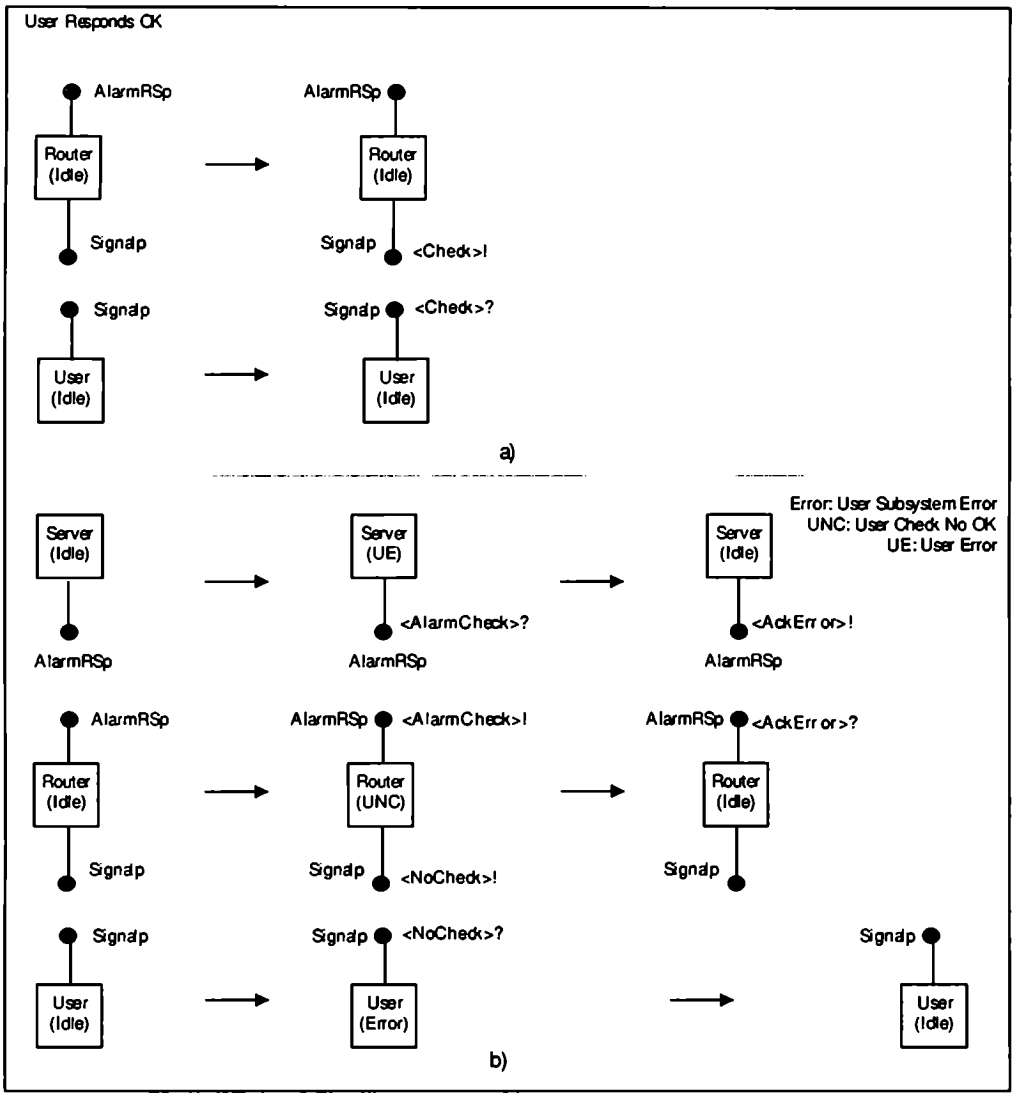

Figure 4.7: Communication Pattern Set: Router Checks User Scenario. a) User responds ok. b) User responds with an error.

was a choice of design, and a second possibility is making the **Router** wait until the problem was fixed.

At this point, we have shown how grammars and graph rewriting can be used to model the evolution of software architecture styles including the communication pattern and the dynamic reconfiguration, while being able to keep a clear separation of configuration and coordination policies. The increased expressiveness is achieved while keeping the simplicity of context-free descriptions for the behavior of each component, and implementing the coordination policies through the synchronization mechanism at the production application level. This seems to be a nice way to express synthetically

but explicitly coordination specification at the architectural level.

Our work goes in the direction of finding suitable ways of modeling software architectures without embodying in the description a specific coordination policy, rather the approach is to specify the minimal requirements on the coordination policy which can then be refined in the subsequent development steps into a specific one. To this respect we think the approach we present using synchronized rewriting is a step in the right direction. Constraint matching and synchronized production selection allows for an explicit, declarative but minimal specification of the coordination policies that can be legitimately adopted when implementing the system. Moreover the context free nature of the productions allows for a clean and simple specification of the reconfiguration step at the component level, in a completely distributed fashion.

Part IV
Adding Dynamic Reconfiguration to Styles

The design of software systems that include mobility or dynamic reconfiguration of their components is becoming more frequent. Consequently, it is necessary to have the right tools to handle their description at all stages of the development process. More specifically, it is fundamental to be able to cope with these type of requirements also in the design phase and specially for software architectures.

With this in mind and understanding the relevance of visual languages specially at the design level, we present in this part of the thesis a graphical model using SHR systems with the addition of *name mobility* (as in π-calculus [Milner, R., 1999; Sangiorgi, D. and Walker, D., 2001]). This extension to SHR systems increases the expressive power of the approach introduced in Part III, where the level of reconfiguration we were able to achieve was still limited. In this way we obtain the good characteristics of a graphical calculus together with the expressive power to describe complex mobile systems. The capability of creation and sharing of ports together with multiple simultaneous synchronizations give us a very powerful tool to specify more complex evolutions, reconfiguring multiples components by identifying specific ports. Apart from the graphical side, we can relate our calculus with π-calculus [Milner, R., 1999; Sangiorgi, D. and Walker, D., 2001]. The difference is that π-calculus is *sequential* in the sense that it allows only one synchronization at a time, while synchronized rewriting allows multiple and concurrent synchronizations all over the graph.

This part of the thesis presents the formal treatment for the addition of mobility to SHR systems. This is needed to obtain a clear semantics of the visual model of graphs and allows the formal definition of the synchronizing mechanisms. The formalization of these notions is given in chapters 5 and 6 by the use of syntactic judgements. Note that in this part of the thesis we put the emphasis in reconfiguration and mobility, but it is clear that the formal model we present here can include the three production sets that were defined in Part III.

SHR with name mobility was first introduced in [Hirsch, D. *et al.*, 2000] for modelling software architecture styles and their reconfigurations. The presentation in [Hirsch, D. *et al.*, 2000] was only for Hoare synchronization and only with the possibility of sharing new nodes (i.e. new names) as in the πI-calculus [Sangiorgi, D., 1996]. Now we follow the presentation in [Hirsch, D. and Montanari, U., 2001b; Hirsch, D. and Montanari, U., 2001a; Hirsch, D. and Montanari, U., 2001c] allowing to pass both new names and old

names in a synchronization and the formal definition for Hoare and Milner transition systems. Also, we introduce the use of bounded nodes for a compositional presentation and allow the synchronization of old names with new names. These three capabilities are necessary to model the π-calculus. In Part V we present a translation for π-calculus using Milner synchronization with the goal of studying the expressive power of the approach.

In the area of graph transformation and its application to system modelling there are interesting work (for example, [Rozenberg, 1997; Ehrig, H. *et al.*, 1999b]) where, in general, system transformations are represented with productions where their left hand side are non context-free graphs (with exception of [Montanari, U. *et al.*, 1999]). This means that they imply a centralized control that needs to know the complete map of the system, which is not well suited for distributed systems. Also none of these techniques includes any synchronization mechanism with mobility. On the other side, our use of context-free productions with synchronization and mobility is a powerful tool for describing self organising distributed systems.

The formalism presented in this part of the thesis gives a solid foundation for graphical mobile calculi which are well-suited for high level description of distributed and concurrent systems, reflected by our main practical goal that is, once again, formalizing the description of software architecture styles including their reconfigurations.

Chapter 5

Hypergraphs and Syntactic Judgements

In this chapter we formalize the notion of hypergraphs as well formed syntactic judgements generated from a set of axioms and inference rules. For an extensive presentation on the foundations of hypergraphs and HR Systems we refer to [Drewes, F. *et al.*, 1997] and for the basic definitions to Section 2.1.

We present a definition of graphs as *syntactic judgements*, where nodes correspond to names, external nodes to free names and edges to basic terms of the form $L(x_1, \ldots, x_n)$, where x_i are arbitrary names and $L \in LE$.

DEFINITION 5.1. [Graphs as Syntactic Judgements] Let $\mathcal{N}$ be a fixed infinite set of names and LE a ranked alphabet of labels. A *syntactic judgement* (or simply a judgement) is of the form $\Gamma \vdash G$ where,

1. $\Gamma \subseteq \mathcal{N}$ is a set of names (the interface of the graph).
2. G is a term generated by the grammar: $G ::= L(\vec{x}) \mid G|G \mid \nu x.G \mid nil$
   where $\vec{x}$ is a vector of names, L is an edge label with $rank(L) = |\vec{x}|$ and y is a name.

Let fn(G) denote the set of all free names of G, i.e. all names in G not bound by a ν operator. We demand that $fn(G) \subseteq \Gamma$.

We use the notation Γ, x to denote the set obtained by adding x to Γ, assuming $x \notin \Gamma$. Similarly, we will write Γ_1, Γ_2 to state that the resulting set of names is the disjoint union of Γ_1 and Γ_2.

DEFINITION 5.2. [Structural Congruence and Well-Formed Judgements]

- St ructural Congruence $\equiv$ obey structural axioms in Table 5.1.
- Th e *well-formed judgements* for constructing graphs over LE and $\mathcal{N}$ are those generated by applying the syntactic rules in Table 5.1 up to axioms of structural congruence.

Structural Axioms

$(AG1)\ (G_1|G_2)|G_3 \equiv G_1|(G_2|G_3)$ $\qquad$ $(AG2)\ G_1|G_2 \equiv G_2|G_1$

$(AG3)\ G|nil \equiv G$ $\qquad$ $(AG4)\ \nu x.\nu y.G \equiv \nu y.\nu x.G$

$(AG5)\ \nu x.G \equiv G\ if x \notin fn(G)$ $\qquad$ $(AG6)\ \nu x.G \equiv \nu y.G\{y/x\}\ if\ y \notin fn(G)$

$$(AG7)\ \nu x.(G_1|G_2) \equiv (\nu x.G_1)|G_2\ if\ x \notin fn(G_2)$$

Syntactic Rules

$$(RG1)\ \frac{}{x_1,\ldots,x_n \vdash nil} \qquad (RG2)\ \frac{L \in LE_m \quad y_i \in \bigcup_{j=1\ldots n}\{x_j\}}{x_1,\ldots,x_n \vdash L(y_1,\ldots,y_m)}$$

$$(RG3)\ \frac{\Gamma \vdash G_1 \quad \Gamma \vdash G_2}{\Gamma \vdash G_1|G_2} \qquad (RG4)\ \frac{\Gamma, x \vdash G}{\Gamma \vdash \nu x.G}$$

Table 5.1: Well-formed judgments

Axioms *(AG1)*, *(AG2)* and *(AG3)* define the associativity, commutativity and identity over *nil* for operation |, respectively. Axioms *(AG4)* and *(AG5)* state that the nodes of a graph can be hidden only once and in any order, and axioms *(AG6)* and *(AG7)* define alpha conversion of a graph with respect to its bounded names and the interplay between hiding and the operator for parallel composition, respectively.

If necessary, thanks to axiom *(AG4)*, we will write νX, with $X = \bigcup x_i$, to abbreviate $\nu x_1.\nu x_2 \ldots \nu x_n$. Note that using the axioms, for any judgement we can always have an equivalent normal form $\Gamma \vdash \nu X.G$, with G a subterm containing only composition of edges. It is clear from the above definitions that Γ and X can be made disjoint sets of nodes using the axioms and that $nodes(G) \subseteq (\Gamma \cup X)$ (with $nodes(G)$ the set of all nodes appearing in G).

Rule *(RG1)* creates a graph with no edges and n nodes and rule *(RG2)* creates a graph with n nodes and one edge labelled by L and with m tentacles (note that there can be repetitions among nodes in $\vec{y}$, i.e. some tentacles can be attached to the same node). Rule *(RG3)* allows to put together (using |) two graphs that share the same set of external nodes. Finally, rule *(RG4)* allows to hide a node from the environment.

We can state the following correspondence theorem.

THEOREM 5.3. [Correspondence of Graphs and Judgements] *Well-formed syntactic judgements up to structural axioms are isomorphic to graphs up to isomorphism.*

Proof. *The first part of the theorem, from judgements to graphs, can be proved by induction on the structure of judgements generated by the grammar presented in Definition 5.1. For this we define the following translation function $[\![-]\!]$ (up to isomorphism),*

Base Case:

- $[\![[x_1, \ldots, x_k \vdash nil]\!] = \langle \{n_1, \ldots, n_k\}, \emptyset, \emptyset, \{n_1, \ldots, n_k\}, \emptyset, \{(n_1, x_1), \ldots, (n_k, x_k)\}\rangle$,
  i.e., is a graph with no edges and k *external nodes.*

- $[\![[x_1, \ldots, x_k \vdash L(y_1, \ldots, y_m)]\!] =$
  $\langle \{n_1, \ldots, n_k\}, \{e_L\}, \{(e_L, < m_1, \ldots, m_m >)\}, \{n_1, \ldots, n_k\}, \{(e_L, L)\},$
  $\{(n_1, x_1), \ldots, (n_k, x_k)\}\rangle$ *with* $(m_i = n_j$ *iff* $y_i = x_j)$,
  i.e., is a graph with k *external nodes and one edge with label* $L \in LE_m$.

Inductive Hypothesis: *Given a judgement* $\Gamma \vdash G$, *the translation* $[\![\Gamma' \vdash G']\!]$, *for any proper subterm* G' *of* G, *is a graph.*

- $[\![[\Gamma \vdash \nu x.G]\!]$
  By the I.H. we know that,
  $[\![\Gamma, x \vdash G]\!] = \langle N \cup \{n_x\}, E, att, ext \cup \{n_x\}, lab_{LE}, lab_{LN} \cup \{(n_x, x)\}\rangle$, *with* $n_x \notin N \cup ext$ *and* $lab_{LN} : N \rightarrow \Gamma$.
  Then by rule $(RG4)$ *we can define,*
  $[\![\Gamma \vdash \nu x.G]\!] = \langle N \cup \{n_x\}, E, att, ext, lab_{LE}, lab_{LN}\rangle$
  In words, we have the same graph for H *but with one less external node.*

- $[\![[\Gamma \vdash G_1 | G_2]\!]$
  By the I.H. we know that,
  $[\![\Gamma \vdash G_1]\!] = \langle N_1, E_1, att_1, ext_1, lab_{LE1}, lab_{LN1} : ex\ t_1 \rightarrow \Gamma\rangle$ *and*
  $[\![\Gamma \vdash G_2]\!] = \langle N_2, E_2, att_2, ext_2, lab_{LE2}, lab_{LN2} : ex\ t_2 \rightarrow \Gamma\rangle$
  Then by rule $(RG3)$ *we can define,*
  $[\![\Gamma \vdash G_1 | G_2]\!] = \langle N_1 \cup \Psi(N_2), E_1 \cup E_2, att_1 \cup \Psi^*(att_2), ext_1, lab_{LE1} \cup lab_{LE2}, lab_{LN1}\rangle$,
  with Ψ *the bijective function for the isomorphic graph for* G_2 *with respect to the external nodes, defined as* $\Psi(n_i^2) = n_j^1$ *if* $lab_{LN1}(n_j^1) = lab_{LN2}(n_i^2)$ *for all* $n_i^2 \in ext_2$
  In words, $[\![\Gamma \vdash G_1 | G_2]\!]$ *is the graph obtained from* $[\![\Gamma \vdash G_1]\!]$ *and* $[\![\Gamma \vdash G_2]\!]$ *by identifying homonymous external nodes.*

Conversely, we prove the second part of the theorem by construction using the syntactic rules of table 5.1.

- *For a graph with no edges and no bound nodes we use rule (RG1).*

- *For a graph* G *with* $m > 0$ *edges,* n *nodes and no bound nodes:*

  1. *Using rule (RG2) applied* m *times to create, for each edge* e *in* G, *the judgement for a graph with* n *external nodes and one edge,*
     $lab^*_{LNG}(ext_G) \vdash lab_{LNG}(e)(lab^*_{LNG}(att_G(e)))$ *and*

2. *Using rule (RG3) applied $m-1$ times over the m judgements created in step 1.*

- *For a graph G with n nodes and $r \leq n$ bound nodes:*

3. *Beginning with the judgement for graph G' of n external nodes that corresponds to graph G but with no bound nodes (obtained by the previous steps), apply r times rule (RG4) for each node to be bound.*

Completing the proof of the isomorphism, it is obvious that the compositions of the corresponding translations result in identities. □

5.1 Ring Example

We use graphs to represent system software architectures. In this context, edges are components and nodes are ports of communication. External nodes are connecting ports to the environment. Edges sharing a node mean that there is a communication link among them. So, let us take the graph in Figure 5.1*a* that represents a ring of four components with two connecting ports. Edges representing components are drawn as boxes attached to their corresponding ports. The label of an edge is the name of the component and the arrow indicates the order of the attachment nodes. In this case we only have edges with two tentacles. Names in filled nodes identify external nodes and empty circles are bound nodes. Figure 5.1*b* shows how the corresponding well-formed judgement is obtained. Note that *(RG3)* needs the same set of names Γ in both premises.

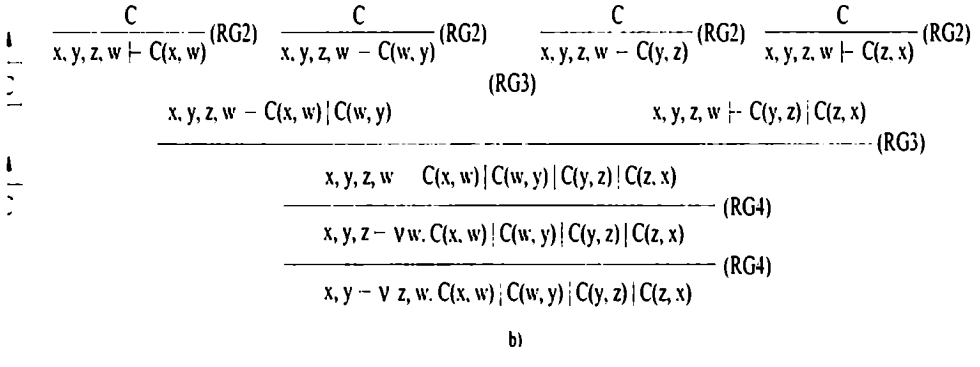

Figure 5.1: The graph and the corresponding judgement for the ring example

Chapter 6

SHR Systems with Name Mobility

In this chapter we introduce the notion of SHR systems adding to it the capability of name creation and mobility. We formalize these notions as well formed syntactic judgements generated from a set of axioms and inference rules, allowing to model in a simple way the synchronization and reconfiguration of graphs. Also, using judgements to represent graphs and their rewriting will be useful for analyzing different synchronization policies and the expressive power of the approach.

S

The following definitions present an extension to SHR systems where we allow the declaration and creation of names on nodes and use synchronized rewriting for name mobility. In this way it is possible to specify reconfigurations over the graphs by changing the connections between edges.

The following sections introduce the approach with a general definition of SHR. This allows the introduction of different synchronization mechanisms achieved by the possibility of defining different (mobile) synchronization algebras (and their corresponding transition rules). Specifically, we present the "implementation" of *Hoare* and *Milner synchronization styles.*

It is worth mention, that [König, B. and Montanari, U., 2001] presents a bisimilarity for synchronized graph rewriting with name mobility (based on the work of [Hirsch, D. *et al.*, 2000]) proving it to be a congruence. Also they introduce a so-called *format* which is a syntactic condition on productions ensuring that bisimilarity is a congruence. This last result is original not only for graph rewriting, but also for mobility in general.

6.1 SHR Systems as Syntactic Judgements

Recalling what was introduced in Chapter 4, to model synchronized rewriting, it is necessary to add some labels to the nodes in productions. Assuming to have an alphabet of actions *Act*, then we associate actions to some of the nodes. In this way, each rewrite of an edge must synchronize actions with (a number of) its adjacent edges and then all the participants will have to move as well (how many depends on the synchronization policy).

It is clear that synchronized rewriting will allow the propagation of synchronization all over the graph where productions are applied. A *SHR grammar*, or simply a grammar, consists of an initial graph and a set of productions. A derivation is obtained by starting with the initial graph and applying a sequence of rewriting rules, each obtained by synchronizing possibly several productions.

As we saw in Chapter 4, for the Hoare synchronization all adjacent edges must match the actions imposed on the shared node and for Milner synchronization only two of the adjacent edges will synchronize by matching their complementary actions (usually called observations). For simplicity, in this part of the thesis we will work only with one action per node.

Now that we have synchronized rewriting, we need to add to productions the capability of sharing nodes. This is obtained by letting a production to declare new names for the nodes it creates, and by sharing these names and/or other existing names with the rest of the graph using the synchronization process. This is done in a production by adding to the action in a node, a tuple of names that it wants to communicate. Therefore, the synchronization of a rewriting rule has to match not only actions, but also the tuples of names. After the matching is obtained and the productions applied, the declared names that were matched are used to obtain the final graph of the rewriting by merging the corresponding nodes.

As is done in π-calculus, we allow to merge new nodes with other nodes (new or old). Merging among already existing nodes is not allowed. Relaxing this constraint, would permit fusions of nodes in the style of the *fusion*-calculus [Victor, B., 1998]. Instead, we will also consider a syntactic restriction of our formalism in which we will allow merging new nodes only, in the style of the πI-calculus [Sangiorgi, D., 1996]. We will study the relationship with π-calculus in Part V discussing the corresponding translations for π and πI-calculus. These policies of which nodes are shared are independent of the synchronization mechanisms applied.

To formalize SHR systems we use, as in Chapter 5, judgements and define the notion of *transitions*.

DEFINITION 6.1. [Transitions] Let $\mathcal{N}$ be a fixed infinite set of names and Act a ranked set of actions, where each action $a \in Act$ is associated with an arity (indicating the number of nodes it can share). We define a *transition* as:

$$\Gamma \vdash G \xrightarrow{\Lambda} \Gamma, \Delta \vdash G'$$

$$\textit{with} \quad \Lambda : \Gamma \multimap (Act \times \mathcal{N}^*) \quad \Delta = \{z \mid \exists x.\ \Lambda(x) = (a, \vec{y}),\ z \notin \Gamma,\ z \in set(\vec{y})\}$$

A transition is represented as a logical sequent which says that G is rewritten into G' satisfying a *set of requirements* Λ. The free nodes of graph G' must include the free nodes of G and those new nodes (Δ) that are used in synchronization. Note that Δ is determined by the Γ and Λ of the corresponding transition.

The set of requirements $\Lambda \subseteq \Gamma \times Act \times \mathcal{N}^*$ can be defined as a partial function in its first argument, i.e. if $(x, a, \vec{y}) \in \Lambda$ we write $\Lambda(x) = (a, \vec{y})$ with $arity(a) = |\vec{y}|$. With

$\Lambda(x)\uparrow$ we mean that the function is not defined for x, i.e. that there is no requirement in Λ with x as first argument. Function $set(\vec{y})$ returns the set of names in vector $\vec{y}$. The definition of Λ as a function means that all edges in G_1 attached to node x that are participating in a synchronization, must satisfy the conditions of the corresponding synchronization algebra. The function is partial since not all nodes need to be loci of synchronization.

Note that to share only new nodes, it is enough to impose on Λ the condition that names of vectors $\vec{y}$ should not be in Γ ($set(\vec{y}) \cap \Gamma = \emptyset$). Then, Δ does not depend on Γ and can be written as:

$$\Delta = \bigcup_{xa\vec{y} \in \Lambda} set(\vec{y})$$

We can redefine productions and grammars with respect to judgements.

DEFINITION 6.2. [Productions] A *SHR production*, or simply a production, is a transition of the form:

$$x_1, \ldots, x_n \vdash L(x_1, \ldots, x_n) \xrightarrow{\Lambda} x_1, \ldots, x_n, \Delta \vdash G$$

Productions have to be applied over different graphs, so they will be alpha convertible. In this way, names can be changed to match the ones of the graph to apply the production and new names can be arranged to avoid name clashing and a correct synchronization. The context-free character of productions is here made clear by the fact that the graph to be rewritten consists of a single edge with distinct nodes.

DEFINITION 6.3. [Grammars] Let $\mathcal{N}$ be a fixed infinite set of names, LE a ranked alphabet of labels and Act a ranked set of actions. A *grammar* consists of:

1. An initial graph $\Gamma_0 \vdash G_0$,
2. a set $\mathcal{P}$ of productions and
3. a synchronization mechanism.

A *derivation* is a finite or infinite sequence of the form $\Gamma_0 \vdash G_0 \xrightarrow{\Lambda_1} \Gamma_1 \vdash G_1 \xrightarrow{\Lambda_2} \ldots \xrightarrow{\Lambda_n} \Gamma_n \vdash G_n \ldots$, where $\Gamma_{i-1} \vdash G_{i-1} \xrightarrow{\Lambda_i} \Gamma_i \vdash G_i$, $i = 1 \ldots n$ is a transition in the set $T(\mathcal{P})$ of transitions generated by $\mathcal{P}$. Transitions $T(\mathcal{P})$ are generated by $\mathcal{P}$ applying the transition rules of the chosen synchronization mechanism, as defined in the next sections.

6.2 Hoare Synchronization

The first synchronization mechanism we present is *Hoare synchronization*, where each rewrite of an edge must share on each attachment node the same action with all the edges connected to that node.

For example, consider n edges which share one node, such that no other edge is attached to that node, and let us take one production for each of these edges. Each of these productions have an action on that node (a_i for $i = 1 \ldots n$). If $a_i \neq a_j$ (for some i, j), then the n edges cannot be rewritten together (using these productions). If all a_i are the same, then they can move, via the context-sensitive rewriting rule obtained by merging the n context-free productions. The use of SHR productions in a rewriting system implies the application of several productions where all edges to be rewritten and sharing a node must apply productions that satisfy the same actions.

Given that Hoare synchronization (*formally*) requires that all edges sharing a node must participate in the synchronization, but since not all nodes need to be loci of synchronization, an identity action ε is defined which is required in a node by all the productions which do not synchronize on that node. We impose the condition that the identity action has arity zero, so if it is imposed on a node then no name can be shared on that node. We have to note that in Chapter 4 we abuse of the graphical notation and identity actions where not included in the examples.

In particular, to model edges which do not move in a transition we need productions with identity actions on their external nodes, where an edge with label L is rewritten to itself. This is called the *id production* $id(L)$. Then, the set $\mathcal{P}$ of productions must include productions $id(L)$ for all symbols L in LE. The corresponding judgements are as follows

$$, \ldots, x_n \vdash L(x_1, \ldots, x_n) \xrightarrow{\{(x_1,\varepsilon,<>),\ldots,(x_n,\varepsilon,<>)\}} x_1, \ldots, x_n \vdash L(x_1, \ldots, x_n).$$

For any relation $R \subseteq \Gamma \times Act \times \mathcal{N}^*$ we define $n(R) = \bigcup_{(x,a,\vec{y}) \in R} set(\vec{y})$ and will call a mapping $\rho_R : \Delta \to n(R)$ *the most general unifier (mgu)* of R (with $\Delta = n(R) \backslash \Gamma$) whenever ρ_R is a function and if, of all ρ' with this property, ρ_R identifies the minimal number of names. The mapping ρ_R is exactly the most general unifier of the equations $(a = b) \wedge (\vec{y} = \vec{z})$ (whenever $(x, a, \vec{y}), (x, b, \vec{z}) \in R$) and is unique up to injective renaming. It does not exist if there are tuples $(x, a, \vec{y}), (x, b, \vec{z}) \in R$ with $a \neq b$ or if the equations $\vec{y} = \vec{z}$ imply an equation $v = w$ with v, w different old names. Thus the external nodes (i.e., $x \in \Gamma$) that appear in $n(R)$ are considered constants. In this way new names are unified with either new or old names, but it is not possible to have a unification among old names (two different constants cannot be unified).

The *mgu* is necessary to resolve the identification of names (i.e. nodes) that is consequence of a synchronization operation and to avoid name capture.

DEFINITION 6.4. [Hoare Transition System] Let $< G_0, \mathcal{P} >$ be a grammar. All transitions $T(\mathcal{P})$ using Hoare synchronization are obtained from the transition rules in Table 6.1.

Rule *(ren)* is necessary to allow applying a production to different graphs (this is done by substitution ξ). In this way, names can be changed to match the ones of the graph to apply the production and new names can be arranged to avoid name clashing and a correct synchronization. For the set of new nodes Δ, ξ is an injective substitution

$$(ren)\ \frac{x_1,..,x_n \vdash L(x_1,..,x_n) \xrightarrow{\Lambda} x_1,..,x_n,\Delta \vdash G \in \mathcal{P}}{\xi(x_1,..,x_n),\Gamma \vdash \xi(L(x_1,..,x_n)) \xrightarrow{\rho_{\xi(\Lambda)}(\xi(\Lambda))} \xi(x_1,..,x_n),\Gamma,\Delta' \vdash \rho_{\xi(\Lambda)}(\xi(G))}$$

if $\rho_{\xi(\Lambda)}$ exists

$$(com)\ \frac{\Gamma \vdash G_1 \xrightarrow{\Lambda_1} \Gamma,\Delta_1 \vdash G'_1 \quad \Gamma \vdash G_2 \xrightarrow{\Lambda_2} \Gamma,\Delta_2 \vdash G'_2}{\Gamma \vdash G_1|G_2 \xrightarrow{\rho_{\Lambda_1\cup\Lambda_2}(\Lambda_1\cup\Lambda_2)} \Gamma,\Delta \vdash \rho_{\Lambda_1\cup\Lambda_2}(G'_1|G'_2)}$$

if $\Delta_1 \cap \Delta_2 = \emptyset$ and $\rho_{\Lambda_1\cup\Lambda_2}$ exists

$$(open)\ \frac{\Gamma,x \vdash G \xrightarrow{\Lambda\cup\{(x,a,\vec{y})\}} \Gamma,x,\Delta \vdash G'}{\Gamma \vdash \nu x.G \xrightarrow{\Lambda} \Gamma,\Delta' \vdash \nu Z.G'}\ x \in n(\Lambda)$$

$$(hide)\ \frac{\Gamma,x \vdash G \xrightarrow{\Lambda\cup\{(x,a,\vec{y})\}} \Gamma,x,\Delta \vdash G'}{\Gamma \vdash \nu x.G \xrightarrow{\Lambda} \Gamma,\Delta' \vdash \nu x,Z.G'}\ x \notin n(\Lambda)$$

where $Z = set(\vec{y})\backslash(\Delta' \cup \Gamma)$

Table 6.1: Transition Rules for Hoare Synchronization

of new names. The non-injectiveness of ξ allows to apply the production over graphs with edges that may have several tentacles attached to the same node.

Notice that $\rho_{\xi(\Lambda)}(\xi(\Lambda))$ is still a function on requirements therefore requirements on nodes identified by ξ must coincide. Also, isolated nodes,with no requirement on them, are added (those in Γ) for the application of successive transition rules. Remember that for any transition, as presented in Definition 6.1, Δ is uniquely identified by the corresponding Γ and Λ (e.g. $\Delta = n(\Lambda)\backslash\Gamma$).

Rule *(com)* is the one that completes the synchronization process. Given that all edges must participate, Hoare synchronization is modeled as the union of the synchronization requirements ($\rho_{\Lambda_1\cup\Lambda_2}(\Lambda_1 \cup \Lambda_2)$) where the existence of $\rho_{\Lambda_1\cup\Lambda_2}$ assures that the rule can only be applied when the requirements on all the nodes are satisfied and the shared nodes are actually identified. Condition $\Delta_1 \cap \Delta_2 = \emptyset$ avoids name capture.

Rule *(open)* allows to share with the environment a node that was originally bounded. This rule may be used for sharing a port of communication that was local among some components and that now they want to allow others to communicate with them by that port. Note as we are opening name x we still have to keep bounded those names that are only shared by x (i.e. set Z). Notice that $\{x\} \cup \Delta = \Delta' \cup Z$.

Also, rule *(open)* is used for what is called an *extrusion* allowing the creation of

privately shared ports. Extrusion allows to export and share bounded nodes. But once synchronization is completed, it hides away those private names that were synchronized meaning that the names are still bound, but their scope has grown. Extrusion is usually (as in Milner synchronization and π-calculus) done by using together rules *(open)* and *(close)*. But in the case of Hoare synchronization, where many edges have to synchronize in a shared node, a *(close)* rule is not very useful because it cannot be sure when to hide the private names that are extruded. It is more reasonable to use the *(com)* and at the end of the whole operation use rule *(hide)* on the corresponding names.

Rule *(hide)* deals with hiding of names. It indicates that we do not only have to hide the wanted name, but also all the names shared by synchronization only on that name (i.e. set Z). Note that there is little difference with rule *(open)*, which is the fact that for rule *(hide)* the node to be hidden (x) must not be shared by other nodes. In this case, $\Delta = \Delta' \cup Z$.

6.2.1 Example

In this example an instance of the ring architecture style starts with a ring configuration and at some point in its evolution is reconfigured to a star. Given that in this part of the thesis we are focused on the reconfiguration and mobility issue, we omit the component state from its identification.

Figure 6.1*a* shows the grammar. The initial graph together with production *Brother* construct rings. Production *Star Reconfiguration* is used to reconfigure a ring into a star by creating a new node *(w)* and synchronizing using action r. The new node is distributed among components to identify it as the center of the star. Requirements $(x, r, < w >)$ and $(y, r, < w >)$ are represented graphically imposing the pair $(r, < w >)$ on nodes x and y on the right hand side, meaning that the rewriting step is only possible if requirements are satisfied.

Figure 6.1*b* shows a possible derivation where a ring of four components is reconfigured (thick arrow). Components with thick border indicate the component where rule *Brother* is applied. Figure 6.2 shows part of the proof that corresponds to the final step of the derivation in Figure 6.1*b*. Given their simplicity and for clarity of the example we omit the *idle* productions and the application of rule *ren* for the proof.

This simple example shows how the approach can be used to specify complex reconfigurations including the combination of different styles. In [Hirsch, D. *et al.*, 2000] another example of reconfiguration can be found based on a real case of a Remote Medical Care System.

6.3 Milner Synchronization

Now that we have presented Hoare synchronization which can be considered as the most general type of synchronization, we formalize in this section the *Milner synchronization*.

This synchronization mechanism only allows, in a node, to synchronize actions from two of all edges sharing that node, and only those two edges will be rewritten as a consequence of that synchronization.

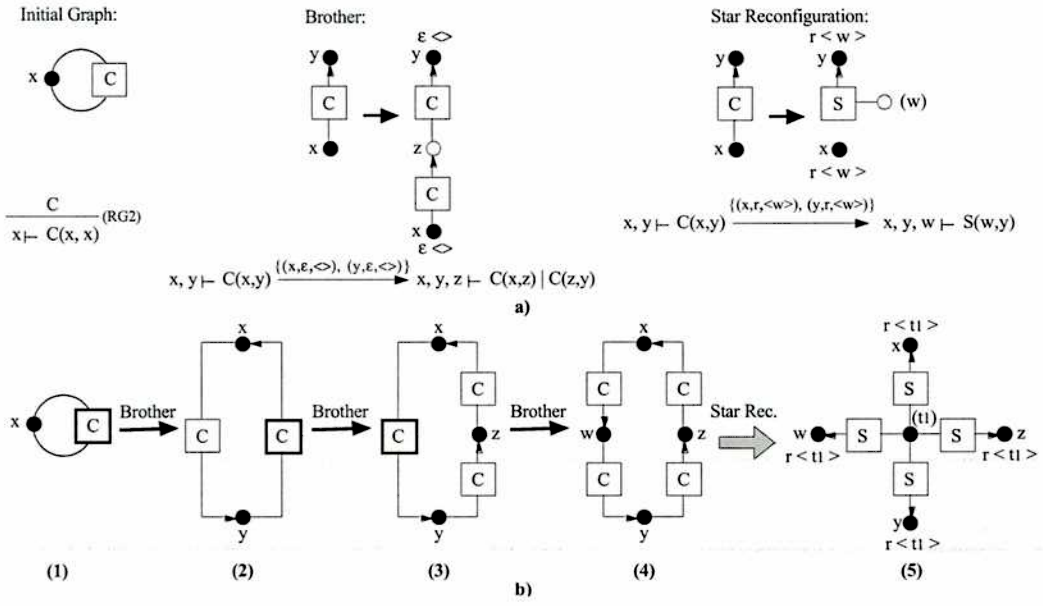

Figure 6.1: Ring grammar with star reconfiguration

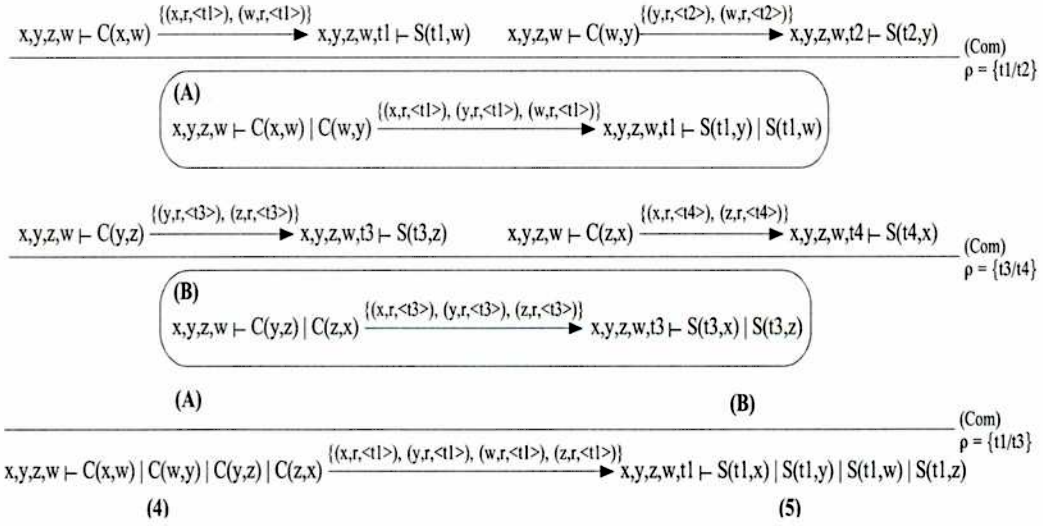

Figure 6.2: Proof of transition between graphs (4) and (5) in Figure 6.1*b*

In this case, the set of actions Act is formed by two disjoint sets of actions and coactions (Act^+ and Act^-), and a special silent action (τ with $arity(\tau) = 0$). For each action $a \in Act^+$ there is a coaction $\bar{a} \in Act^-$. A requirement of the form $(x, \bar{a}, \vec{y})$ represents an output of names in $\vec{y}$ via port x with action a and a requirement of the form $(x, a, \vec{y})$ represents an input of names in $\vec{y}$ via port x with action a. A synchronization will result of the matching of an action and its corresponding coaction with the resulting unification of their shared names as it was done for Hoare synchronization. Given that after synchronizing two requirements we are sure that the synchronization in that node

is finished, the corresponding tuples are replaced by a silent action and an empty list of names.

Note that what we are defining is a general Milner synchronization where simultaneous synchronizations are allowed, so the π-calculus synchronization mechanism is a special case where only one synchronization at a time is allowed. For Milner synchronization we do not need an idle action ϵ.

For a better understanding of their relationship the selection of the notation for actions and coactions for this part of the thesis is similar to π-calculus. In Section 4.4, we use a **CSP** like notation for the communication pattern.

DEFINITION 6.5. [Milner Transition System] Let $< G_0, \mathcal{P} >$ be a grammar. All transitions $T(\mathcal{P})$ using Milner synchronization are obtained from the transition rules in Table 6.2 starting from the set of productions $\mathcal{P}$ over the initial graph G_0.

Property $P(R)$, mgu ρ_R and function $m(R)$ are defined at the top of Table 6.2 and are needed in rules *(ren)* and *(com/close)*.

Property $P(R)$ is satisfied if set R contains tuples that respect the Milner synchronization condition over actions in a node. This means that if in node x there are (only) two tuples trying to synchronize, then there is an action a in the first one and a coaction $\bar{a}$ in the other. The mgu ρ_R is defined in a similar way as for Hoare Synchronization but satisfying the corresponding Milner condition. Function $m(R)$ is in charge of solving the synchronization, generating the new set of requirements where the synchronizing tuples are replaced by silent actions.

Rule *(ren)* is necessary to allow applying a production to different graphs (this is done by ξ that is a possibly non-injective substitution). In this way, names can be changed to match the ones of the graph to apply the production and new names can be arranged to avoid name clashing and a correct synchronization. For the set of new nodes Δ, ξ is an injective substitution of new names. The non-injectiveness of ξ allows to apply the production over graphs with edges that may have several tentacles attached to the same node. Set Δ' includes only those new names that are still being shared by other external nodes that have not been synchronized yet. The application of function m is necessary given that if an edge have several tentacles attached to the same node, we have to check if putting together their respective requirements still respects Milner synchronization. Therefore, $m(\xi(\Lambda))$ is still a function on requirements and requirements on nodes identified by ξ must satisfy property P. Condition $((\Lambda(x_i)\downarrow \wedge \xi(\Lambda(x_i)) = \xi(\Lambda(x_j)))$ *implies* $x_i \neq x_j \Rightarrow \xi x_i \neq \xi x_j)$ avoids the case where substitution ξ makes two tentacles share the same node $(\xi x_i = \xi x_j)$ with exactly the same requirement tuples $(\xi(\Lambda(x_i)) = \xi(\Lambda(x_j)))$.

Also, isolated nodes are added (those in Γ) for the application of successive transition rules. Remember that for any transition, as presented in Definition 6.1, Δ is uniquely identified by the corresponding Γ (in this case $x_1, \ldots, x_n$) and Λ.

Rule *(com/close)* is the responsible of identifying nodes. This rule takes care of two types of communication. The first one is when an existing node is shared and identified with some new nodes (an old node is merged with new nodes). The result of the synchronization works as a usual *(com)* rule (i.e synchronization occurs but no name is restricted, see for example rule *(Com)* in Section 2.4). The second type of

$$P(R) = [\forall x.\{(\alpha,\vec{v})|(x,\alpha,\vec{v}) \in R\} = \{(a,\vec{v}),(\bar{a},\vec{w})\} \text{ or } \{(\alpha,\vec{v})\} \text{ or } \emptyset]$$

$$\rho_R = mgu\,\{\vec{v} = \vec{w} | \exists x.(x,a,\vec{v}),(x,\bar{a},\vec{w}) \in R\}$$

$$\begin{array}{c} m(R)(x) = case\{(\alpha,\vec{v})|(x,\alpha,\vec{v}) \in R\} \\ \{(a,\vec{v}),(\bar{a},\vec{w})\} \rightarrow (\tau,<>) \\ \{(\alpha,\vec{v})\} \rightarrow (\alpha,\rho_R\vec{v}) \end{array}$$

$$(ren)\ \frac{x_1,..,x_n \vdash L(x_1,..,x_n) \xrightarrow{\Lambda} x_1,..,x_n,\Delta \vdash G \in \mathcal{P}}{\Gamma,\xi(x_1,..,x_n) \vdash \xi L(x_1,..,x_n) \xrightarrow{m(\xi\Lambda)} \Gamma,\xi(x_1,..,x_n),\Delta' \vdash \nu Z.\rho_{\xi\Lambda}(\xi G)}$$

if $[(\Lambda(x_i)\downarrow\ \wedge\ \xi(\Lambda(x_i)) = \xi(\Lambda(x_j)))$ implies $x_i \neq x_j \Rightarrow \xi x_i \neq \xi x_j]$
and $P(\xi\Lambda)$ and $\rho_{\xi\Lambda}$ *exists*
where
$Z = \rho_{\xi\Lambda}(\xi\Delta)\backslash(\xi(x_1,..,x_n) \cup \Delta')$

$$(com/close)\ \frac{\Gamma \vdash G_1 \xrightarrow{\Lambda_1} \Gamma,\Delta_1 \vdash G'_1 \quad \Gamma \vdash G_2 \xrightarrow{\Lambda_2} \Gamma,\Delta_2 \vdash G'_2}{\Gamma \vdash G_1|G_2 \xrightarrow{m(\Lambda_1\cup\Lambda_2)} \Gamma,\Delta \vdash \nu Z.\rho_{\Lambda_1\cup\Lambda_2}(G'_1|G'_2)}$$

if $\Lambda_1 \cap \Lambda_2 = \emptyset$ and $P(\Lambda_1 \cup \Lambda_2)$ and $\rho_{\Lambda_1\cup\Lambda_2}$ *exists*
where
$Z = \rho_{\Lambda_1\cup\Lambda_2}(\Delta_1 \cup \Delta_2)\backslash(\Gamma \cup \Delta)$

$$(par)\ \frac{\Gamma \vdash G_1 \xrightarrow{\Lambda} \Gamma,\Delta \vdash G'_1 \quad \Gamma \vdash G_2}{\Gamma \vdash G_1|G_2 \xrightarrow{\Lambda} \Gamma,\Delta \vdash (G'_1|G_2)}$$

$$(res)\ \frac{\Gamma,x \vdash G \xrightarrow{\Lambda} \Gamma,x,\Delta \vdash G'}{\Gamma \vdash \nu x.G \xrightarrow{\Lambda'} \Gamma,\Delta' \vdash \nu Z.G'}$$

where $(\Lambda(x)\uparrow\ \vee\ \Lambda(x) = (\tau,<>))$ and $\Lambda' = \Lambda\backslash\{(x,\tau,<>)\}$ and

$$Z = \begin{cases} \emptyset & if\ x \in n(\Lambda) \\ \{x\} & otherwise \end{cases}$$

Table 6.2: Transition Rules for Milner Synchronization

communication is when only new nodes are identified, which usually corresponds to what is called a *(Close)* rule. In this case when the rule is used together with rule *(res)* they cause an *extrusion*. These rules allow to export and share bounded nodes. But once synchronization is completed, rule *(com/close)* hides away those private names that were synchronized (Z) meaning that the names are still bound, but their scope has grown. Set Δ includes only those new names that are still being shared by other external nodes that have not been synchronized yet. Condition ($\Lambda_1 \cap \Lambda_2 = \emptyset$) is necessary to avoid (similarly to what happens in (*ren*)) the case where the two transitions contain requirement tuples that are exactly equal and that are not validated by property $P(\Lambda_1 \cup \Lambda_2)$.

Rule *(par)* is defined as usual allowing the application of a transition over a subgraph of a bigger one.

Rule *(res)* takes account of four cases. The first two ($\Lambda(x) \uparrow$ or $\Lambda(x) = (\tau, <>)$ with x the name to extrude, i.e. $x \in n(\Lambda)$) correspond to what is usually called an *(Open)* rule that works in a similar way as for Hoare synchronization. They are used to export bounded nodes (in these cases $Z = \emptyset$). The other cases are for the bounding operation that in Milner Synchronization is called restriction. In the first case the rule restricts a node not participating in a synchronization ($\Lambda(x) \uparrow$) and in the second one the rule is applied over nodes where a synchronization has taken place because we are sure that it is complete ($\Lambda(x) = (\tau, <>)$). A node that can still participate in a synchronization ($\Lambda(x) \neq (\tau, <>)$) cannot be bound.

6.3.1 Example

This example presents a tree structure style and the necessary reconfiguration productions to move a leaf from one father component to another. The instances of this style are trees with a root (**R**), a middle level of components (**M**) connected to the root, and for each middle component a number of leafs (**L**) connected to it.

Figure 6.3*b* shows the so-called *static productions* that construct the style configurations. The initial graph (not shown in the figure) is a root component. Figure 6.3*a* corresponds to the reconfiguration productions and their judgements. The first production is the one applied to the leaf to be moved, declaring the node (t) where it will be received. The second production is for the actual father of the leaf, which synchronizes with the leaf (using action *Move*) and with the new father (using action *MoveLeaf*). And the last production is the one for the new father synchronizing and attaching the leaf to the corresponding node (r).

A system instance of the style is shown in Figure 6.3*c* where a leaf (noted as L^*) is moved in one step from the third **M** component to the first one. Components and ports with thick border and grey color, respectively, are the ones participating in the synchronization. Figure 6.3*d* shows part of a possible proof that corresponds to Figure 6.3*c*. The first part of the proof (until (**A**)) corresponds to the synchronization in node w using the first and second production over **M(w,x)** and **L(w)**. The second part of the proof (until (**B**)) is obtained applying rule *(par)* on the third production over **M(y,x)** and the subgraph that does not participate in the synchronization. Then, rule *(com/close)* is applied again on transitions (**A**) and (**B**) for the synchronization in node x. Again for clarity of the example we omit the application of rule *ren* over the productions.

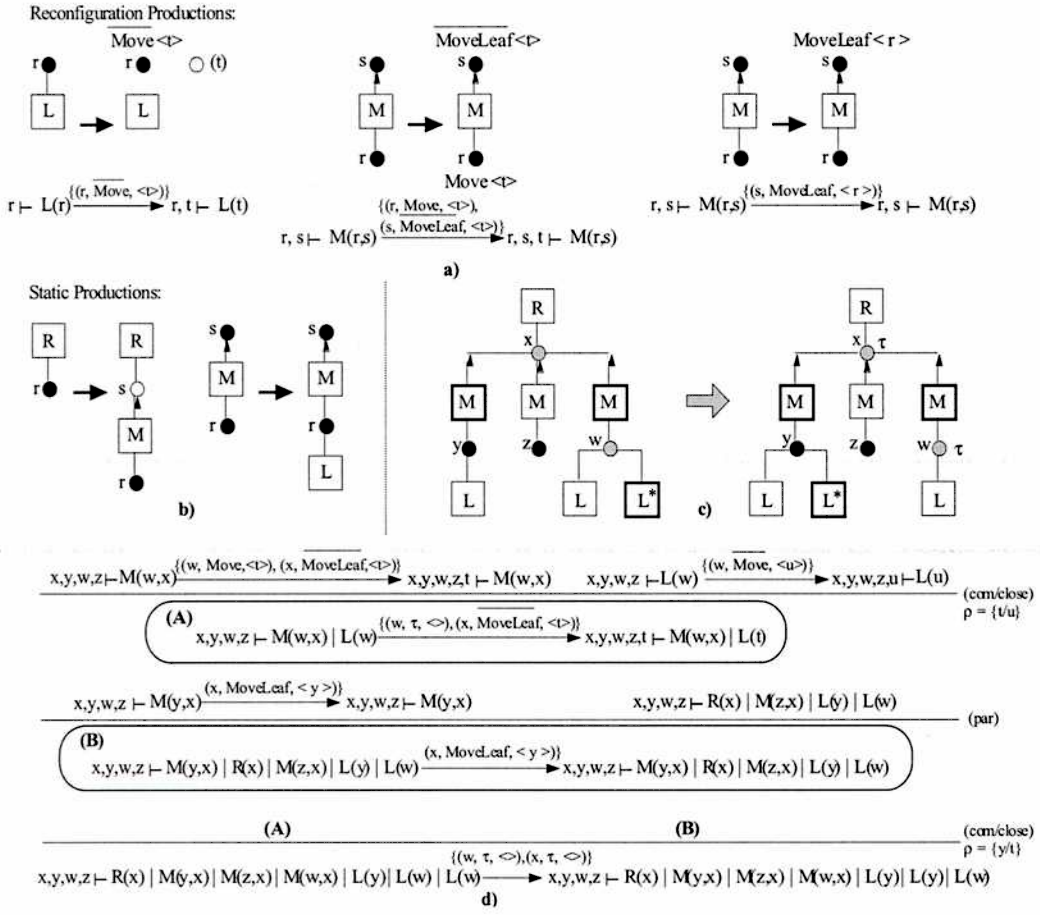

Figure 6.3: Milner Synchronization Example

6.4 Two Examples of Expressive Power

This section presents two examples to show a glimpse of the expressive power obtained by the addition of name mobility to SHR systems.

The first example is the generation of complete graphs and the second example is the generation of $n \times n$ square tiles (or grids). Both examples use Hoare synchronization. These examples are usual examples to measure expressive power. Specially, the tiles example is used by many authors in the area of graph transformation systems.

6.4.1 Complete Graphs

A complete graph is a graph where for each pair of nodes there is a connection between them. In this example directed graphs are used but it is very simple to modify the grammar to include edges in both directions. This example shows a grammar where each synchronized rewriting step generates the next complete graph. Figure 6.4 shows the two productions that are used to generate graphs. The initial graph is the complete graph of two nodes.

The idea is to impose a total order among nodes where in each step the new node *(x)* is the new maximum of the graph. In this way this node is the only one where productions can synchronize (using constraint $a < x >$). With this, only the $n-1$ (with n the number of nodes) edges that are connected to this node are rewritten generating the edges from the older nodes to the new $n+1$ node. Given that for n nodes the degree is $n-1$, it is necessary to distinguish a *director edge* that is in charge (besides adding one of the normal edges like the other ones) to create the missing edge from the actual maximum (node n) to the new one (node $n+1$). Also this edge will be the director for the next step. Figure 6.5

The idea of imposing a total order is a special case of finding an arborescence (a directed tree) in a graph where the maximum corresponds to the root of the tree.

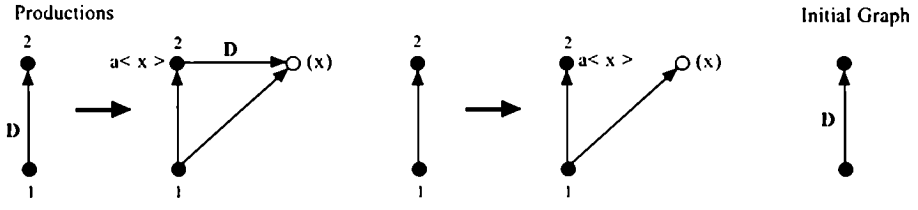

Figure 6.4: Complete Graphs Grammar.

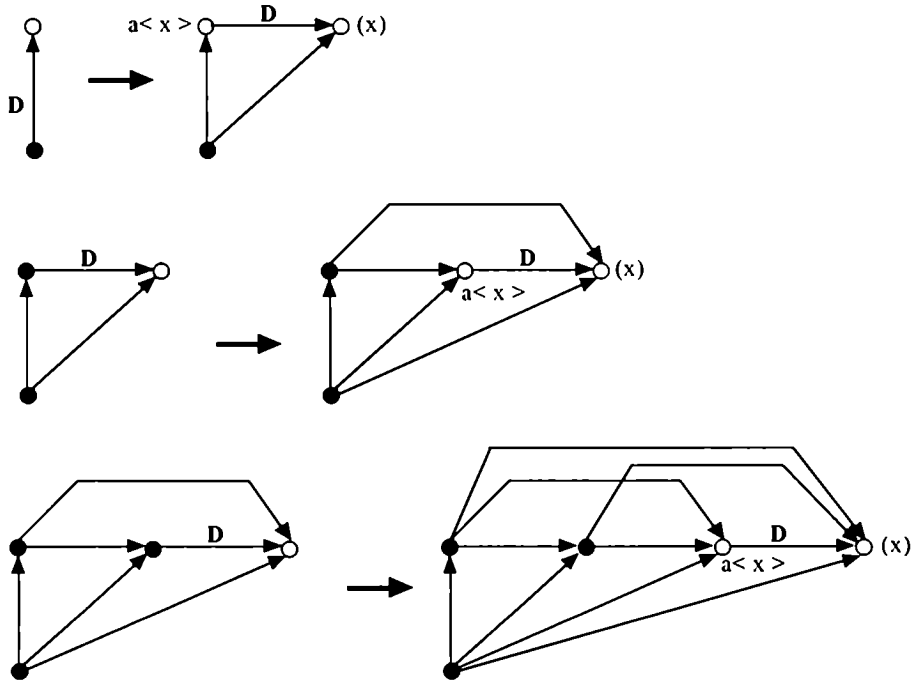

Figure 6.5: Derivation of Complete Graphs.

6.4.2 Tiles

The grammar in Figure 6.6 generates in rewriting step n the $(n+1) \times (n+1)$ *tile*. The idea is similar to the previous example. Edges are oriented towards the right and the bottom side. In this case synchronization will only occur on the nodes of the right and bottom sides. Only the edges on these sides (namely V , H , DV and DH) and the internal edges connected to these nodes are rewritten. Each of the V and H edges will be in charge of creating part of a new cell from rows and columns 1 to $n-1$. As with the complete graph we have to distinguish two director edges (one vertical DV and one horizontal DH), that are in charge of completing cells for column and row n, and the new $n+1$ extreme bottom cell. The two last productions allow for the internal edges to agree on the synchronization. The initial graph is the 1×1 tile. Figure 6.7 shows the first three derivation steps.

In both grammars synchronous termination for reaching a terminal graph can be obtained adding the corresponding productions and terminal symbols.

It is worth noticing that both examples have been done using Hoare synchronization, which it is the basic kind of synchronization, but that the Tiles example can also be done in a similar way with Milner synchronization. In this case only four productions are needed (very similar to the ones in the example, except that the last two for internal edges are not used). On the contrary, the Complete Graph example cannot be done using Milner synchronization.

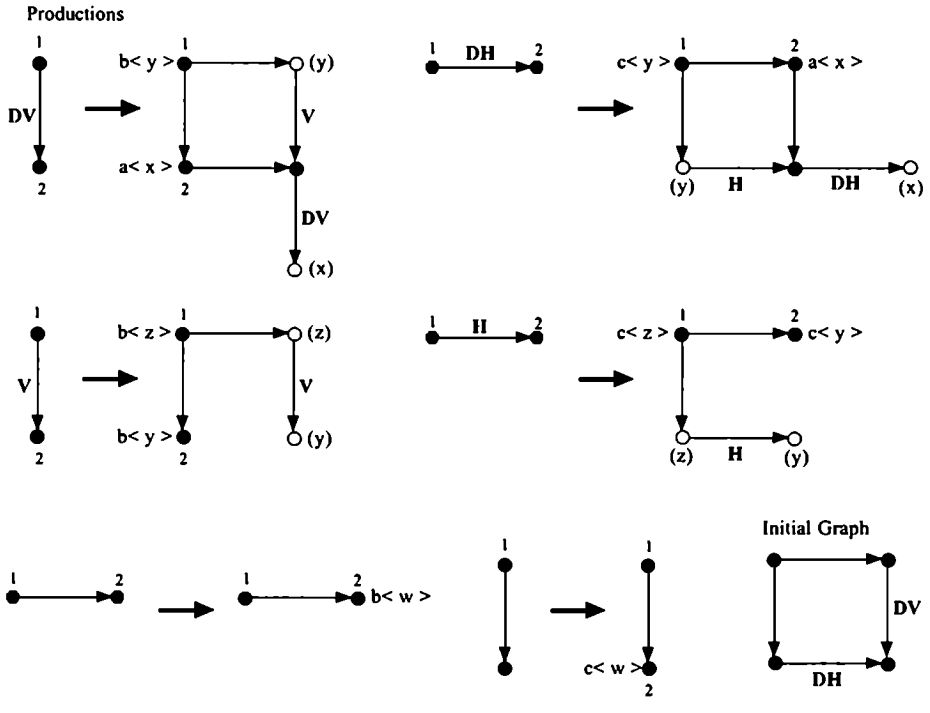

Figure 6.6: N × N Tiles Grammar.

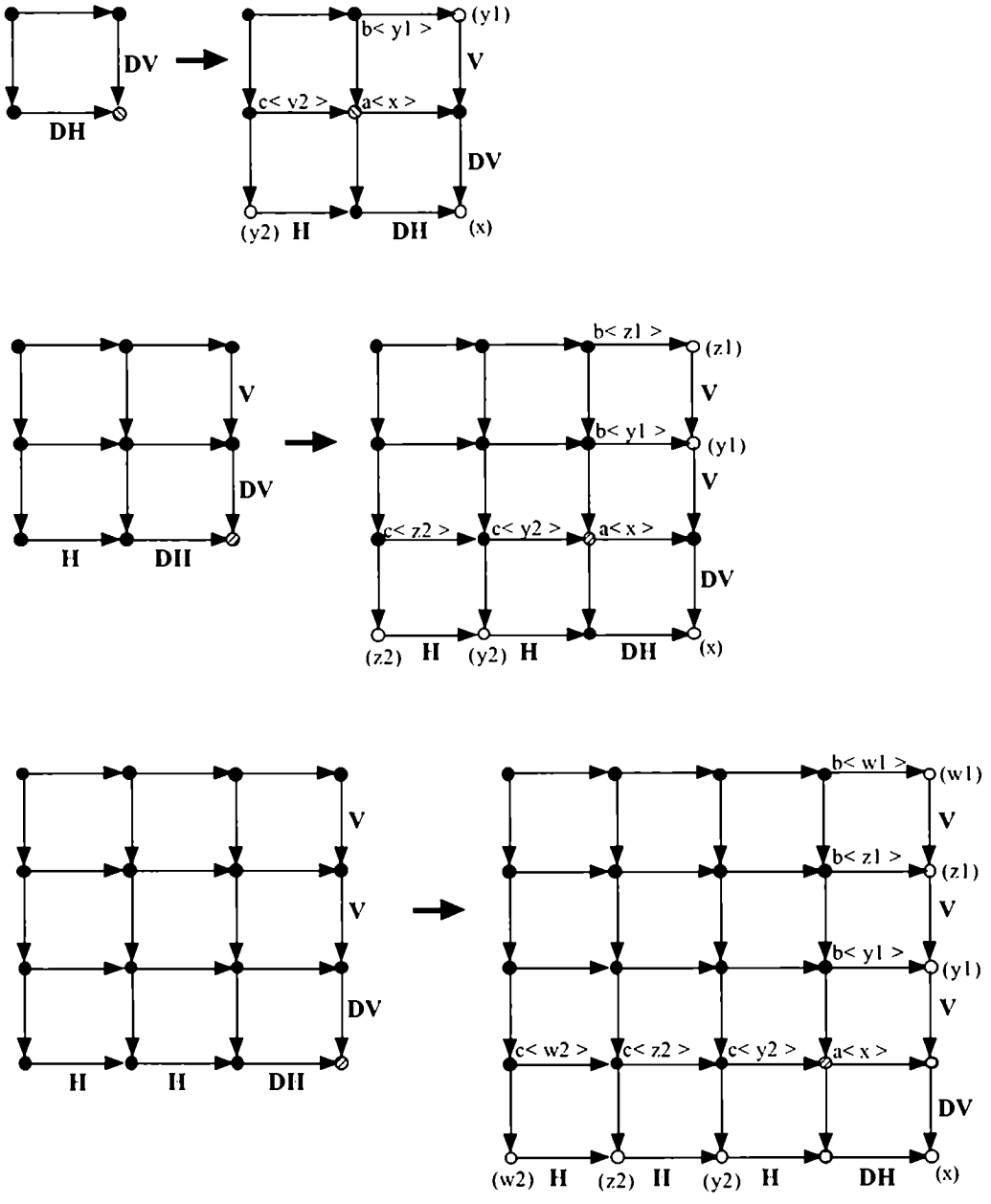

Figure 6.7: Derivation of Tiles.

Part V

A Translation for π-Calculus

With the goal of studying the expressive power of the formalism presented in Part IV we are presenting in Chapter 7 a translation for π-calculus using SHR Systems with Milner synchronization.

We consider this correspondence result as fundamental for the thesis given the relevance of π-calculus for process calculi specially for mobility languages. The fact that we obtain a graphical calculus that subsumes π-calculus is a strong support for graphical formal languages as a next step in the high-level description of distributed, concurrent and mobile systems.

Chapter 7

SHR Systems vs. π-Calculus

The π-calculus [Milner, R., 1999; Sangiorgi, D. and Walker, D., 2001] is a name passing process algebra. Many different versions of the π-calculus have appeared in the literature. The π-calculus we present here is synchronous, *monadic*, guarded recursion and guarded sum. For a definition of π-calculus and its operational semantics see Section 2.4.

For this example we will use the *late operational semantics* of π-calculus and its corresponding transition system.

7.1 Translation

Now we present a translation function from π-calculus to SHR Systems with Milner synchronization and state the correspondence theorems.

At this point, it is necessary to comment on the differences between π-calculus and synchronized rewriting that will affect the definition of the translation from one to the other. For the π-calculus we have an interleaving operational semantics that allows only a sequential evolution of agents, i.e. only one action is allowed at a time. On the other side, we are using graph transformation with synchronized rewriting which is a distributed concurrent model allowing for multiple and simultaneous synchronizations and rewriting. In spite of the fact that the translation function that we are defining in Section 7.1 does not allows multiple synchronization on one edge, it is still possible to have concurrent independent transition steps. So, to avoid this and be able to prove the correspondence of π-calculus with the more expressive universe of graph rewriting, we impose a simple condition over rule *(com/close)* of the Milner Transition System (see Table 6.2). The condition for this rule states that, $|\Lambda_1| = |\Lambda_2| = |m(\Lambda_1 \cup \Lambda_2)|$.

The above condition together with the translation restriction imposed by the "sequentiality" of π-calculus give that for any rule, applied in a proof under these conditions, it cannot be the case that $|\Lambda| > 1$. Therefore, for the rest of the section we will use the simplified version of Milner Transition System of Table 7.1. We refer to this system as the $Milner_\pi$ Transition System or $Milner_\pi$ Synchronization. For simplicity in the

$$(ren_\pi)\ \frac{\Gamma \vdash G \xrightarrow{\Lambda} \Gamma, \Delta \vdash G' \in \mathcal{P}}{\Gamma', \xi\Gamma \vdash \xi G \xrightarrow{\xi\Lambda} \Gamma', \xi\Gamma, \xi\Delta \vdash \xi G'}$$

where $\Lambda = \{(v, out, < w >)\}$, $\Delta = \emptyset$ or $\Lambda = \{(v, in, < w >)\}$, $\Delta = \{w\}$

$$(com_\pi)\ \frac{\Gamma \vdash G_1 \xrightarrow{\{(x,out,<y>)\}} \Gamma \vdash G'_1 \quad \Gamma \vdash G_2 \xrightarrow{\{(x,in,<z>)\}} \Gamma, z \vdash G'_2}{\Gamma \vdash G_1|G_2 \xrightarrow{\{(x,\tau,<>)\}} \Gamma \vdash G'_1|G'_2\{y/z\}}$$

where $\rho_{\{(x,out,<y>)\}\cup\{(x,in,<z>)\}} = \{y/z\}$

$$(close_\pi)\ \frac{\Gamma \vdash G_1 \xrightarrow{\{(x,out,<y>)\}} \Gamma, y \vdash G'_1 \quad \Gamma \vdash G_2 \xrightarrow{\{(x,in,<y>)\}} \Gamma, y \vdash G'_2}{\Gamma \vdash G_1|G_2 \xrightarrow{\{(x,\tau,<>)\}} \Gamma \vdash \nu y.(G'_1|G'_2)}$$

$$(par_\pi)\ \frac{\Gamma \vdash G_1 \xrightarrow{\Lambda} \Gamma, \Delta \vdash G'_1 \quad \Gamma \vdash G_2}{\Gamma \vdash G_1|G_2 \xrightarrow{\Lambda} \Gamma, \Delta \vdash (G'_1|G_2)}$$

where $\Lambda = \emptyset \mid \{(v, out, < w >)\}^{|}{}_{|} \{(v, in, < w >)\} \mid \{(v, \tau, <>)\}$

$$(res_\pi)\ \frac{\Gamma, x \vdash G \xrightarrow{\Lambda} \Gamma, x, \Delta \vdash G'}{\Gamma \vdash \nu x.G \xrightarrow{\Lambda} \Gamma, \Delta \vdash \nu x.G'}\ x \notin n(\Lambda),\ \Lambda(x)\uparrow$$

where $\Lambda = \emptyset \mid \{(v, out, < w >)\}^{|}{}_{|} \{(v, in, < w >)\} \mid \{(v, \tau, <>)\}$ with $v, w \neq x$

$$\frac{\Gamma, x \vdash G \xrightarrow{\{(x,\tau,<>)\}} \Gamma, x, \Delta \vdash G'}{\Gamma \vdash \nu x.G \xrightarrow{\emptyset} \Gamma, \Delta \vdash \nu x.G'}$$

$$(open_\pi)\ \frac{\Gamma, y \vdash G \xrightarrow{\{(x,out,<y>)\}} \Gamma, y \vdash G'}{\Gamma \vdash \nu y.G \xrightarrow{\{(x,out,<y>)\}} \Gamma, y \vdash G'}\ y \neq x$$

Table 7.1: Transition Rules for $Milner_\pi$ Synchronization

1. $[\![nil]\!]_\Gamma = \Gamma \vdash nil$

2. $[\![rec\,X.\,P]\!]_\Gamma = [\![P\,[rec\,X.\,P/X]]\!]_\Gamma$

3. $$\frac{[\![P]\!]_\Gamma = \Gamma \vdash G_P \quad [\![Q]\!]_\Gamma = \Gamma \vdash G_Q}{[\![P|Q]\!]_\Gamma = \Gamma \vdash G_P|G_Q}$$

4. $$\frac{[\![P]\!]_{\Gamma,x} = \Gamma, x \vdash G}{[\![\nu x.\,P]\!]_\Gamma = \Gamma \vdash \nu x.G}$$

5. $$\frac{P = \sigma_P \widehat{P} \;\; sequential\ standard,\ fn(P) \subseteq \Gamma}{[\![P]\!]_\Gamma = \Gamma \vdash \sigma_P L_{\widehat{P}}(v_1, \ldots, v_{|fn(\widehat{P})|})}$$

Productions (Late)

6. $$\frac{P \stackrel{\bar{v}w}{\longmapsto} Q \quad P\ sequential\ standard\ agent}{[\![P]\!]_{\mathrm{fn}(P)} \xrightarrow{\{(v,out,<w>)\}} [\![Q]\!]_{\mathrm{fn}(P)}}$$

7. $$\frac{P \stackrel{v(w)}{\longmapsto} Q \quad P\ sequential\ standard\ agent\ with\ w\ a\ fixed\ name,\ w \notin fn(P)}{[\![P]\!]_{\mathrm{fn}(P)} \xrightarrow{\{(v,in,<w>)\}} [\![Q]\!]_{\mathrm{fn}(P)\cup\{w\}}}$$

Table 7.2: Translation Function for π-calculus

treatment of the cases, we separate rule *(com/close)* in rules (com_π) and ($close_\pi$), and rule *(res)* in rules (res_π) and ($open_\pi$) .

DEFINITION 7.1. [Translation Function for π-Calculus] Let $\mathcal{N}$ be a fixed infinite set of names. We define a translation function $[\![P]\!]_\Gamma$ for π-calculus agents, with respect to a set of names ($\Gamma \subset \mathcal{N}$), as follows:

1. Any agent term for agent P is chosen.
2. The agent term is translated by structural recursion using the definition in Table 7.2.

The edges that are created by the translation have labels that correspond to the standard agents of the uppermost level of sequential subterms of the agent term to be translated. Two agents with the same standard agent will produce the same label, with its rank corresponding to the number of free variables of the standard agent (which by definition is the maximum number of distinguishable free variables occurrences, see Definition 2.10). The standard substitution defines the attachment nodes of each edge with that label. This means that if a variable occurs n times in a sequential agent, the

corresponding translated edge will have n tentacles attached to the same node in the graph.

The use the standard decomposition is fundamental to correctly define the correspondence between agents and graphs. Productions are defined over edge labels and have to be applied over different edges with the same label (which corresponds to a sequential subagent). The variable names in an agent define the connections among its different subagents, and the evolution of an agent implies the possible substitution of names. If we do not use the standard decomposition, then the translation of an agent will produce labels containing variable names that may change with each rewriting step ending up with an infinite number of edge labels (i.e. a grammar with infinite productions) and two equivalent agents may be translated to two non isomorphic graphs. This includes the case when the renaming implies the matching of two previously different names (i.e. two channels are connected together reducing the number of free variables in the agent) which may change also the rank (i.e. number of tentacles) of the labels.

A transition in the π-calculus will be represented as a transition of the corresponding translated judgement (i.e. a rewriting step). Productions are generated based on the possible transitions of these sequential agents with the corresponding label as the left hand side of the production and the target agent as the right hand side. Actions are translated as requirements on the transitions. For π-calculus we have one action for inputs (*in*) and one coaction for outputs (*out*). Then, the evolution of the agent is modeled using the Milner_{π} transition system introduced in table 7.1. Note that synchronization of π-calculus agents can happen only between sequential agents, and this corresponds exactly to the result of synchronization among edges.

All the above intuitive correspondence presentation is what is formally proved in the rest of this section ending with Theorem 7.20.

To close the definition of the translation function the following theorem proves that $[\![-]\!]_\Gamma$ is well defined.

THEOREM 7.2. *The translation function $[\![-]\!]_\Gamma$ is well defined. For any agent terms P and Q, $P = Q \Rightarrow [\![P]\!]_\Gamma = [\![Q]\!]_\Gamma$.*

Proof Sketch. *The theorem is proved by induction on the structural axioms of π-calculus.*

- *For α-conversion it is obvious by graph axiom $(AG6)$.*
- *For (par) it is obvious by graph axioms (AG_1), (AG_2) and (AG_3).*
- *For (res) it is obvious by graph axioms (AG_4), (AG_5) and (AG_7).*
- *For (rec) it is obvious by definition of $[\![recX.p]\!]_\Gamma$.* □

The next step is to prove the bijective correspondence of agents and judgements (i.e. graphs). This is done in Theorem 7.5, but first we need to define the following inverse function for judgements.

DEFINITION 7.3. We define {[-]}, as a translation function for graphs (with agent terms as labels) to π-calculus agents.

1. $\{[\Gamma \vdash nil]\} = nil$.
2. $\{[\Gamma \vdash \sigma_P L_{\widehat{P}}(v_1, \ldots, v_{|fn(\widehat{P})|})]\} = \sigma_P \widehat{P}$ with $fn(\sigma_P \widehat{P}) \subseteq \Gamma$.
3. $\{[\Gamma \vdash G_1 | G_2]\} = \{[\Gamma \vdash G_1]\} | \{[\Gamma \vdash G_2]\}$.
4. $\{[\Gamma \vdash \nu x.G]\} = \nu x.\{[\Gamma, x \vdash G]\}$.

THEOREM 7.4. *The translation function $\{[-]\}$ is well defined. For any graphs $\Gamma \vdash G_1$ and $\Gamma \vdash G_2$, $\Gamma \vdash G_1 = \Gamma \vdash G_2 \Rightarrow \{[\Gamma \vdash G_1]\} = \{[\Gamma \vdash G_2]\}$.*

Proof Sketch. *The theorem is proved by induction on the structural axioms of graphs, as it was done for $[\![-]\!]_\Gamma$, matching the corresponding structural axioms of π-calculus. π-calculus.* □

THEOREM 7.5. [Correspondence of π-calculus agents and syntactic judgements] *There is a bijective correspondence, with respect to the translation function $[\![-]\!]_\Gamma$, of π-calculus agents (up to structural axioms) and well-formed syntactic judgements with sequential standard agents as labels (up to structural axioms and isolated nodes).*

Proof. *To prove this theorem we have to prove that:*

1. $\{[[\![P]\!]_\Gamma]\} = P$.
2. $[\![\{[\Gamma \vdash G]\}]\!]_\Gamma = \Gamma \vdash G$.

We restrict Γ to $\mathrm{fn}(P)$ and the names used in G. Any other $\Gamma' \supseteq \Gamma$, corresponds to isolated nodes as the only difference in the translation.

1. $\{[[\![P]\!]_{\mathrm{fn}(P)}]\} = P$ *? It is proved by structural induction.*
   Base Case:
   - *For* $\{[[\![nil]\!]_\emptyset]\} = \{[\emptyset \vdash nil]\} = nil$.
   - *For* $P = \sigma_P \widehat{P}$ *sequential standard,*

$$\{[[\![P]\!]_{\mathrm{fn}(P)}]\} = \{[\mathrm{fn}(P) \vdash \sigma_P L_{\widehat{P}}(v_1, \ldots, v_{|fn(\widehat{P})|})]\} = \sigma_P \widehat{P}$$

   - *We have by Inductive Hypothesis that,* $\{[[\![P]\!]_{\mathrm{fn}(P)}]\} = P$ *and* $\{[[\![Q]\!]_{\mathrm{fn}(Q)}]\} = Q$. *This implies that (with* $\Gamma = \mathrm{fn}(P) \cup \mathrm{fn}(Q) = \mathrm{fn}(P|Q)$*),* $\{[[\![P]\!]_\Gamma]\} = P$ *and* $\{[[\![Q]\!]_\Gamma]\} = Q$. *Then by definition,*

$$P|Q = \{[[\![P]\!]_\Gamma]\} | \{[[\![Q]\!]_\Gamma]\} = \{[[\![P|Q]\!]_\Gamma]\}$$

- *We have by Inductive Hypothesis that,* $\{\![[\![P]\!]_{\mathrm{fn}(P)}]\!\} = P$. *Then by definition of* $\{\![-]\!\}$ *and* $[\![-]\!]$,

$$\begin{aligned} \nu x.\{\![[\![P]\!]_{\mathrm{fn}(P)}]\!\} &= \nu x.P \\ \{\![[\![\nu x.P]\!]_{\mathrm{fn}(P)/x}]\!\} &= \nu x.\{\![[\![P]\!]_{\mathrm{fn}(P)}]\!\} \end{aligned}$$

- *We have by Inductive Hypothesis and axiom* (rec) *that,*

$$\begin{aligned} \{\![[\![rec\,X.\,P]\!]_{\mathrm{fn}(rec\,X.\,P)}]\!\} = \{\![[\![P[rec\,X.\,P/X]]\!]_{\mathrm{fn}(rec\,X.\,P)}]\!\} &= P[rec\,X.\,P/X] \\ P[rec\,X.\,P/X] &= rec\,X.\,P \end{aligned}$$

2. $[\![\{\!\{\Gamma \vdash G\}\!\}]\!]_\Gamma = \Gamma \vdash G$ *? It is proved by induction.*

   Base Case:

   - *For* $[\![\{\!\{\emptyset \vdash nil\}\!\}]\!]_\emptyset = [\![nil]\!]_\emptyset = \emptyset \vdash nil$.
   - *For* $P = \sigma_P \widehat{P}$ *sequential standard,*

$$\begin{aligned} &[\![\{\!\{\mathrm{fn}(P) \vdash \sigma_P L_{\widehat{P}}(v_1, \ldots, v_{|fn(\widehat{P})|})\}\!\}]\!]_{\mathrm{fn}(P)} \\ &\quad = [\![\sigma_P \widehat{P}]\!]_{\mathrm{fn}(P)} = \mathrm{fn}(P) \vdash \sigma_P L_{\widehat{P}}(v_1, \ldots, v_{|fn(\widehat{P})|}) \end{aligned}$$

   - *We have by Inductive Hypothesis that,* $[\![\{\!\{\Gamma \vdash G_1\}\!\}]\!]_\Gamma = \Gamma \vdash G_1$ *and* $[\![\{\!\{\Gamma \vdash G_2\}\!\}]\!]_\Gamma = \Gamma \vdash G_2$. *Then by definition of* $[\![-]\!]$ *and* $\{\![-]\!\}$,

$$\begin{aligned} \Gamma \vdash G_1|G_2 &= [\![\{\!\{\Gamma \vdash G_1\}\!\}|\{\!\{\Gamma \vdash G_2\}\!\}]\!]_\Gamma \\ [\![\{\!\{\Gamma \vdash G_1|G_2\}\!\}]\!]_\Gamma &= [\![\{\!\{\Gamma \vdash G_1\}\!\}|\{\!\{\Gamma \vdash G_2\}\!\}]\!]_\Gamma \end{aligned}$$

   - *We have by Inductive Hypothesis that,* $[\![\{\!\{\Gamma, x \vdash G\}\!\}]\!]_{\Gamma,x} = \Gamma, x \vdash G$. *Then by definition of* $\{\![-]\!\}$ *and* $[\![-]\!]$,

$$\begin{aligned} \Gamma \vdash \nu x.G &= [\![\nu x.\{\!\{\Gamma, x \vdash G\}\!\}]\!]_\Gamma \\ [\![\{\!\{\Gamma \vdash \nu x.G\}\!\}]\!]_\Gamma &= [\![\nu x.\{\!\{\Gamma, x \vdash G\}\!\}]\!]_\Gamma \end{aligned}$$

□

Now that the correspondence between agents and judgements has been proved, Theorem 7.16 shows that the derivations of any graph reachable from an initial graph G_0 (obtained by the translation) use productions from a finite set.

DEFINITION 7.6. For any agent term P, we define $S(P)$ as the set of standard sequential subagents of P.

$$\begin{aligned} S'(nil) &= \emptyset \\ S'\left(\sum_{i=1}^{n} \alpha_i.P_i\right) &= \left\{\sum_{i=1}^{n} \alpha_i.P_i\right\} \cup \bigcup_{i=1}^{n} S'(P_i) \\ S'(rec\,X.\,P) &= S'(P)[rec\,X.\,P/X] \\ S'(X) &= \emptyset \\ S'(P_1|P_2) &= S'(P_1) \cup S'(P_2) \\ S'(\nu x.\,P) &= S'(P) \\ S(P) &= \widehat{S'(P)} \end{aligned}$$

The definition of standard decomposition is extended over sets as usual.

DEFINITION 7.7. For any graph obtained by translation function $[\![-]\!]$, we define $Seq(G)$ as the set of standard sequential subagents of the agents in the labels of G.

$$Seq([\![P]\!]_\Gamma) = S(\{[\![P]\!]_\Gamma\})$$

PROPOSITION 7.8. *Functions $S(P)$ and $Seq(G)$ are well defined:*

1. *$P = Q$ implies $S(P) = S(Q)$.*
2. *$G_1 = G_2$ implies $Seq(G_1) = Seq(G_2)$.*

Proof Sketch. *By induction on the structural axioms. Obvious by definition of $S(P)$ and $Seq(G)$.* □

DEFINITION 7.9. [Finite Derivation] We define the notion of finite derivation (noted as $\Rightarrow$) over transitions as:

$$\frac{\Gamma_1 \vdash G_1 \xrightarrow{\Lambda} \Gamma_2 \vdash G_2}{\Gamma_1 \vdash G_1 \Rightarrow \Gamma_2 \vdash G_2}$$

$$\frac{\Gamma_1 \vdash G_1 \xrightarrow{\Lambda} \Gamma_2 \vdash G_2 \quad \Gamma_2 \vdash G_2 \Rightarrow \Gamma_3 \vdash G_3}{\Gamma_1 \vdash G_1 \Rightarrow \Gamma_3 \vdash G_3}$$

DEFINITION 7.10. [Proof Term] We define the notion of proof term (using connector //) as:

$$\frac{r : \{x_i\}/y \in R \quad d_i//x_i}{r(\{d_i\})//y}$$

Set R is a set of inference rules, $r : \{x_i\}/y$ is a rule with name r and preconditions $\{x_i\}$ and result y. Terms $d_i//x_i$ are the corresponding proof terms for the preconditions of y. Finally, $r(\{d_i\})//y$ is the proof term for y.

DEFINITION 7.11. For any derivation proof term D we define the set $Prod(D)$ of productions it uses as:

$$\begin{aligned} Prod\left(\emptyset//\Gamma_1 \vdash L_P \xrightarrow{\Lambda} \Gamma_2 \vdash G\right) &= \left\{\Gamma_1 \vdash L_P \xrightarrow{\Lambda} \Gamma_2 \vdash G\right\} \\ Prod\,(r(\{d_i\})//y) &= \bigcup_i Prod(d_i//x_i) \end{aligned}$$

The proof system for finite derivations includes rules in Definition 7.9 and transition rules for Milner$_\pi$ synchronization including the production axiom (see table 7.1). Set $Prod$ is defined over the proof system for finite derivations.

LEMMA 7.12. *$P \xrightarrow{\alpha} Q$ implies $S(Q) \subseteq S(P)$.*

Proof. *We proceed by induction on rules for π-calculus transition system.*

Base Case:

- *For (Sum),* $S\left(\sum_{i=1}^n \alpha_i.P_i\right) = \{\widehat{\sum_{i=1}^n \alpha_i.P_i}\} \cup \bigcup_{i=1}^n \widehat{S'(P_i)} \supseteq \widehat{S'(P_j)} = S(P_j)$.
- *For (Par),* $S(P|Q) = S(P) \cup S(Q) \overset{IH}{\supseteq} S(P') \cup S(Q) = S(P'|Q)$.
- *For (Com), by definition of standard decomposition we have that for any sequential agent P and substitution γ, $\widehat{P} = \widehat{\gamma P}$. Then,* $S(P|Q) = S(P) \cup S(Q) \overset{IH}{\supseteq} S(P') \cup S(Q') = S(P') \cup S(Q'\{y/z\}) = S(P'|Q'\{y/z\})$.
- *For (Close),* $S(P|Q) = S(P) \cup S(Q) \overset{IH}{\supseteq} S(P') \cup S(Q') = S(P'|Q') = S(\boldsymbol{\nu} y.\,(P'|Q'))$.
- *For (Open),* $S(\boldsymbol{\nu} y.\,P) = S(P) \overset{IH}{\supseteq} S(P')$.
- *For (Res),* $S(\boldsymbol{\nu} x.\,P) = S(P) \overset{IH}{\supseteq} S(P') = S(\boldsymbol{\nu} x.\,P')$.
- *For (Cong),* $S(P') = S(P) \overset{IH}{\supseteq} S(Q) = S(Q')$. □

LEMMA 7.13. *$S(P) = Seq(\llbracket P \rrbracket_\Gamma)$ $\text{fn}(P) \subseteq \Gamma$.*

Proof. *Obvious by definition of Seq and Theorem 7.5.* □

LEMMA 7.14. $\Gamma_1 \vdash G_1 \xrightarrow{\Lambda} \Gamma_2 \vdash G_2$ *implies* $Seq(\Gamma_2 \vdash G_2) \subseteq Seq(\Gamma_1 \vdash G_1)$.

Proof. *We proceed by induction on rules for Milner$_\pi$ transition system.*

Base Case:

- *For productions, it is obvious by definition of* $[\![-]\!]_\Gamma$ *and Lemmas 7.12 and 7.13.*
- *For rules* (ren_π), (res_π) *and* $(open_\pi)$, *it is obvious since* $lab(G_1) = lab(G_1')$, $lab(G_2) = lab(G_2')$ *and they are of the form,*

$$\frac{\Gamma_1 \vdash G_1 \xrightarrow{\Lambda} \Gamma_2 \vdash G_2}{\Gamma_1' \vdash G_1' \xrightarrow{\Lambda} \Gamma_2' \vdash G_2'}$$

- *For rule* (par_π), *we have that* $lab(\Gamma, \Delta \vdash G_2) = lab(\Gamma \vdash G_2)$ *and by Inductive Hypothesis that* $Seq(\Gamma, \Delta \vdash G_1') \subseteq Seq(\Gamma \vdash G_1)$. *Then,* $Seq(\Gamma, \Delta \vdash G_1') \cup Seq(\Gamma, \Delta \vdash G_2) \subseteq Seq(\Gamma \vdash G_1) \cup Seq(\Gamma \vdash G_2)$.
- *For rule* (com_π) *we want,*

$$\Gamma \vdash G_1|G_2 \xrightarrow{\{(x,r,<>)\}} \Gamma \vdash G'_1|G'_2\{y/z\} \quad implies$$
$$Seq(\Gamma \vdash G'_1|G'_2\{y/z\}) \quad \subseteq \quad Seq(\Gamma \vdash G_1|G_2).$$

Then by definition of Seq,

$$Seq(\Gamma \vdash G'_1\{y/z\}) \cup Seq(\Gamma \vdash G'_2\{y/z\}) \quad \subseteq \quad Seq(\Gamma \vdash G_1) \cup Seq(\Gamma \vdash G_2).$$

which is true by Inductive Hypothesis and that $lab(G_1') = lab(G_1'\{y/z\})$, $lab(G_2') = lab(G_2'\{y/z\})$.

- *For rule* $(close_\pi)$ *it is similar to rule* (com_π). □

LEMMA 7.15. *We Prove the following properties for Seq.*

1. $Seq(\Gamma \vdash nil) = \emptyset$.
2. $Seq(\Gamma \vdash L_P(\vec{x})) = S(P)$ *with P sequential standard.*
3. $Seq(\Gamma \vdash G_1|G_2) = Seq(\Gamma \vdash G_1) \cup Seq(\Gamma \vdash G_2)$.
4. $Seq(\Gamma \vdash \nu x.G) = Seq(\Gamma, x \vdash G)$.

Proof.

1. $Seq(\Gamma \vdash nil) = Seq([\![nil]\!]_\Gamma) = S(\{[\![[\![nil]\!]_\Gamma]\!]\}) = \emptyset$.
2. $Seq(\Gamma \vdash L_P(\vec{x})) = S(P), by Lemma\ 7.13$ *with P sequential standard.*

3. *By Translation and Theorem 7.5,* $Seq(\Gamma \vdash G_1|G_2) = S(P|Q)$ *with* $[\![P]\!]_\Gamma = \Gamma \vdash G_1$ *and* $[\![Q]\!]_\Gamma = \Gamma \vdash G_2$. *Then,* $S(P|Q) = S(P) \cup S(Q) = Seq(\Gamma \vdash G_1) \cup Seq(\Gamma \vdash G_2)$ *by Lemma 7.13.*

4. *By Translation and Theorem 7.5,* $Seq(\Gamma \vdash \nu x.G) = S(\boldsymbol{\nu} x.P) = S(P) = Seq(\Gamma, x \vdash G)$ *with* $[\![P]\!]_{\Gamma,x} = \Gamma, x \vdash G$. □

THEOREM 7.16. *The evolution of any graph* $\Gamma \vdash G_0$ *can be described by a finite set of productions.*

Proof. *The Proof has three steps:*

1. $Prod\,(d//\Gamma_0 \vdash G_0 \Rightarrow \Gamma \vdash G) \subseteq \left\{\Gamma \vdash L_P \xrightarrow{\Lambda} \Gamma' \vdash G' \mid P \in Seq(G_0)\right\}$.

2. *For all* L_P, *an edge label, the set* $\{\Gamma \vdash L_P \xrightarrow{\Lambda} \Gamma' \vdash G'\}$ *is finite.*

3. $Seq(G_0)$ *is finite.*

Point 1) says that the set of productions used as axioms for a finite derivation proof starting from a graph G_0, *only corresponds to productions where the edge label in the left hand side corresponds to some sequential standard agent in set* $Seq(G_0)$. *Points 2) says that for any edge label* L_P *of the corresponding signature, the set of possible productions that can be obtained as transitions from edges labelled by* L_P *is finite. Finally, to complete the proof point 3) says that the set of sequential standard subagents from which come all edge labels that are used in a derivation (i.e.* $Seq(G_0)$. *is finite.*

1. *It is proved by computation induction on the proof of derivations:*

   Base Case:

   - *For* (ren_π), *we know by definition that*

$$\begin{array}{rcl} Q & \in & Seq(\Gamma \vdash L_Q) \\ Prod\left(\emptyset//\Gamma \vdash L_Q \xrightarrow{\Lambda} \Gamma, \Delta \vdash G\right) & \subseteq & \left\{L_P \xrightarrow{\Lambda'} G' \mid P \in Seq(\Gamma \vdash L_Q)\right\} \end{array}$$

$$Prod\left(r(d_1)//\Gamma', \xi\Gamma \vdash \xi L_Q \xrightarrow{\xi\Lambda} \Gamma', \xi\Gamma, \xi\Delta \vdash \xi G\right) \stackrel{def}{=} Prod\left(\emptyset//\Gamma \vdash L_Q \xrightarrow{\Lambda} \Gamma, \Delta \vdash G\right)$$

   Then, given that Q *is sequential standard and by definition* $\widehat{\gamma Q} = Q$, *we have* $Seq(\Gamma \vdash L_Q) = Seq(\Gamma', \xi\Gamma \vdash \xi L_Q)$ *and,*

$$Prod\left(r(d_1)//\Gamma', \xi\Gamma \vdash \xi L_Q \xrightarrow{\xi\Lambda} \Gamma', \xi\Gamma, \xi\Delta \vdash \xi G\right) \subseteq \left\{L_P \xrightarrow{\Lambda'} G' \mid P \in Seq(\Gamma', \xi\Gamma \vdash \xi L_Q)\right\}$$

- *For rule,*

$$\frac{\Gamma_1 \vdash G_1 \xrightarrow{\Lambda} \Gamma_2 \vdash G_2 \quad \Gamma_2 \vdash G_2 \Rightarrow \Gamma_3 \vdash G_3}{\Gamma_1 \vdash G_1 \Rightarrow \Gamma_3 \vdash G_3}$$

We know that,

$$\begin{aligned} Prod\left(d_1//\Gamma_1 \vdash G_1 \xrightarrow{\Lambda} \Gamma_2 \vdash G_2\right) &\overset{IH}{\subseteq} \left\{L_P \xrightarrow{\Lambda'} G' \mid P \in Seq(\Gamma_1 \vdash G_1)\right\} \\ Prod\left(d_2//\Gamma_2 \vdash G_2 \Rightarrow \Gamma_3 \vdash G_3\right) &\overset{IH}{\subseteq} \left\{L_P \xrightarrow{\Lambda'} G' \mid P \in Seq(\Gamma_2 \vdash G_2)\right\} \end{aligned}$$

$$\begin{aligned} &Prod\left(r(d_1,d_2)//\Gamma_1 \vdash G_1 \Rightarrow \Gamma_3 \vdash G_3\right) \overset{def}{=} \\ &\quad Prod\left(d_1//\Gamma_1 \vdash G_1 \xrightarrow{\Lambda} \Gamma_2 \vdash G_2\right) \cup Prod\left(d_2//\Gamma_2 \vdash G_2 \Rightarrow \Gamma_3 \vdash G_3\right) \end{aligned}$$

Then, by Lemma 7.14, $\Gamma_1 \vdash G_1 \xrightarrow{\Lambda} \Gamma_2 \vdash G_2$ *implies* $Seq(G_2) \subseteq Seq(G_1)$. *And this implies,*

$$\left\{L_P \xrightarrow{\Lambda'} G' \mid P \in Seq(\Gamma_1 \vdash G_1)\right\} \subseteq \left\{L_P \xrightarrow{\Lambda'} G' \mid P \in Seq(\Gamma_2 \vdash G_2)\right\}$$

Then we have,

$$\begin{aligned} &Prod\left(d_1//\Gamma_1 \vdash G_1 \xrightarrow{\Lambda} \Gamma_2 \vdash G_2\right) \cup Prod\left(d_2//\Gamma_2 \vdash G_2 \Rightarrow \Gamma_3 \vdash G_3\right) \subseteq \\ &\left\{L_P \xrightarrow{\Lambda'} G' \mid P \in Seq(\Gamma_1 \vdash G_1)\right\} \cup \left\{L_P \xrightarrow{\Lambda'} G' \mid P \in Seq(\Gamma_2 \vdash G_2)\right\}, \end{aligned}$$

which implies

$$Prod\left(r(d_1,d_2)//\Gamma_1 \vdash G_1 \Rightarrow \Gamma_3 \vdash G_3\right) \subseteq \left\{L_P \xrightarrow{\Lambda} G' \mid P \in Seq(\Gamma_1 \vdash G_1)\right\}$$

- *For rule,*

$$\frac{\Gamma_1 \vdash G_1 \xrightarrow{\Lambda} \Gamma_2 \vdash G_2}{\Gamma_1 \vdash G_1 \Rightarrow \Gamma_2 \vdash G_2}$$

We know that,

$$\begin{aligned} Prod\left(d_1//\Gamma_1 \vdash G_1 \xrightarrow{\Lambda} \Gamma_2 \vdash G_2\right) &\overset{IH}{\subseteq} \left\{L_P \xrightarrow{\Lambda'} G' \mid P \in Seq(\Gamma_1 \vdash G_1)\right\} \\ Prod\left(r(d_1)//\Gamma_1 \vdash G_1 \Rightarrow \Gamma_2 \vdash G_2\right) &\overset{def}{=} Prod\left(d_1//\Gamma_1 \vdash G_1 \xrightarrow{\Lambda} \Gamma_2 \vdash G_2\right) \end{aligned}$$

Then,

$$Prod\left(r(d_1)//\Gamma_1 \vdash G_1 \Rightarrow \Gamma_2 \vdash G_2\right) \subseteq \left\{L_P \xrightarrow{\Lambda'} G' \mid P \in Seq(\Gamma_1 \vdash G_1)\right\}$$

- *For* (com_π) *we know that,*

$$Prod\left(d_1//\Gamma \vdash G_1 \xrightarrow{\Lambda_1} \Gamma \vdash G_1'\right) \overset{IH}{\subseteq} \left\{L_P \xrightarrow{\Lambda'} G' \mid P \in Seq(\Gamma \vdash G_1)\right\}$$
$$Prod\left(d_2//\Gamma \vdash G_2 \xrightarrow{\Lambda_2} \Gamma, z \vdash G_2'\right) \overset{IH}{\subseteq} \left\{L_P \xrightarrow{\Lambda'} G' \mid P \in Seq(\Gamma \vdash G_2)\right\}$$

$$Prod\left(r(d_1,d_2)//\Gamma \vdash G_1|G_2 \xrightarrow{\Lambda} \Gamma \vdash G_1'|G_2'\{y/z\}\right) \overset{def}{=}$$
$$Prod\left(d_1//\Gamma \vdash G_1 \xrightarrow{\Lambda_1} \Gamma \vdash G_1'\right) \cup Prod\left(d_2//\Gamma \vdash G_2 \xrightarrow{\Lambda_2} \Gamma, z \vdash G_2'\right)$$

Then, given that $Seq(\Gamma \vdash G_1|G_2) = Seq(\Gamma \vdash G_1) \cup Seq(\Gamma \vdash G_2)$ *we have,*

$$Prod\left(r(d_1,d_2)//\Gamma \vdash G_1|G_2 \xrightarrow{\Lambda} \Gamma \vdash G_1'|G_2'\{y/z\}\right) \subseteq$$
$$\left\{L_P \xrightarrow{\Lambda'} G' \mid P \in Seq(\Gamma \vdash G_1)\right\} \cup \left\{L_P \xrightarrow{\Lambda'} G' \mid P \in Seq(\Gamma \vdash G_2)\right\} =$$
$$\left\{L_P \xrightarrow{\Lambda'} G' \mid P \in Seq(\Gamma \vdash G_1|G_2)\right\}$$

- *For* $(close_\pi)$*, it is similar to* (com_π)*.*
- *For* (par_π) *we know that,*

$$Prod\left(d_1//\Gamma \vdash G_1 \xrightarrow{\Lambda} \Gamma, \Delta \vdash G_1'\right) \overset{IH}{\subseteq} \left\{L_P \xrightarrow{\Lambda'} G' \mid P \in Seq(\Gamma \vdash G_1)\right\}$$

Then, given that $Seq(\Gamma \vdash G_1) \subseteq Seq(\Gamma \vdash G_1) \cup Seq(\Gamma \vdash G_2) = Seq(\Gamma \vdash G_1|G_2)$ *we have,*

$$Prod\left(r(d_1)//\Gamma \vdash G_1|G_2 \xrightarrow{\Lambda} \Gamma, \Delta \vdash G_1'|G_2\right) \overset{def}{\subseteq}$$
$$\left\{L_P \xrightarrow{\Lambda'} G' \mid P \in Seq(\Gamma \vdash G_1)\right\} \subseteq \left\{L_P \xrightarrow{\Lambda'} G' \mid P \in Seq(\Gamma \vdash G_1|G_2)\right\}$$

- *For* (res_π) *we know that,*

$$Prod\left(d_1//\Gamma, x \vdash G_1 \xrightarrow{\Lambda} \Gamma, x, \Delta \vdash G_2\right) \overset{IH}{\subseteq} \left\{L_P \xrightarrow{\Lambda''} G' \mid P \in Seq(\Gamma, x \vdash G_1)\right\}$$

$$Prod\left(r(d_1)//\Gamma \vdash \nu x.G_1 \xrightarrow{\Lambda'} \Gamma, \Delta' \vdash \nu x.G_2\right) \overset{def}{=}$$
$$Prod\left(d_1//\Gamma, x \vdash G_1 \xrightarrow{\Lambda} \Gamma, x, \Delta \vdash G_2\right)$$

Then, given that $Seq(\Gamma \vdash \nu x.G_1) = Seq(\Gamma, x \vdash G_1)$ *we have,*

$$Prod\left(r(d_1)//\Gamma \vdash \nu x.G_1 \xrightarrow{\Lambda'} \Gamma, \Delta' \vdash \nu x.G_2\right) \subseteq$$
$$\left\{L_P \xrightarrow{\Lambda''} G' \mid P \in Seq(\Gamma \vdash \nu x.G_1)\right\}$$

- *For* $(open_\pi)$ *we know that,*

$$Prod\left(d_1//\Gamma, y \vdash G_1 \xrightarrow{\Lambda} \Gamma, y \vdash G_2\right) \overset{IH}{\subseteq} \left\{L_P \xrightarrow{\Lambda'} G' \mid P \in Seq(\Gamma, y \vdash G_1)\right\}$$

$$Prod\left(r(d_1)//\Gamma \vdash \nu y.G_1 \xrightarrow{\Lambda} \Gamma, y \vdash G_2\right) \overset{def}{=} Prod\left(d_1//\Gamma, y \vdash G_1 \xrightarrow{\Lambda} \Gamma, y \vdash G_2\right)$$

Then, given that $Seq(\Gamma \vdash \nu y.G_1) = Seq(\Gamma, y \vdash G_1)$ *we have,*

$$Prod\left(r(d_1)//\Gamma \vdash \nu y.G_1 \xrightarrow{\Lambda} \Gamma, y \vdash G_2\right) \subseteq \left\{L_P \xrightarrow{\Lambda'} G' \mid P \in Seq(\Gamma \vdash \nu y.G_1)\right\}$$

2. *Any edge label* L_P *corresponds, by translation, to a sequential agent that is a guarded sum. This guarded sum has a finite number of terms, so finally, under the late semantics the set* $\{\Gamma \vdash L_P \xrightarrow{\Lambda} \Gamma' \vdash G'\}$ *is finite.*

3. *First we prove that* $S(P)$ *is finite. By structural induction:*

   Base Case:

   - $|S(nil)| = 0$
   - $|S(X)| = 0$
   - $S\ (\sum_{i=1}^{n} \alpha_i.P_i) = 1 + \sum_{i=1}^{n} |S(\alpha_i.P_i)|$ *which is finite by I.H.*
   - $|S(rec\,X.P)| = |S(P)[rec\,X.P/X]| = |S(P)|$ *which is finite by I.H.*
   - $|S(P|Q)| = |S(P)| + |S(Q)|$ *which are finite by I.H.*
   - $|S(\nu x.P)| = |S(P)|$ *which is finite by I.H.*

   Now we prove that $Seq(G)$ *is finite. By structural induction:*

   Base Case:

   - $|Seq(\Gamma \vdash L_P)| = |S(P)|$ *which is finite.*
   - $|Seq(\Gamma \vdash nil)| = 0$
   - $|Seq(\Gamma \vdash G_1|G_2)| = Seq(\Gamma \vdash G_1)| + Seq(\Gamma \vdash G_2)|$ *which are finite by I.H.*
   - $|Seq(\Gamma \vdash \nu x.G)| = |Seq(\Gamma, x \vdash G)|$ *which is finite by I.H.* □

COROLLARY 7.17. *For all agent term* P, $Prod(d//[\![P]\!]_{\Gamma_0} \Rightarrow \Gamma \vdash G)$ *is finite.*

Proof. *By the translation function* $[\![P]\!]_{\Gamma_0}$ *is a graph. Then the corollary is proved by Theorem 7.16.* □

As we already mentioned, we are mapping π-calculus that has a sequential operational semantics (one transition at a time) to a distributed concurrent context. So it is clear that there are transitions for judgements (the concurrent ones) that cannot be obtained in π-calculus. Then, the following theorem states that a transition step in π-calculus is done, if and only if, there is a judgement transition between the corresponding translations under the Milner$_\pi$ transition system. Proofs constructed with Milner$_\pi$ transition system are transitions corresponding to sequential steps of π-calculus. Also, we have to say that the theorem below is sufficient to prove the semantic correspondence given that Theorem 7.5 already proved the bijective correspondence of the translation from agents to judgements (i.e. graphs). In this way, we are sure that any graph resulting from a transition has a corresponding agent.

One thing to mention is about the translation of τ actions. The standard operational semantics of the π-calculus is defined via labelled transitions $P \xrightarrow{\alpha} P'$ with α an action, where $P \xrightarrow{\tau} P'$ indicates that agent P goes to P' by an internal action. This is done without the need of specifying on which port the internal action takes place and is due to the fact that as being sequential it is the only action occurring. In a distributed concurrent context we need to know where actions are taking place, as more than one action can happen at the same time.

Under the conditions of Milner$_\pi$ rules we have no concurrent actions but still we are in a distributed context, so for the case of τ actions we have two possible translations ($(x, \tau, <>)$ or $\emptyset$). This translation depends if it is the case of a synchronization in a visible node or if the node where the synchronization has taken place is being restricted using rule (res_π). Actually, this is the only case where $|\Lambda| \neq 1$ for a transition obtained from Milner$_\pi$ rules.

LEMMA 7.18. *For any agent P and substitution σ, $[\![P]\!]_\Gamma \sigma = [\![\sigma P]\!]_{\sigma\Gamma}$.*

Proof. *We proceed by structural induction:*

Base Case:

- *For $[\![nil]\!]_\Gamma$ it is obvious.*
- *For P sequential, we have that $P = \sigma_P \widehat{P}$ and $\gamma P = \sigma_{\gamma P} \widehat{\gamma P}$. Then, by translation:*

$$\begin{aligned} [\![P]\!]_\Gamma\ \gamma &= \gamma\Gamma \vdash \gamma\sigma_P L_{\widehat{P}}(v_1, \ldots, v_{|\mathrm{fn}(\widehat{P})|}) \\ [\![\gamma P]\!]_{\gamma\Gamma} &= \gamma\Gamma \vdash \sigma_{\gamma P} L_{\widehat{\gamma P}}(v_1, \ldots, v_{|\mathrm{fn}(\widehat{\gamma P})|}) \end{aligned}$$

And they are equal by properties of standard decomposition that say $\gamma\sigma_P = \sigma_{\gamma P}$ and $\widehat{P} = \widehat{\gamma P}$.

-

$$\begin{aligned} [\![P|Q]\!]_\Gamma\ \sigma &= \sigma\Gamma \vdash \sigma G_P | \sigma G_Q \\ [\![\sigma(P|Q)]\!]_{\sigma\Gamma} &= [\![\sigma P|\sigma Q]\!]_{\sigma\Gamma} = \sigma\Gamma \vdash G_{\sigma P} | G_{\sigma Q} \end{aligned}$$

Then, by Inductive Hypothesis we have $\sigma G_P = G_{\sigma P}$ and $\sigma G_Q = G_{\sigma Q}$.

$$\begin{aligned} [\![\nu x.P]\!]_\Gamma\ \sigma &= \sigma\Gamma \vdash \nu x.\sigma G_P \\ [\![\sigma(\nu x.P)]\!]_{\sigma\Gamma} &= \sigma\Gamma \vdash \nu x.G_{\sigma P} \end{aligned}$$

Then, by Inductive Hypothesis we have $\sigma G_P = G_{\sigma P}$.

- $$\begin{aligned} [\![rec\,X.P]\!]_\Gamma\ \sigma &= [\![P[rec\,X.P/X]]\!]_\Gamma\ \sigma \\ [\![\sigma rec\,X.P]\!]_{\sigma\Gamma} &= [\![rec\,X.\sigma P]\!]_{\sigma\Gamma} = [\![\sigma P[rec\,X.\sigma P/X]]\!]_{\sigma\Gamma} \end{aligned}$$

  Then, by Inductive Hypothesis we have,

  $$[\![P[rec\,X.P/X]]\!]_\Gamma\ \sigma = [\![\sigma P[rec\,X.\sigma P/X]]\!]_{\sigma\Gamma}$$

□

LEMMA 7.19. *For all* σ *and* P *sequential such that* $P = \gamma P'$,

1. $P \xrightarrow{\alpha} Q$ *implies* $P' \xrightarrow{\alpha'} Q'$ *with* $\alpha = \gamma\alpha'$, $Q = \gamma Q'$.
2. $P' \xrightarrow{\alpha}{}' Q'$ *implies* $P \xrightarrow{\alpha} Q$ *with* $\alpha = \gamma\alpha'$, $Q = \gamma Q'$.

Proof. *First, given that* P *is sequential we have that* $\alpha = x(y), \bar{x}y$. *Then, by standard decomposition and with* $P = \gamma P'$ *we obtain that:*

$(P = \sigma_P\widehat{P}) \wedge (\widehat{P'} = \widehat{\gamma P'}) \wedge (\sigma_{\gamma P'} = \gamma\sigma_{P'}) \wedge (P = \gamma P')$ *implies* $(\widehat{P'} = \widehat{P} = \widehat{\gamma P'}) \wedge (\sigma_P = \sigma_{\gamma P'})$.

1. $$\begin{aligned} P \xrightarrow{\alpha} Q \quad \textit{implies} \quad & \sigma_P\widehat{P} \xrightarrow{\sigma_P\alpha"} \sigma_P Q" \textit{ with } (\sigma_P Q" = Q = \gamma\sigma_{P'}Q" = \gamma Q') \\ & \wedge\ (\sigma_P\alpha" = \alpha = \gamma\sigma_{P'}\alpha" = \gamma\alpha') \\ \textit{implies} \quad & \widehat{P} \xrightarrow{\alpha"} Q" \\ \textit{implies} \quad & stP' \xrightarrow{\alpha"} Q" \\ \textit{implies} \quad & \sigma_{P'}\widehat{P'} \xrightarrow{\sigma_{P'}\alpha"} \sigma_{P'}Q" \\ \textit{implies} \quad & P' \xrightarrow{\alpha'} Q' \textit{ with } (\alpha' = \sigma_{P'}\alpha") \wedge (Q' = \sigma_{P'}Q") \end{aligned}$$

2. $$\begin{aligned} P' \xrightarrow{\alpha'} Q' \quad \textit{implies} \quad & \sigma_{P'}\widehat{P'} \xrightarrow{\alpha'} Q' \textit{ implies } \sigma_{P'}\widehat{P} \xrightarrow{\alpha'} Q' \textit{ implies } \gamma\sigma_{P'}\widehat{P} \xrightarrow{\gamma\alpha'} \gamma Q' \\ \textit{implies} \quad & \sigma_{\gamma P'}\widehat{P} \xrightarrow{\gamma\alpha'} \gamma Q' \textit{ implies } \sigma_P\widehat{P} \xrightarrow{\gamma\alpha'} \gamma Q' \textit{ implies } P \xrightarrow{\alpha} Q \end{aligned}$$

□

THEOREM 7.20. [Semantic Correspondence] *For any π-calculus agents P and Q, $P \xrightarrow{\alpha} Q$, if and only if, there is a transition $[\![P]\!]_\Gamma \xrightarrow{\{[\alpha]_{\Gamma'}\}} [\![Q]\!]_{\Gamma'}$, with $\mathrm{fn}(P) \subseteq \Gamma$, $\mathrm{fn}(Q)$, $\mathrm{fn}(\alpha) \subseteq \Gamma'$ and $\Gamma \subseteq \Gamma'$.*

Proof. *For the left-right direction, we prove the theorem by rule induction on the π-calculus operational semantics.*

Base Case:

- *For* (sum), *we have by standard decomposition that* $P = \sum_{i=1}^{n} \alpha_i.P_i = \sigma_P \widehat{\sum_{i=1}^{n} \alpha_i.P_i}$ *and by Lemma 7.19 that* $\widehat{\sum_{i=1}^{n} \alpha_i.P_i} \xrightarrow{\sigma_P^{-1}(\alpha_j)} \sigma_P^{-1}(P_j)$. *Then, by translation of productions we obtain,*

$$\widehat{\mathrm{fn}(\sum_{i=1}^{n} \alpha_i.P_i)} \vdash L_{\widehat{P}}(v_1, \ldots, v_{|\mathrm{fn}(\widehat{P})|}) \xrightarrow{\{[\sigma_P^{-1}(\alpha_j)]_{\Gamma'}\}} [\![\sigma_P^{-1}(P_j)]\!]_{\Gamma'} \quad \mathrm{fn}(P) \subseteq \Gamma'$$

Now applying rule (ren_π) *with* $\xi = \sigma_P$ *and any* Γ, *and by Lemma 7.18 we obtain,*

$$\Gamma, \sigma_P \widehat{\mathrm{fn}(\sum_{i=1}^{n} \alpha_i.P_i)} \vdash \sigma_P L_{\widehat{P}}(v_1, \ldots, v_{|\mathrm{fn}(\widehat{P})|}) \xrightarrow{\{[\sigma_P^{-1}(\alpha_j)]_{\Gamma, \Gamma'\ \sigma_P}\}} [\![\sigma_P^{-1}(P_j)]\!]_{\Gamma,\Gamma'\ \sigma_P} =$$

$$\Gamma, \mathrm{fn}(\sum_{i=1}^{n} \alpha_i.P_i) \vdash \sigma_P L_{\widehat{P}}(v_1, \ldots, v_{|\mathrm{fn}(\widehat{P})|}) \xrightarrow{\{[\alpha_j]_{\Gamma, \sigma_P \Gamma'}\}} [\![P_j]\!]_{\Gamma, \sigma_P \Gamma'}$$

Inductive Hypothesis: *For any transition of a subterm P' of P of the form $P' \xrightarrow{\alpha} Q$ there is a basic transition $\Gamma \vdash G_{P'} \xrightarrow{\{[\alpha]_{\Gamma'}\}} \Gamma' \vdash G_Q$ (up to alpha conversion of bounded nodes)*

- *For* (par), *we know that* $[\![P|Q]\!]_\Gamma = \Gamma \vdash G_P|G_Q$ *and by Inductive Hypothesis there is a transition* $\Gamma \vdash G_P \xrightarrow{\{[\alpha]_{\Gamma'}\}} \Gamma' \vdash G_{P'}$. *Then using rule* (par_π) *with* $\Gamma \vdash G_Q$ *we obtain* $\Gamma \vdash G_P|G_Q \xrightarrow{\{[\alpha]_{\Gamma'}\}} \Gamma' \vdash G_{P'|G_Q}$ *with* $\Gamma' \vdash G_{P'}|G_Q = [\![P'|Q]\!]_{\Gamma'}$.

- *For* (com), *we have by Inductive Hypothesis (with* $\mathrm{fn}(P)$, $\mathrm{fn}(Q) \subseteq \Gamma$*),*

$$P \xrightarrow{\bar{x}y} P' \quad \textit{implies} \quad \Gamma \vdash G_P \xrightarrow{\{(x,out,<y>)\}} \Gamma \vdash G_{P'}$$
$$Q \xrightarrow{x(z)} Q' \quad \textit{implies} \quad \Gamma \vdash G_Q \xrightarrow{\{(x,in,<z>)\}} \Gamma, z \vdash G_{Q'}$$

Then, using rule (com_π) *we obtain* $\Gamma \vdash G_P|G_Q \xrightarrow{\{(x,\tau,<>)\}} \Gamma \vdash G_{P'}|G_{Q'}\{y/z\}$ *and by Lemma 7.18 we have that* $[\![P'|Q'\{y/z\}]\!]_\Gamma = [\![P'|Q']\!]_{\Gamma,z}\ \{y/z\}$.

- *For* (close), *we have by Inductive Hypothesis (with* $\mathrm{fn}(P)$, $\mathrm{fn}(Q) \subseteq \Gamma$ *and* $y \notin \Gamma$*),*

$$P \xrightarrow{\bar{x}(y)} P' \quad implies \quad \Gamma \vdash G_P \xrightarrow{\{(x,out,<y>)\}} \Gamma, y \vdash G_{P'}$$
$$Q \xrightarrow{x(y)} Q' \quad implies \quad \Gamma \vdash G_Q \xrightarrow{\{(x,in,<y>)\}} \Gamma, y \vdash G_{Q'}$$

Then, using rule $(close_\pi)$ *we obtain* $\Gamma \vdash G_P | G_Q \xrightarrow{\{(x,\tau,<>)\}} \Gamma \vdash \nu y.(G_{P'} | G_{Q'})$.

- *For* (open), *we have by Inductive Hypothesis (with* $\mathrm{fn}(P) \subseteq \Gamma \cup \{y\}$*),*

$$P \xrightarrow{\bar{x}y} P' \quad implies \quad \Gamma, y \vdash G_P \xrightarrow{\{(x,out,<y>)\}} \Gamma, y \vdash G_{P'}$$

Then, using rule $(open_\pi)$ *we obtain* $\Gamma \vdash \nu y.G_P \xrightarrow{\{(x,out,<y>)\}} \Gamma, y \vdash G_{P'}$.

- *For* (res), *we have to check two cases:*

  1. *For* $\Lambda = \{(v, out, < w >)\} | \{(v, in, < w >)\} | \{(y, \tau, <>)\}$ *with* $v, w, y \neq x$*, we have by Inductive Hypothesis that,* $P \xrightarrow{\alpha} P'$ *implies* $\Gamma, x \vdash G_P \xrightarrow{\Lambda} \Gamma, x, \Delta \vdash G_{P'}$ *with* $x \notin n(\Lambda)$ *and* $\Lambda(x) \uparrow$ *for* $\Lambda = \{[\![\alpha]\!]_{\Gamma,x,\Delta}\}$

$$P \xrightarrow{\bar{x}y} P' \quad implies \quad \Gamma, y \vdash G_P \xrightarrow{\{(x,out,<y>)\}} \Gamma, y \vdash G_{P'}$$

  Then, using the first rule (res_π) *we obtain* $\Gamma \vdash \nu x.G_P \xrightarrow{\Lambda} \Gamma, \Delta \vdash \nu x.G_{P'}$.

  2. *In the second case we have* $\alpha = \tau$ *and we assume that it happens on the node that is being restricted. So, by inductive Hypothesis we have,*

$$P \xrightarrow{\tau} P' \quad implies \quad \Gamma, x \vdash G_P \xrightarrow{\{(x,tau,<>)\}} \Gamma, x \vdash G_{P'}$$

  Then, using the second rule (res_π) *we obtain* $\Gamma \vdash \nu x.G_P \xrightarrow{\emptyset} \Gamma \vdash \nu x.G_{P'}$.

Conversely, for the second part of the theorem, we want to prove that, given P*, a* π*-calculus agent,* $[\![P]\!]_\Gamma \xrightarrow{\Lambda} \Gamma' \vdash G$ *implies* $P \xrightarrow{\alpha} Q$, $\Lambda = [\![\alpha]\!]_{\Gamma'}, \Gamma' \vdash G = [\![Q]\!]_{\Gamma'}$*, with* $\mathrm{fn}(P) \subseteq \Gamma \subseteq \Gamma'$ *and* $\mathrm{fn}(\alpha), \mathrm{fn}(Q) \subseteq \Gamma'$*. Theorem 7.5 assures that the inverse function for the translation exists.*

We proceed by rule induction on Milner$_\pi$ *Transition System.*

Base Case:

- *For productions is obvious by definition.*

Inductive Hypothesis: *For any subterm* P' *of* P *with* $[\![P']\!]_\Gamma \xrightarrow{\Lambda} \Gamma' \vdash G$ *there is a transition* $P' \xrightarrow{\alpha} Q$*, with* $\Lambda = [\![\alpha]\!]_{\Gamma'}, \Gamma' \vdash G = [\![Q]\!]_{\Gamma'}$, $\mathrm{fn}(P) \subseteq \Gamma$, $\mathrm{fn}(P'), \subseteq \Gamma'$.

- *For* (rcn_π), *we have that* $\Gamma', \xi\Gamma \vdash \xi G = [\![Q]\!]_{\Gamma', \xi\Gamma}$ *and by Inductive Hypothesis,* $\Gamma \vdash G \xrightarrow{\Lambda} \Gamma, \Delta \vdash G' \in \mathcal{P}$ *implies* $P \xrightarrow{\alpha} P'$, P *sequential standard and* $\Gamma \vdash G = [\![P]\!]_\Gamma$, $\Gamma, \Delta \vdash G' = [\![P']\!]_{\Gamma,\Delta}$.

  Then, by Lemma 7.18, $\xi\Gamma \vdash \xi G = [\![Q]\!]_{\xi\Gamma} = [\![P]\!]_\Gamma\ \xi = [\![\xi P]\!]_{\xi\Gamma}$, *and by Theorem 7.5,* $Q = \xi P$.

  Finally, by Lemmas 7.18 and 7.19, and Theorem 7.5 we obtain $\xi P \xrightarrow{\xi\alpha} \xi P'$ *with,*

$$[\![\xi P]\!]_{\xi\Gamma} \xrightarrow{[\xi\alpha]_{\xi\Gamma,\xi\Delta}} [\![\xi P']\!]_{\xi\Gamma,\xi\Delta} =$$
$$[\![P]\!]_\Gamma\ \xi \xrightarrow{[\alpha]_{\Gamma,\Delta}\ \xi} [\![P']\!]_{\Gamma,\Delta}\ \xi = \xi\Gamma \vdash \xi G \xrightarrow{\xi\Lambda} \xi\Gamma, \xi\Delta \vdash \xi G'$$

- *For* (par_π), *we have by Inductive Hypothesis,* $\Gamma \vdash G_1 \xrightarrow{\Lambda} \Gamma, \Delta \vdash G_1'$ *implies* $P \xrightarrow{\alpha} P'$. *Then, by Theorem 7.5 we have* $Q = \{[\![\Gamma \vdash G_2]\!]\}$. *Using rule* (par) *we obtain* $P|Q \xrightarrow{\alpha} P'|Q$ *which by translation gives* $\Gamma \vdash G_1|G_2 \xrightarrow{\Lambda} \Gamma, \Delta \vdash G_1'|G_2$.

- *For* (com_π), *we have by Inductive Hypothesis (with* $\mathrm{fn}(P), \mathrm{fn}(Q) \subseteq \Gamma$*),*

$$\Gamma \vdash G_1 \xrightarrow{\{(x,out,<y>)\}} \Gamma \vdash G_1' \quad \textit{implies} \quad P \xrightarrow{\bar{x}y} P'$$
$$\Gamma \vdash G_2 \xrightarrow{\{(x,in,<z>)\}} \Gamma, z \vdash G_2' \quad \textit{implies} \quad Q \xrightarrow{x(z)} Q'$$

  Then, using rule (com), $P|Q \xrightarrow{\tau} P'|Q'\{y/z\}$. *And by translation,* $[\![P|Q]\!]_\Gamma = \Gamma \vdash G_1|G_2$ *and* $[\![\tau]\!]_\Gamma = (x, \tau, <>)$, *we conclude by Lemma 7.18 that* $[\![P'|Q'\{y/z\}]\!]_\Gamma = \Gamma \vdash G_1'|G_2'\{y/z\}$.

- *For* $(close_\pi)$, *we have by Inductive Hypothesis (with* $\mathrm{fn}(P), \mathrm{fn}(Q) \subseteq \Gamma$*),*

$$\Gamma \vdash G_1 \xrightarrow{\{(x,out,<y>)\}} \Gamma, y \vdash G_1' \quad \textit{implies} \quad P \xrightarrow{\bar{x}(y)} P'$$
$$\Gamma \vdash G_2 \xrightarrow{\{(x,in,<y>)\}} \Gamma, y \vdash G_2' \quad \textit{implies} \quad Q \xrightarrow{x(y)} Q'$$

  Then, using rule (close), $P|Q \xrightarrow{\tau} \nu x.\, P'|Q'$. *And by translation,* $[\![P|Q]\!]_\Gamma = \Gamma \vdash G_1|G_2$ *and* $[\![\nu x.\, P'|Q']\!]_\Gamma = \Gamma \vdash \nu x.(G_1'|G_2')$ *with* $[\![\tau]\!]_\Gamma = (x, \tau, <>)$.

- *For* $(open_\pi)$, *we have by Inductive Hypothesis (with* $y \neq x$*),*

$$\Gamma, y \vdash G \xrightarrow{\{(x,out,<y>)\}} \Gamma, y \vdash G' \quad \textit{implies} \quad P \xrightarrow{\bar{x}y} P'$$

  Then, using rule (open) *we obtain* $\nu y.\, P \xrightarrow{\bar{x}(y)} P'$. *And by translation,* $[\![\nu y.\, P]\!]_\Gamma = \Gamma \vdash \nu y.G$.

- *For (res_π), we have two cases:*

  1. *For $x \notin n(\Lambda) and \Lambda(x)\uparrow$.*

     By Inductive Hypothesis, we have $\Gamma, x \vdash G \xrightarrow{\Lambda} \Gamma, x, \Delta \vdash G'$ implies $P \xrightarrow{\alpha} P'$ with $x \in \mathrm{fn}(P)$ and $x \notin \mathrm{n}(\alpha)$. Then, using rule (res) *we obtain, $\nu x.\, P \xrightarrow{\alpha} \nu x.\, P'$ where $[\![\nu x.\, P]\!]_\Gamma = \Gamma \vdash \nu x.G$ and $[\![\nu x.\, P']\!]_{\Gamma,\Delta} = \Gamma, \Delta \vdash \nu x.G'$ by translation.*

  2. *This is the same case as bellow, but instead of choosing $[\![\tau]\!]_\Gamma = \{(x, \tau, <>)\}$, we choose for the new transition $[\![\tau]\!]_\Gamma = \emptyset$ given that the synchronization is on the node to be restricted.* □

7.2 π-calculus vs. πI-calculus

In this section we briefly present a study of a restriction of our formalism to share only new nodes, in the style of πI-calculus [Sangiorgi, D., 1996].

With the goal of studying the expressive power of π-calculus, in [Sangiorgi, D., 1996], the mobility mechanisms of the π-calculus were separated in two, respectively called *internal* and *external mobility*. The study of the πI-calculus, which corresponds to the calculus that uses only internal mobility (i.e. only new nodes can be shared), showed that internal mobility is responsible for much of the expressive power of π-calculus.

In the case of our formalism to share only new nodes we have, for a transition $\Gamma \vdash G \xrightarrow{\Lambda} \Gamma, \Delta \vdash G'$, to impose on Λ the condition that names in $n(\Lambda)$ should not be in Γ ($n(\Lambda) \cap \Gamma = \emptyset$). Then, Δ does not depend on Γ and can be written as:

$$\Delta = \bigcup_{x a \vec{y} \in \Lambda} set(\vec{y})$$

The application of this restriction has almost no effect in any of the presented transition systems. Only the corresponding open operation (rule **(open)** for Hoare and the specific case of rule **(res)** for Milner) is not necessary because no old name can by shared. With respect to the translation in Section 7.1, with πI-calculus we have only the actions:

$$\tau \mid\mid x(z) \mid \bar{x}(z)$$

Then, the transition system in Table 2.2 does not need the *(Com)* and *(Open)* rules, and the *(Sum)* rule is replaced for the bounded rule:

$$(Sum) \sum_{i=1}^{n} \alpha_i.P_i \xrightarrow{\alpha_i} P_i \text{ with } \alpha_i = x(y),\ \bar{x}(y)$$

Also, the Milner$_\pi$ transition system in Table 7.1 does not need the (com_π) and $(open_\pi)$ rules, and rule number **6** of the translation in table 7.2 has to be changed by:

$$\frac{P \overset{\tilde{v}(w)}{\longmapsto} Q \quad P \text{ sequential standard agent with } w \text{ a fixed name, } w \notin fn(P)}{[\![P]\!]_{fn(P)} \overset{\{(v,out,<w>)\}}{\longrightarrow} [\![Q]\!]_{fn(P)\cup\{w\}}}$$

All the correspondence results are still valid.

Part VI
Adding Design Transformations to Styles

One major problem in the specification and verification of software architectures and especially with distributed systems, is when system evolution includes reconfiguration. Some of the existing approaches for modeling software architecture styles deal with these problems but only partially. Others only allow for fixed description of styles, components and connectors.

One important issue is how to assure consistency of the specified reconfiguration with respect to a style. In this part of the thesis, we continue with the idea of supporting graph and graph transformation as a general framework for style specifications. Then, based on the use of graph grammars as language for style description, we complement the approach with a method for specifying more complex reconfigurations (called *transformations*) over the topology of an architecture style. The method warranties that if the transformation can be specified, then its application over system instances will be consistent with respect to the expected architecture style configuration. For this method, we do not use synchronized rewriting, we only use graph grammars as style description language and *transformations* for defining the evolution of the topology.

The main difference of this method with respect to the one presented in Part IV using name mobility, is that the method in Part IV is more dynamic in the sense that it applies to running open-ended systems without global control except for synchronization, whereas the approach with transformations may be useful for working at the level of blueprints, i.e. it rearranges the design steps of the system to produce a different but consistent system. Thus, the latter method can be applied to specify very general kinds of reconfigurations and mobility (as it is shown in the thesis examples), but it requires a global knowledge of system structure. Some possible steps for combining both methods are described in the future work in Chapter 10.

Our motivation can be summarized agreeing with and quoting Le Métayer [Le Métayer, D., 1998]:

> *We believe that a better basis for mastering large software systems is to ensure that they respect the desired topology and properties by construction rather than trying to try to prove it a posteriori.*

Including dynamic changes as part of the style let us be sure that the specified changes are the only ones allowed for all style instances. Therefore, consistency checking is valid also for all style instances.

As related work on specifying consistent reconfigurations we can mention again [Le Métayer, D., 1998]. In his paper Le Métayer, together with the context free grammar for the static structure of the style and the coordinator conditional rules for reconfiguration, proposes a semi-decidable algorithm for "type checking" of styles to ensure that the coordinator does not break the style structure (which defines the type). The algorithm corresponds to a proof of convergence of graph rewriting rules. Also, part of the checking of the coordinator rules is due to the set representation that was chosen in [Le Métayer, D., 1998]. For example, when a component is removed it means that the coordinator rule has to take into account the deletion of the node and also all the edges corresponding to its communication links (this is not needed in our representation).

In Chapter 8 we show how to specify reconfigurations by *transformation rules* over grammar derivations. For this we introduce a new (and more useful) notion of grammar derivation and then formalize the approach by introducing an innovative representation of HR systems as higher-order terms of a typed λ-calculus. In Chapter 9 we give a first idea of how this approach can be used by a designer that may want to specify a reconfiguration in a more constructive (and maybe more intuitive) way with intermediate steps that may not correspond to valid configurations of the style.

Chapter 8

Consistent Transformations over Derivations

As it was done in Part III, in Section 8.1 we first give an informal introduction to the method, with an example from the *TRMCS* case study of Section 3.1. In Section 8.2 we present the formalization of the method based on λ-Calculus.

8.1 Reconfiguration by Transformations

The method specifies reconfigurations by *transformation rules* over grammar derivations. Again, we assume HR grammars as formalism for describing styles. Then, for each graph of a style there is a set of derivations of the grammar that are the possible ways of constructing it from the application of grammar productions. A transformation is applied over a derivation segment and returns a new derivation segment. The idea is that to apply a transformation over a graph (i.e. architecture) you have to find a derivation segment of that graph (from the style) that matches the transformation. After applying the transformation, its result corresponds to the reconfigured segment of a graph derivation. This result is part of a valid derivation of the graph obtained by the desired reconfiguration. This assures that a transformation is between valid derivations. It is important to mention that because transformations are over a derivation segment they can be composed and applied to several parts of a graph. Also, after a transformation is obtained, you can start from any of the derivations of the new graph allowing for other transformations to be applied.

It is clear that our aim is to give architects a tool to specify in a consistent way complex reconfigurations over the architectures they are working with. This is fundamental for software architecture modeling because once a transformation is obtained and its correctness checked (that it starts and ends with valid derivations), it can be included in a library of transformations for its future use. This work was introduced in [Hirsch, D. and Montanari, U., 2000] and [Hirsch, D. and Montanari, U., 1999]. In this part of the thesis we do not use SHR productions.

Given a HR Grammar grammar and the set of edge labels for the graphs it generates, we distinguish two sets of symbols, *nonterminal* (**NT**) and terminal labels (**T**). Nonterminals are labels that appear in the left-hand side of productions and terminals are labels that appear only in the right-hand side of productions.

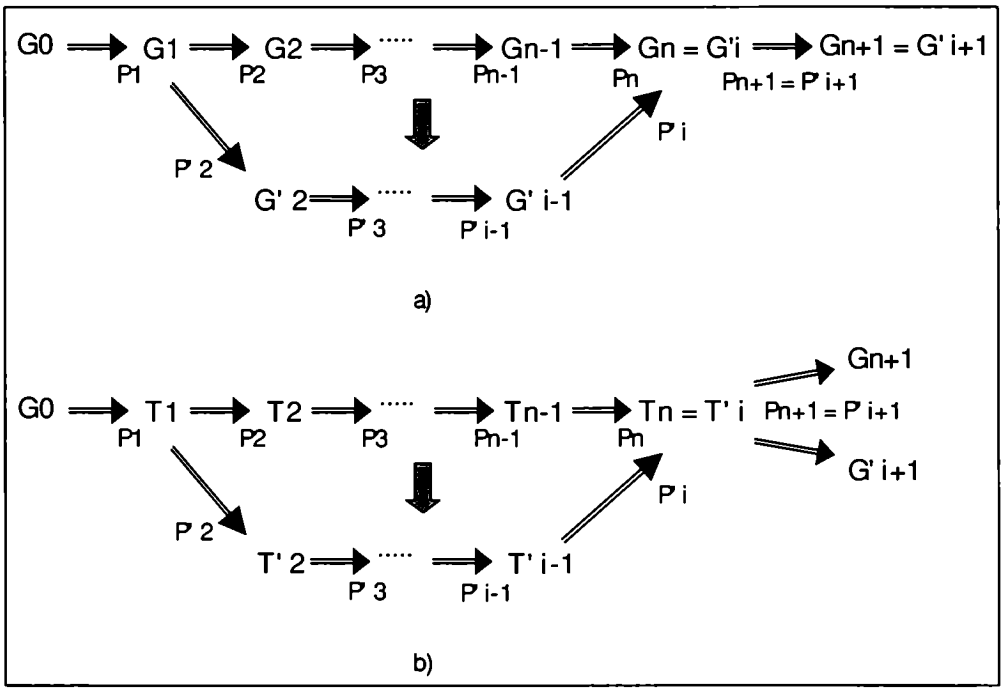

Figure 8.1: Definitions of Derivation.

A transformation is a rewrite rule of the form $L \Rightarrow R$ that transforms a derivation segment L (*input segment*) into a derivation segment R (*output segment*). It is easy to see that the ordinary notion of graph derivation (see Figure 8.1a.), i.e. a sequence of graphs and productions, could not serve this purpose. In fact, a derivation segment is limited by (we could also say: is typed with) two graphs, and could be reasonably replaced only with a segment typed with the same two graphs, which thus would eventually produce the same result. To be able to specify transformations, we introduce a new notion of derivation that equips derivation segments with a less stringent notion of type. For this we slightly change the definition of graph and production. Given a graph G_i that contains a set **NTG** of nonterminal symbols we identify its nonterminal edges as an ordered list (actually, the order is only necessary among edges with the same labels). Then, we identify the type of G_i with that list of nonterminal edges (note that graph types are not uniquely defined). In the same way the nonterminal edges in the right-hand side of productions are ordered. In this way, in a functional setting, we can see now that each derivation step ($Gi \Rightarrow_{pi+1} G_{i+1}$) is typed by the graph types ($T_i \Rightarrow_{pi+1} T_{i+1}$), and is obtained as a parallel composition of the application of production p_{i+1} over

a nonterminal symbol L_{i+1} that appears in G_i with a set of identities for the rest of nonterminal edges in G_i (see Figure 8.2). Finally, a derivation can be described as a sequential compositions (starting from the initial graph) of productions and auxiliary rules which permutes the nonterminals (ρ in Figure 8.2) composed in parallel with identities.

Figure 8.1b. shows (not including the identities for a simpler picture), the new definition for derivations and how a transformation can be applied. Now, we can transform a derivation segment (from p_2 to p_n, typed by T_1an d T_n) by another segment (p'_2 to p'_i) with the same type. This means that p'_2 has to expect a label that is compatible with T_1(not necessary the same one of p_2) and that the application of p'_i must have as result a graph with type $T'_i = T_n$ (with G'_i probably different from G_n). The derivation can be divided in three parts, the segment before the transformation input segment, the input segment itself and the rest of the derivation. As the interfaces (the types) of the input and output segments of the transformation are the same, the rest of the derivation is not changed. But as the result of the application of the transformation we obtain a new derivation (by composing again the three segments) ending with a different graph ($G_{n+1} \neq G'_{i+1}$).

8.1.1 Example

Using the *TRMCS* we exemplified the method. We specify a transformation that describes the reconfiguration of a set of **Users** when their **Router** fails. The failure of a **Router** is specified as a production that changes from a **Router** in an **Idle** state to a **Router** in a **Fail** state. What we want is that if a **Router** fails then all its **Users** have to be moved to another **Router**. Both segments must respect the type of the subgraph they have to be applied to, and they should talk about the involved **Routers**, i.e. the one that fails and the one receiving the **Users**, and also it has to be applicable for any number of **Users**. This is done by a simple transformation (Figure 8.2) over the derivation segment that contains the *Fail Production*, which corresponds to the input transformation segment. Note that the expected type for the input segment is two **Router** symbols in **Idle** state, where the *Fail Production* is applied over the first **Router** (noted with double line) together with an identity for the second **Router** label. The ending type of the segment is one **Fail Router** and one **Idle Router** labels.

In this case the output segment of the transformation is the same production but with an additional permutation (ρ) over the two **Routers** that are involved. Figure 8.2 includes a functional notation style specification of the transformation. Symbols $\bullet$ and $\otimes$ correspond to the functional and parallel compositions, respectively.

Also, the designer has to choose the correct derivation. In this case it is easy, the *Fail Production* has to be applied to the first **Router** before the second one generates its **Users**. In this way after the transformation all the **Users** (the ones from the failing **Router** and the rest from the receiving **Router**) are together. Figure 8.3 shows the application of the transformation over a derivation. As you can see the transformation is applied over the corresponding input segment, while the other parts of the initial derivation are the same. Then, after the transformation, a new valid derivation is

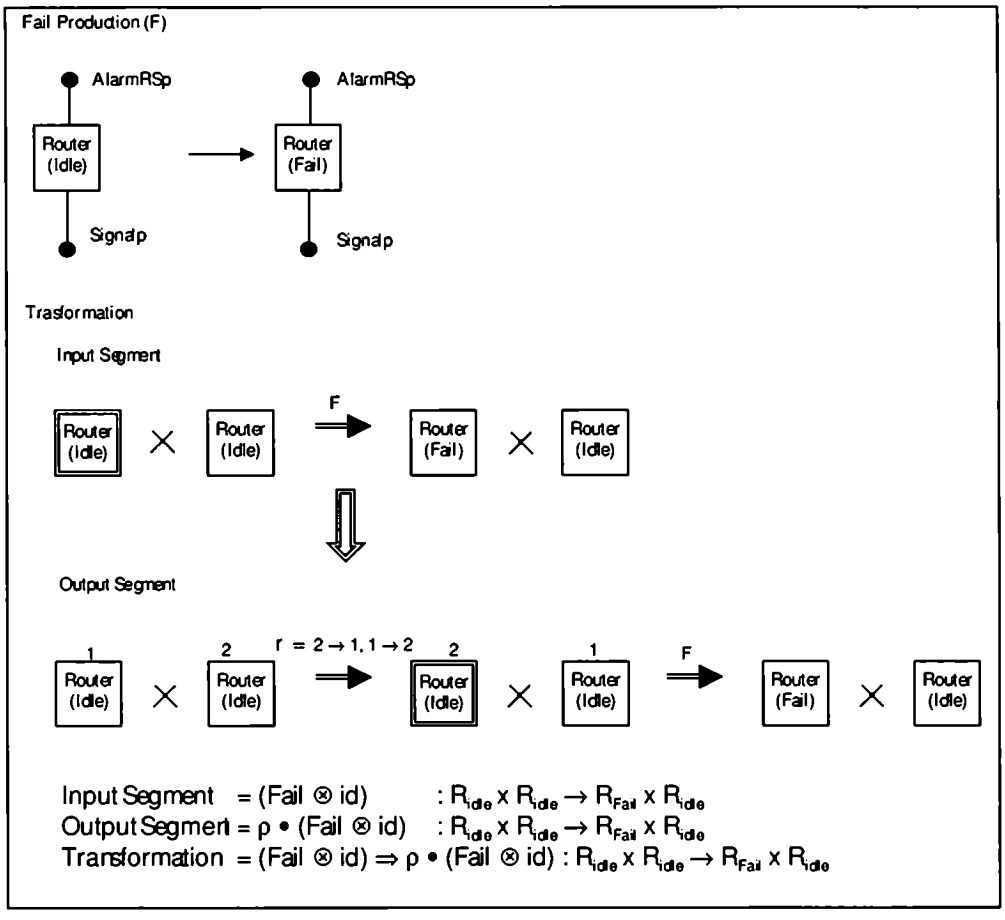

Figure 8.2: Transformation Rule.

obtained that respects the types but finishes with a different graph where all **Users** of the **Fail Router** are connected to the working **Router**.

The use of this method is intended in two forms. The first one is at the time when a designer wants to specify a reconfiguration for systems of a specific style. The second one is when a designer wants to apply one or more transformations, that are already specified and included in a library, to the style of the system she is working with. In the first case, to specify a transformation, the designer has to find an input segment for the subgraph that she wants the reconfiguration to happen, and a derivation for the resulting subgraph after the reconfiguration takes place. With these, the desired transformation is formally specified and can be added to the style description, being sure that it is consistent with respect to the style. For the second case, if a designer wants to apply an already specified transformation to follow the dynamic evolution of a specific system instance, it means that he has its graph specification. So, he can apply the transformation to a derivation that matches the input segment of the transformation and then obtain the new derivation for the reconfigured system.

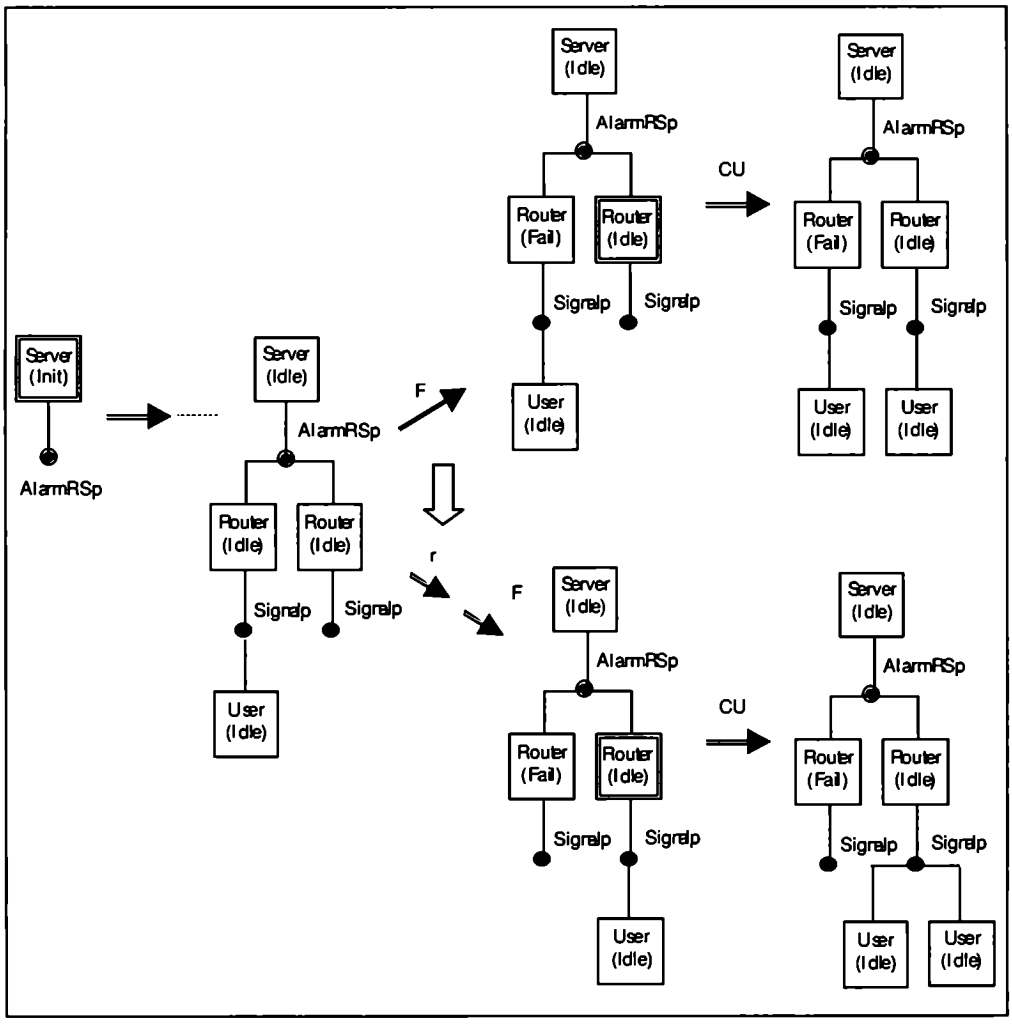

Figure 8.3: Application of the Transformation Rule.

With this method, transformations can be composed in parallel and applied simultaneously to different parts of a system being sure that the composition respects the style, given that it has to respect the type of the corresponding transformations.

8.2 Higher-Order Replacement Systems

For the rest of this chapter we present the formalization of the ideas introduced in Section 8.1.

In [Hirsch, D. and Montanari, U., 1999] we introduce a first approach to model reconfigurations with HR grammars as language for software architecture style description, with a representation of graphs as terms of a simply typed λ-calculus and using tiles [Gadducci, F. and Montanari, U., 1999] to specify consistent transformations over a

style. Then, in [Hirsch, D. and Montanari, U., 2000] we presented a deeper analysis of the relationship between HR systems and higher-order terms of a λ-calculus. The work presented in this section follows this line of work. For this purpose, we define a structure of type classes that is used to achieve a formalization (and differentiation) of the entities that appear in hyperedge replacement systems (graphs, productions, grammars, derivations and derivation trees). Also correspondence theorems are presented. The formalization with higher-order terms gives a clear representation of structures, shows that the specification method can have an efficient implementation and also allows the definition of a more expressive concept of derivation which is necessary to obtain a useful notion of consistent transformation already introduced in Section 8.1.

For the rest of this chapter we will use an example of a software architecture style explaining the different topics we will introduce. The example is an architecture style that describes a class of distributed systems that are arranged as trees (for example WANs) where their nodes are rings of components (for example LANs). The different rings are interconnected using bridge components.

8.2.1 Higher-Order Replacement and λ-Calculus

In this section we give a representation of HR systems as higher order terms of a simply typed λ-calculus. We consider typed λ-terms in $\lambda^{\rightarrow,\times,\epsilon}$ [Mitchell, J., 1996].

First of all, we need to add some details to the definitions of replacement systems of Chapter 2.

As we already mentioned, we define a set NT of nonterminal edge labels (i.e. labels on the left part of productions) and T a set of terminal edge labels (i.e. labels that only appear on the right part of productions) with $T \cap NT = \emptyset$. We define a *nonterminal graph* as a graph with an *ordered* list of edges labelled by nonterminals. A partial derivation is defined in the same way as a derivation except that the final graph is a nonterminal graph.

A derivation is characterized by its *derivation tree*. A *(partial) derivation tree* is a term whose constructors are the productions of the grammar and whose free variables are the nonterminals in the leafs.

Figure 8.4*a* shows the initial graph and grammar productions of the ring-tree style previously introduced. In this example, the tree of rings is composed of three types of components (i.e. edges): **C** is a generic component of a ring, **P** is a port component that indicates a point from where a new ring can be created and **b** is an external connection inserted in a ring. Two such **b** components form a bridge. **C** and **P** are nonterminal symbols, and **b** and **c** are terminals where **c** is a terminal symbol for components of type **C**. The initial graph of a ring tree is a ring with only one component. Productions *Brother* and *Port* create a new component or port, respectively, and production *Bridge* replaces a port **P** with a bridge that connects an existing ring with a new one. Production *Component* is the terminal rule for components **C**. Numbered nodes indicate the order of external nodes of productions and numbers on nonterminal edges indicate their order in nonterminal graphs.

Figure 8.4*b* is an example of a partial derivation over the grammar. The boxes in bold indicate the nonterminal over which the next production is applied. For example,

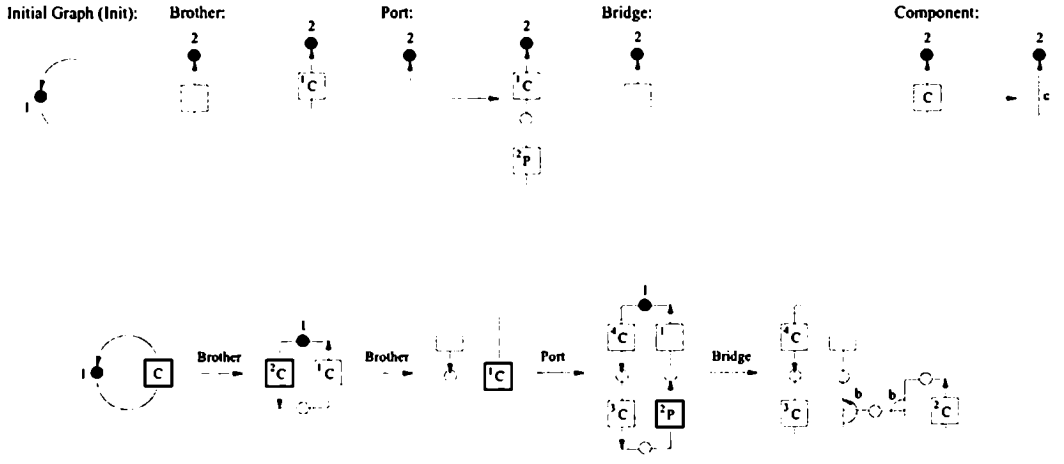

Figure 8.4: The grammar and a derivation of the ring-tree style.

the partial derivation tree of Figure 8.4*b* is,

$$Init < Brother < Port < C1, Bridge < C2 >>, Brother < C3, C4 >>> \qquad (8.1)$$

As we already mentioned, for the formalization we use typed λ-terms (or simply terms) in $\lambda^{\rightarrow,\times,\epsilon}$,i.e., terms with *exponential types*, *product types* and *unit*. As we will show below, a graph is described as a term of the calculus whose type is given by the number of external nodes and the nonterminal symbols of the graph it represents. Note that, as nonterminals define the interfaces for the application of productions on derivations, here the type of their corresponding terms will have the same role assuring the correct construction of derivations. Also, productions and derivations are represented as terms. For this we have to define the λ-signature $\Sigma = \langle \mathcal{B}, \mathcal{C} \rangle$ of the language. For unit type ϵ there is only one value (the empty list) which we write as $<>$. First we define a set $\mathcal{B}$ of type constants.

$$Type\ Constants: \qquad \mathcal{B} = \{1\ ,\ *\} \cup \mathcal{T}_l \qquad \mathcal{T}_l = \bigcup_k \mathcal{T}_l^k$$

where $\mathcal{T}_l$ is the ranked set of *Label Types* with a type in $\mathcal{T}_l^k$ for each k-hyperedge nonterminal symbol that appears in the productions. These types are needed to differentiate among nonterminal edges with the same number of attachment nodes. This set will change for each particular case. For the ring-tree example we have, $\mathcal{T}_l = \{C\ ,\ P\}$. We will denote $\underbrace{1 \times 1 \times \ldots}_{n}$ as n with $0 = \epsilon$.

Now we define the set $\mathcal{C}$of term constants. We introduce two operators in the signature. The | operator is used to put together edges. The ν operator is for the new nodes created by the productions and term $\nu(\lambda n.M)$, is abbreviated as $\nu n.M$ in the style of Higher Order Abstract Syntax (HOAS) [Church, A., 1940].

$$Term\ Constants: \qquad \mathcal{C} = \{|: * \times * \rightarrow *\ ,\ \nu : (1 \rightarrow *) \rightarrow *\} \cup\ T_c\ \cup\ NT_c$$

where T_c is the set of terminal constants with a constant for each terminal edge that appears in the productions, and NT_c is the set of nonterminal constants with a constant for each nonterminal symbol that appears in the productions. These sets will change for each particular case. For the ring-tree example we have, $T_c = \{b : 3 \rightarrow *\ ,\ c : 2 \rightarrow *\}$ and $NT_c = \{c_C : C\ ,\ c_P : P\}$.

$$\begin{array}{lll} \textit{Axioms:} & p \mid q = q \mid p & \nu m.\nu n.M = \nu n.\nu m.M \\ & p \mid (q \mid r) = (p \mid q) \mid r & M \mid \nu\, n.N = \nu n.M \mid N\ ,\ n \notin fv(M) \end{array}$$

It is important to distinguish between constants of the method (i.e. $\mid$ and ν) and constants of the particular examples (i.e. term and type constants for terminal and nonterminal edge labels).

Graphs as Functions

Given the definition of the corresponding λ-signature, now we define in this section the correspondence between graphs and terms and in the next section the correspondence with productions and derivations.

There are two classes of graphs, nonterminal and terminal graphs. The graphs in the first class possibly contain nonterminal symbols, the others do not. The next theorem states the correspondence between these graphs and certain terms of the calculus. For this we first define a set of type classes used to type terms corresponding to graphs.

$$\begin{array}{c} \textit{Graph Types}: \quad \mathcal{T}_g ::= \mathcal{T}_{nodes} \rightarrow * \quad \mathcal{T}_{nodes} ::= 0 \mid 1 \mid . \ . \\ \textit{Nonterminal Edge Types}: \ \mathcal{T}_{ne} ::= \mathcal{T}_{Tnodes} \rightarrow * \ \ \mathcal{T}_{Tnodes} ::= 0 \times \mathcal{T}_l^0 \mid 1 \times \mathcal{T}_l^1 \mid .. \\ \textit{Nonterminal Graph Types}: \mathcal{T}_{ng} ::= \mathcal{T}_{nonterm} \rightarrow \mathcal{T}_g \ \mathcal{T}_{nonterm} ::= \epsilon \mid \mathcal{T}_{nonterm} \times \mathcal{T}_{ne} \end{array}$$

Class $\mathcal{T}_g$ is the class of types for terminal graphs. These graphs are represented as second order terms of type $n \rightarrow *$ with n the number of external nodes of the corresponding graph.

Class $\mathcal{T}_{ne}$ is the class of types for the first order terms that represent nonterminal edges. The type of these terms is given by the number of attachment nodes of the edge and its corresponding label type.

Class $\mathcal{T}_{ng}$ is the class of types for nonterminal graphs. These graphs are represented as higher order terms. As we mentioned, types define the interfaces, in this cases the types specify functions which take as many arguments as are the nonterminals ($\mathcal{T}_{nonterm}$) and return a value of terminal graph type ($\mathcal{T}_g$).

For example, taking the ring-tree style we can define the terms for the graphs in Figure 8.5. The terminal graph on figure *a* will have a type of class $\mathcal{T}_g$ and the nonterminal graph on figure *b* that has two non terminals will have a type of class $\mathcal{T}_{ng}$. Parenthesized node labels in the figure are just for making the correspondence with terms, but they are not needed.

$$\begin{array}{c} \textit{Terminal Graph:} \quad \lambda < n_1, n_2 > .\ \nu m_1.\nu m_2.\nu m_3. \\ c(n_1, n_2) \mid b\,(n_2, m_1, n_1) \mid b(m_2, m_1, m_3) \mid c(m_3, m_2) \quad 2 \rightarrow * \\ \\ \textit{Nonterminal Graph:} \quad \lambda < x_{C1}, x_{C2} > .\ \lambda < n_1, n_2, n_3, n_4 > .\ \nu m. \\ x_{C1}(n_1, n_2, c_C) \mid b\,(n_2, m, n_1) \mid b\,(n_3, m, n_4) \mid x_{C2}(n_4, n_3, c_C) \\ : \ (2 \times C \rightarrow *) \times (2 \times C \rightarrow *) \rightarrow (4 \rightarrow *) \end{array}$$

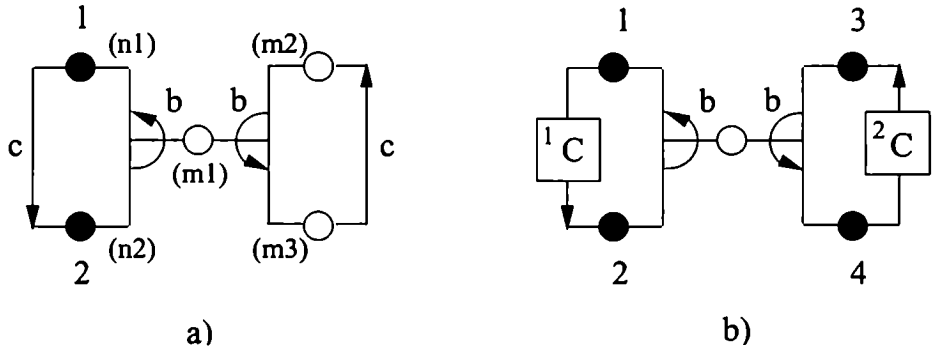

Figure 8.5: Example of terminal and nonterminal graphs

For Figure 8.5*a* we define two external nodes and for Figure 8.5*b* four external nodes. The rest of the nodes are created using the ν operator. The number of external nodes is part of the definition of a graph. We state the correspondence in the next theorem.

THEOREM 8.1. [Graphs and Terms Correspondence] *There is a one-to-one correspondence between graphs (up to isomorphism) and terms (up to equivalence) of type $\mathcal{T}_g$ for terminal graphs and $\mathcal{T}_{ng}$ for nonterminal graphs.*

Proof. *The first part of the theorem, from λ-terms to graphs (with respect to the corresponding λ-signature), is proved by construction defining the following translation function $[\![-]\!]$:*

First, as presented in Section 8.2.1, by definition of the λ-signature we have that, for each label type $\mathcal{T}_l^k$ in $\mathcal{T}_l$ there is a k-hyperedge nonterminal symbol in NT, *and for each term constant of type $k \to *$ in T_c there is a k-hyperedge terminal symbol in* T.

For terms of type $\mathcal{T}_g$:

- $[\![[\lambda < n_1, \ldots, n_r > . \ \nu m_1 \ldots \nu m_s . \ t \quad r \to *]\!] =$
  $\langle \{n_1, \ldots, n_r, m_1, \ldots, m_s\}, E_t, att_t, \{n_1, \ldots, n_r\}, lab_{LEt}, lab_{LNt} \rangle,$
  where E_t, att_t, $lab_{LEt} : E_t \to T$, lab_{LNt} are given as expected by the structure of subterm t (constructed using operator $|$) and the definition of the λ-calculus signature.

For terms of type $\mathcal{T}_{ng}$:

- $[\![[\lambda < x_1, \ldots, x_n > . \ \lambda < n_1, \ldots, n_r > . \ \nu m_1 \ldots \nu m_s . \ t \ : \ T_{x1} \times \ldots \times T_{xn} \to (r \to *)]\!]$
  is defined in the same way as for terms of type $\mathcal{T}_g$, but in this case $lab_{LEt} : E_t \to T \cup NT$ (i.e. Nonterminal graphs).

Conversely, we prove again the second part of the theorem by construction in a similar way as the first part. The signature of the λ-calculus is constructed from the terminal and nonterminal symbol sets of graphs defining the corresponding terms.

- *For a graph G with only terminal labels and n external nodes, the corresponding term t_G is of type $n \to *$.*

- *For a graph G with n edges with nonterminal labels and r external nodes, the corresponding term t_G is of type $T_{x1} \times \ldots \times T_{xn} \rightarrow (r \rightarrow *)$, with T_{xi} of the corresponding rank.*

Completing the proof of the isomorphism, it is obvious that the compositions of the corresponding translations result in identities. □

Productions and Derivations as Functions

In this section we define the correspondence of productions and derivations with terms. We first introduce the use of an environment in the calculus representing the notion of a HR system. Then, a set of type classes is defined to type terms corresponding to productions and derivations. As in the previous section, a theorem for the correspondence is presented.

An environment ρ is a set of variable definitions giving for each variable Var_i a corresponding term $t_i : \tau_i$ (i.e. $\rho = [t_1/Var_1, ..t_n/Var_n]$). This allows to include in terms the variables in the environment, variables being typed with the types of their interpretation. The application ρM over a term M is a substitution where each free variable of M is replaced by its definition in ρ. So, given a fixed λ-signature corresponding to an alphabet of nonterminal and terminal symbols NT and T of a graph rewriting system $HRG =< G_0, P, NT, T >$, we define a corresponding environment that contains the definitions of the initial graph G_0 and the productions in P. This environment defines the interpretation of terms for graphs and derivations over the grammar produced by HRG.

Now we define the type classes needed to type productions and derivations.

$$\textit{Production Types}: \quad \mathcal{T}_p ::= \mathcal{T}_{nonterm} \rightarrow \mathcal{T}_{ne}$$

The class $\mathcal{T}_p$ defines types for productions which are represented as higher order terms. The input interface is the nonterminal symbol to be replaced on the left side and the output interface is the set of nonterminals on the graph on the right side. It should be observed that, in the usual graphical representation, production "arguments" are on the right, while the "result" is on the left. But this representation is *inverted* in functional types, where the variable types are on the left and result types are on the right of the arrows. Thus a production is a function which takes a tuple of terms for the nonterminals on the right side ($\mathcal{T}_{nonterm}$) and returns a term of the type of the nonterminal on the left side ($\mathcal{T}_{ne}$).

As an example, we define the environment of the ring-tree with the terms of the productions and initial graph in Figure 8.4*a*.

$$Brother \stackrel{def}{=} \lambda < x_{C1}, x_{C2} > . \lambda < n_1, n_2, n_C > .\nu m.$$
$$x_{C1}(m, n_2, c_C) \,|x\, _{C2}(n_1, m, c_C) : (2 \times C \to *) \times (2 \times C \to *) \to (2 \times C \to *)$$

$$Port \stackrel{def}{=} \lambda < x_C, x_P > . \lambda < n_1, n_2, n_C > . \nu m.\ x_C(m, n_2, c_C) \mid x_P(n_1, m, c_P)$$
$$: (2 \times C \to *) \times (2 \times P \to *) \to (2 \times C \to *)$$

$$Bridge \stackrel{def}{=} \lambda < x_C > . \lambda < n_1, n_2, n_P > . \nu m_1.\nu m_2.\nu m_3.$$
$$b(n_1, m_1, n_2) \,|b\, (m_2, m_1, m_3) \,|x\, _C(m_3, m_2, c_C) : (2 \times C \to *) \to (2 \times P \to *)$$

$$Component \stackrel{def}{=} \lambda <> . \lambda < n_1, n_2, n_C > . c(n_1, n_2) \ :\epsilon \to (2 \times C \to *)$$

$$Init \stackrel{def}{=} \lambda < x_C > .\lambda < n > . x_C(n, n, c_C) \ : (2 \times C \to *) \to (1 \to *)$$

Derivation Types:

In the case of derivations, we have derivations that start with the initial graph of the system and end in a terminal graph and partial derivations that start with the initial graph of the system and end in a nonterminal graph. Hence, the derivation type is the graph type $\mathcal{T}_g$ and the partial derivation type is the nonterminal graph type $\mathcal{T}_{ng}$.

Then by definition a (partial derivation) derivation is a sequence of applications of productions of the form $G_0 \Rightarrow_{p1} G_1 \Rightarrow_{p2} \ldots \Rightarrow_{pn} G_n$, where $p_1, \ldots, p_n$ are in P and G_n is a (nonterminal) terminal graph. More formally each derivation step $G_i \Rightarrow_{pi+1} G_{i+1}$ can be seen as a parallel composition of the application of production p_{i+1} over a nonterminal symbol L_{i+1} that appears on G_i with a set of identities for the rest of nonterminal edges in G_i. Then, the corresponding term that represents the derivation can be described as a sequence of functional compositions of production names (starting from the initial graph) and identities composed in parallel. The operations of identity and sequential and parallel composition over closed terms id_τ, $(M \circ N)$ and $(M \otimes N)$ are defined as:

Identity

$$id_\tau \ = \ \lambda x : \tau \,.\, x \ : \tau \ \to \tau$$

Sequential Composition

$$\frac{M:\sigma \to \delta \ ,\ N:\tau \to \sigma}{M \circ N \ = \ \lambda x{:}\tau.\ M(Nx) \ :\tau \to \delta}$$

Parallel Composition

$$\frac{M:\tau \to \sigma \ ,\ N:\delta \to \pi}{M \otimes N \ = \ \lambda <x,y>:\tau \times \delta.<Mx,\ Ny> \ :(\tau \times \delta) \to (\sigma \times \pi)}$$

For example, the term for the partial derivation of Figure 8.4*b* is,

$$Init \circ Brother \circ (id_{2\times C\to *} \otimes Brother) \circ (Port \otimes id_{2\times C\to *} \otimes id_{2\times C\to *})$$
$$\circ(id_{2\times C\to *} \otimes Bridge \otimes id_{2\times C\to *} \otimes id_{2\times C\to *})$$
$$: ((2 \times C \to *) \times (2 \times C \to *) \times (2 \times C \to *) \times (2 \times C \to *) \to (1 \to *))$$

Note that the names of the productions and the initial graph are free variables which will be replaced by their definitions in the corresponding environment. The type of the derivation term is given by the definitions in ρ.

As it is already mentioned, a derivation is characterized by a derivation tree. For each derivation tree there is a set of equivalent sequential derivations. So derivations can be characterized also as a term based on its derivation tree. The term based on derivation tree (8.1) for derivation of Figure 8.4*b* is,

$$\lambda < x_{C1}, x_{C2}, x_{C3}, x_{C4} > .$$
$$(Init < Brother < Port < x_{C1}, Bridge < x_{C2} >>, Brother < x_{C3}, x_{C4} >>>)$$
$$: ((2 \times C \to *) \times (2 \times C \to *) \times (2 \times C \to *) \times (2 \times C \to *) \to (1 \to *))$$

Note that this term and the previous one are equivalent.

THEOREM 8.2. [Productions and Derivations Correspondence] *Given a HR Grammar* $HRG = \langle G_0, P, NT, T \rangle$, *there is a one-to-one correspondence between productions and partial derivations (up to isomorphism) over HRG and terms (up to equivalence) respectively of the form,*

1. $\lambda < x_1, .., x_n > .\lambda < n_1, .., n_k, n_{k+1} > . \ t$
   $: (i_1 \times LT^{i1} \to *) \times \ldots \times (i_n \times LT^{in} \to *) \to (k \times LT^k \to *)$
   where term t can only contain ν *but not* λ *binders or* $\circ$ *and* $\otimes$ *operators.*

2. $init \circ \ t_1 \circ \ldots \circ t_n : \ \tau$ *with* $\tau \in \mathcal{T}_{ng}$
   where each t_i *is a parallel composition of a production name in the environment of HRG with a set of identities.*

Proof. *For the first part of the theorem, from* λ*-terms to productions and partial derivations (with respect to the corresponding* λ*-signature) we have:*
For productions we have that a production is a pair (L, R) *where* L *is a hyperedge (label) and* R *is a graph. Then, we define a translation function* $[\![-]\!]$ *in a similar way as in Theorem 8.1 and construct the two graphs.*

- $[\![\lambda < x_1, .., x_n > .\lambda < n_1, .., n_k, n_{k+1} > . \ t$
  $: (i_1 \times LT^{i1} \to *) \times \ldots \times (i_n \times LT^{in} \to *) \to (k \times LT^k \to *)]\!] = (L, R)$ *with*

$$L = \langle \{n_1, \ldots, n_k\}, \{e_{LTk}\}, \{(e_{LTk}, < n_1, \ldots, n_k >)\}, \{n_1, \ldots, n_k\}, \{(e_{LTk}, LT_k)\}, \{(n_1, 1), \ldots, (n_k, k)\}\rangle$$

$$R = \langle \{n_1, \ldots, n_k\} \cup N_t, \{e_{LTi1}, \ldots, e_{LTin}\} \cup E_t, \{(e_{LTi1}, \vec{n_{i1}}), \ldots, (e_{LTin}, \vec{n_{in}})\} \cup att_t, \{n_1, \ldots, n_k\}, \{(e_{LTi1}, LT_{i1}), \ldots, (e_{LTin}, LT_{in})\} \cup lab_{LEt}, lab_{LNt}\rangle$$

where (given as expected by the structure of subterm t) N_t *are the new nodes created by the production (defined by* ν*),* E_t *are the edges with terminal symbols,* att_t *are the assignments of attachment nodes for edges with terminal symbols,* $lab_{LEt} : E_t \to T$ *is the labelling function for edges with terminal symbols,* lab_{LNt} *is the labelling of nodes and* $\vec{n_{ij}}$ *are the attachment nodes for the nonterminal edges.*

For partial derivations we can prove by induction on the structure of terms.
Base Case:

- *For init we have the translation defined in Theorem 8.1 where the term with type in* $\mathcal{T}_{ng}$ *corresponds to the initial (nonterminal) graph.*

Inductive Hypothesis: *Given a term* $init \circ t_1 \circ \ldots \circ t_q$, *where* $init \circ t_1 \circ \ldots \circ t_{q-1}$ *with type in* $\mathcal{T}_{ng}$ *corresponds to a partial derivation* $G_0 \Rightarrow_{p1} \ldots \Rightarrow_{p(q-1)} G_{q-1}$ *and* t_q *corresponds to the parallel composition of a set of identities and a term for a production* p_q, *then* $init \circ t_1 \circ \ldots \circ t_q$ *corresponds to the partial derivation* $G_0 \Rightarrow_{p1} \ldots \Rightarrow_{p(q-1)} G_{q-1} \Rightarrow_{pq} G_q$

- *We have term* $init \circ t_1 \circ \ldots \circ t_{q-1}$ *of type* $(i_1 \times LT^{i1} \rightarrow *) \times \ldots \times (i_n \times LT^{in} \rightarrow *) \rightarrow (r \rightarrow *)$ *which by Inductive Hypothesis corresponds to derivation* $G_0 \Rightarrow_{p1} \ldots \Rightarrow_{p(q-1)} G_{q-1}$ *and* LT^{ij} *correspond to the nonterminal symbols of* G_{q-1}.

  Then, we have term t_n *of type* $(h_1 \times LT^{h1} \rightarrow *) \times \ldots \times (h_m \times LT^{hm} \rightarrow *) \rightarrow ((i_1 \times LT^{i1} \rightarrow *) \times \ldots \times (i_n \times LT^{in} \rightarrow *))$.

  Then, the sequential composition $init \circ t_1 \circ \ldots \circ t_{q-1} \circ t_q$ *has type* $(h_1 \times LT^{h1} \rightarrow *) \times \ldots \times (h_m \times LT^{hm} \rightarrow *) \rightarrow (r \rightarrow *)$ *where, by the translation function defined above from terms to productions, results in the addition of derivation step* $G_{q-1} \Rightarrow_{p(q-1)} G_q$ *to the partial derivation for* G_{q-1}.

Conversely, for productions we prove again the second part of the theorem by construction in a similar way as the first part.

- *For a production* (L, R) *we can construct the corresponding subterm* t *from the structure of graph* R *using* ν *and* $|$ *and the corresponding signature. Then, we add the binding* $\lambda < n_1, \ldots, n_k, n_{k+1} >$ *for the external nodes* $(n_1, \ldots, n_k)$ *and the labelling* n_{k+1} *of* L; *and the binding* $\lambda < x_1, \ldots, x_n >$*for the nonterminal symbols of* R.

- *For a derivation* $G_0 \Rightarrow_{p1} \ldots \Rightarrow_{pn} G_n$ *we give the environment with the definition of Init for the translation for* G_0 *and* $[t_1/Var_1, ..t_n/Var_n]$ *for productions* $p_1, \ldots, p_n$. *Then, to construct the term for the derivation starting with Init, for each derivation step* $G_i \Rightarrow_{pi} G_{i+1}$ *we apply (the sequential composition of) the parallel composition of production name* Var_i *with the corresponding identities for the symbols that are not rewritten. The ordering of each parallel compositions is given by the ordering of the nonterminals symbols of graph* G_i.

Completing the proof of the isomorphism, it is obvious that the compositions of the corresponding translations result in identities. □

An interesting point is if we take derivations and abstract the names of the productions (i.e. bound free variables with a λ binder). In this way, we obtain (with respect to a signature) a production independent abstract derivation tree, with the names as placeholders, that can be applied over different grammars satisfying the production types. We define the corresponding type classes.
Abstract Derivation Types:

$$\mathcal{T}_{ad} ::= \mathcal{T}_{prod} \rightarrow \mathcal{T}_g \qquad \mathcal{T}_{prod} ::= \mathcal{T}_{prod} \times \mathcal{T}_p$$

Abstract Partial Derivation Types:

$$\mathcal{T}_{apd} ::= \mathcal{T}_{prod0} \rightarrow \mathcal{T}_{ng} \qquad \mathcal{T}_{prod0} ::= \epsilon \mid \mathcal{T}_{prod0} \times \mathcal{T}_{p}$$

Classes $\mathcal{T}_{ad}$ and $\mathcal{T}_{apd}$ take a tuple of productions and returns a terminal and nonterminal graph, respectively.

Abstract Derivation:

$$\begin{gathered}\lambda < Brother, Port, Bridge > . \lambda < x_{C1}, x_{C2}, x_{C3}, x_{C4} > . \\ Brother(Brother(x_{C1}, x_{C2}), Port(x_{C3}, Bridge(x_{C4}))) \\ : ((2 \times C \rightarrow *) \times (2 \times C \rightarrow *) \rightarrow (2 \times C \rightarrow *)) \times \\ ((2 \times P \rightarrow *) \times (2 \times C \rightarrow *) \rightarrow (2 \times C \rightarrow *)) \times ((2 \times C \rightarrow *) \rightarrow (2 \times P \rightarrow *)) \rightarrow \\ ((2 \times C \rightarrow *) \times (2 \times C \rightarrow *) \times (2 \times C \rightarrow *) \times (2 \times C \rightarrow *) \rightarrow (2 \times C \rightarrow *))\end{gathered}$$

8.2.2 Derivation Transformations

Once derivations are represented as terms, what is needed is a notion of rewriting system which ensures that derivations are rewritten into derivations. In the area of software architecture, this could be used to specify complex reconfigurations which however transform systems which are instances of a particular style into instances of the same style.

A rather general strategy to achieve this aim is to require that a set of rewrite rules is specified, where each rule $L \Rightarrow R$ transforms a derivation segment L of the grammar into a derivation segment R. Applying the rule would mean to replace L with R in the derivation of some graph G, obtaining a possibly very different graph G', which however is automatically guaranteed to belong to the language generated by the grammar.

It is easy to see that the ordinary notion of graph derivation, i.e. a sequence of graphs and productions, could not serve this purpose. In fact, a derivation segment is limited by (we could also say: is typed with) two graphs, and could be reasonably replaced only with a segment typed with the same two graphs, which thus would eventually produce the same result.

To allow for a more expressive concept, we need to equip derivation segments with a less stringent notion of type. This is exactly what we have achieved with the notion of derivation introduced in the previous section. In our running example

$$\begin{gathered}Init \circ Brother \circ (id_{2\times C\rightarrow *} \otimes Brother) \circ (Port \otimes id_{2\times C\rightarrow *} \otimes id_{2\times C\rightarrow *}) \\ \circ (id_{2\times C\rightarrow *} \otimes Bridge \otimes id_{2\times C\rightarrow *} \otimes id_{2\times C\rightarrow *}) \\ : ((2 \times C \rightarrow *) \times (2 \times C \rightarrow *) \times (2 \times C \rightarrow *) \times (2 \times C \rightarrow *) \rightarrow (1 \rightarrow *))\end{gathered}$$

we have a derivation composed of four steps, where the composition of the steps is achieved via the second order function composition operator $\circ$. The composition is possible if the (functional) type of the range of the first function is the same as the type of the domain of the second. For instance for the last $\circ$ operator in the above term, the type is:

$$(2 \times C \rightarrow *) \times (2 \times P \rightarrow *) \times (2 \times C \rightarrow *) \times (2 \times C \rightarrow *)$$

which clearly expresses the labels and arities of the nonterminals still to be expanded at that stage of the derivation.

It is now clear what a rewrite rule $L \Rightarrow R$ is, which is adequate for our aim: both L and R are derivations segments of the given grammar with the same initial and final functional types.

For instance, we may want a transformation over the ring-tree style which specifies a reconfiguration that allows to break any ring of the tree into two smaller rings obtaining a new consistent instance of the style. The transformation is specified by the following rewrite rule:

$$\begin{array}{c} \mathit{Brother} \;\Rightarrow\; \mathit{Port} \circ (\mathit{id}_{2\times C\to *} \otimes \mathit{Bridge}) \\ ((2 \times C \to *) \times (2 \times C \to *) \to (1 \times C \to *)) \end{array}$$

which, in words, replaces the creation of a new **C** component with the creation of a **P** component and with its evolution into a bridge and a new **C** component. Notice that replacing one derivation segment with the other one leaves the rest of the derivation as it was, in this case the evolution of the two **C** nonterminals.

Figure 8.6 shows in graphical form the initial and final derivation segments of the transformation and the transformation applied over the derivation on Figure 8.4*b*.

Note that transformations of the above kind can be applied in sequence to build more complex effects, and also they can be applied simultaneously over non overlapping segments of a derivation.

Derivation transformations can be used in the area of software architecture to specify complex reconfigurations of styles. This is an important issue, especially for distributed systems, allowing the designer to have a library of reconfigurations and to select the reconfigurations that a system will be allowed to undergo. To specify a reconfiguration of a style, a designer has to find the initial and final derivation segments over the style grammar. Once he has the transformation it can be applied over any instance of that style. The language of graphs gives a visual presentation of styles and transformations and on the other side the correspondence to lambda calculus gives the semantics and shows that the method is implementable.

How could this notion of rewrite system be fully formalized, generalized and implemented? An obvious option is to resort to type theory and to some higher order theorem prover, eg. Isabelle. In this setting, the power of higher order graph rewriting could be fully exploited, allowing to parameterize the design process with component and connector features which could be specified later, still guaranteeing consistency.

Another option would be to employ rewriting logic by Jose Meseguer [Meseguer, J., 1992], which is equipped with a rather general theory and is the base of several implementations, notably *Maude* [Clavel, M.G. *et al.*, 1998] at SRI International. True, rewriting logic is presently only first order, but general conditional and nonconditional axioms are allowed. In our approach, it should be possible to model our higher order derivation trees as first order terms, provided that suitable axioms are added that identify all the derivation trees generating the same graph.

Tile logic [Gadducci, F. and Montanari, U., 1999] would be a more concrete alternative. Tile logic is based on rewriting rules with side effects and generalizes both SOS and rewriting logic (in the nonconditional case). Side effects can be used to synchronize several rules, thus building complex atomic transformations reminiscent of nondeterministic transactions. Tile logic has been equipped with a higher order theory [Bruni,

R. and Montanari, U., 1999], where both configurations to be rewritten and side effects are simply typed lambda terms. Stripping side effects from higher order tile logic, one would get a higher order version of rewriting logic, as needed here. However, also the additional features of tile logic could be very useful. For instance, several complex transformations of practical interest in software architecture cannot be broken down into sequences of simpler transformations, since the intermediate configurations would not satisfy the requirements of the style. The synchronized transformations which can be expressed in tile logic would not require the consistency with the style of the intermediate configurations, and would still make sure that when the transformation is completed, consistency is fully reinstated.

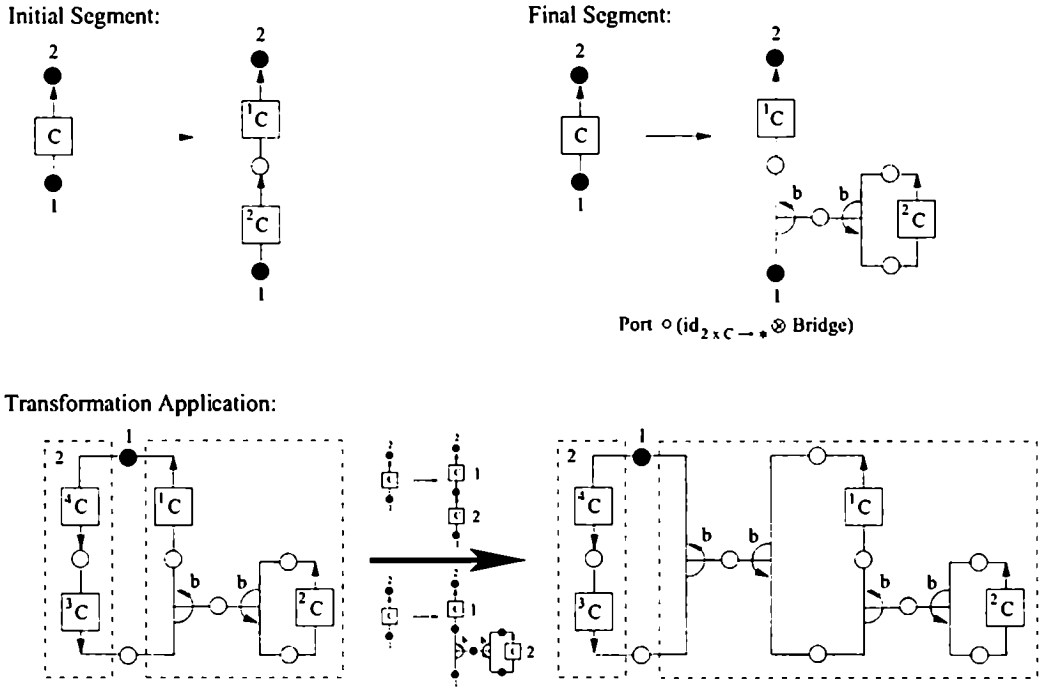

Figure 8.6: A transformation

Chapter 9

Consistent Transformations via Inconsistent Steps

9.1 Building Consistent Transformations

This approach is useful for many types of transformations but there may exist complex transformations that cannot be broken down into sequences of simpler transformations, since the intermediate configurations would not satisfy the requirements of the style. And also, a designer may want to specify a reconfiguration in a more constructive (and maybe more intuitive) way with intermediate steps that may not end in valid configurations of the style.

To deal with these cases, we can use again the new definition of derivation. The designer is allowed to specify a complex reconfiguration as the composition of smaller ones that can have intermediate non-valid configurations. For example, Figure 9.1a. shows a geometrical representation of the transformation that is obtained by diagram pasting of three smaller ones (Figure 9.1b.). The three transformations in Figure 9.1b. are obtained defining two new productions (I_1 and I_2) that do not belong to the valid productions of the style. This means that the application of this type of productions can generate graphs that do not belong to the valid style grammar. Looking at the geometrical diagram, non-valid productions correspond to vertical arrows. For example, in Figure 9.1b. we have three transformations of the form:

$$\begin{array}{lcll} P_{i+1} \bullet I_1 & \Rightarrow & P'_{i+1} & :T_i \rightarrow T'_{i+1} \\ P_{i+2} \bullet I_2 & \Rightarrow & I_1 \bullet P'_{i+2} & :T_{i+1} \rightarrow T'_{i+2} \\ P_{i+3} & \Rightarrow & I_2 \bullet P'_{i+3} & :T_{i+2} \rightarrow T_{i+3} \end{array}$$

The pasting of these transformations is given by matching of the interfaces defined by productions I_1 and I_2 and their types. But, in spite of the fact that non-valid productions can be applied, given that they are typed (by nonterminals), the final transformation

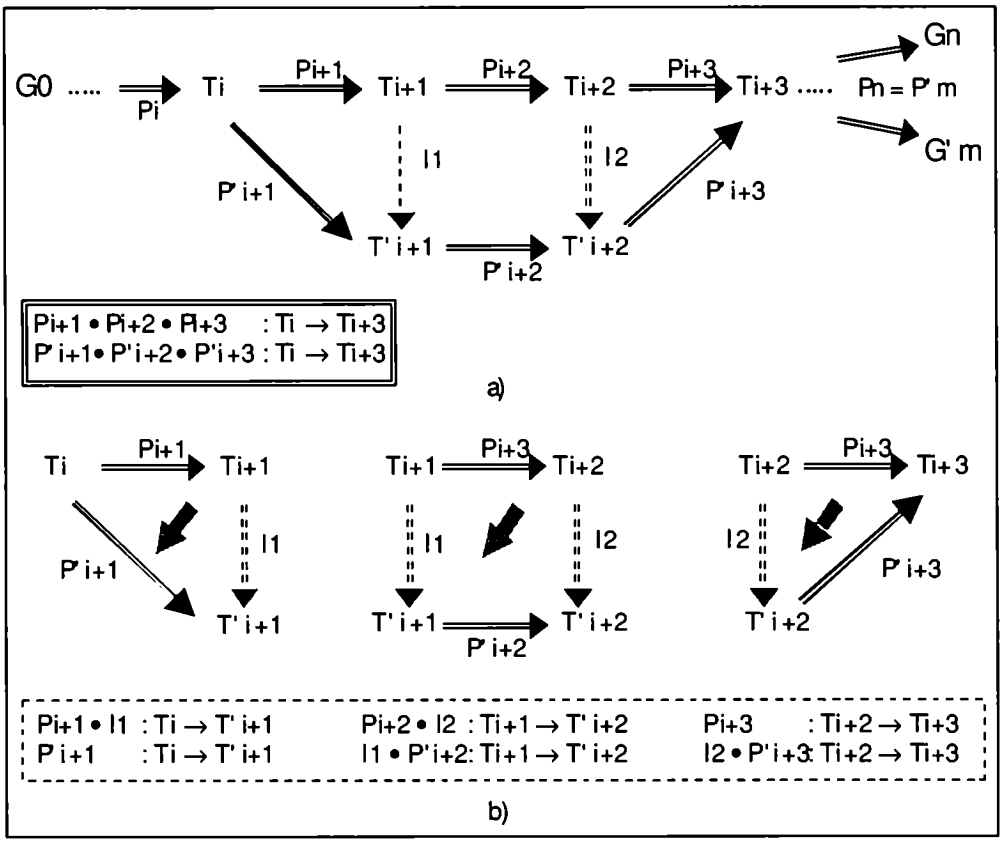

Figure 9.1: Transformations by construction. a) A consistent transformation by composition of three inconsistent ones. b) Inconsistent transformations.

is a valid transformation between consistent derivation segments (they do not contain non-valid productions).

In the case of Figure 9.1a. we have transformation:

$$P_{i+1} \bullet P_{i+2} \bullet P_{i+3} \Rightarrow P'_{i+1} \bullet P'_{i+2} \bullet P'_{i+3} : T_i \rightarrow T_{i+3}$$

This final valid transformation is the result of looking at the upper and lower derivation segments obtained after putting together the smaller transformations. But we can also see this transformation as a sequence of steps through the inconsistent intermediate states using the smaller transformations. In the example we have (graphically you can follow the arrows of Figure 9.1a):

$$P_{i+1} \bullet P_{i+2} \bullet P_{i+3} \Rightarrow P_{i+1} \bullet P_{i+2} \bullet I_2 \bullet P'_{i+3} \Rightarrow P_{i+1} \bullet I_1 \bullet P'_{i+2} \bullet P'_{i+3} \Rightarrow P'_{i+1} \bullet P'_{i+2} \bullet P'_{i+3}$$

With this constructive method we can incrementally specify a reconfiguration with possible intermediate inconsistent states, but at the end when the upper and lower

derivation segments do not contain non-valid productions we are sure that the resulting transformation is a valid one. You should notice that it is possible not only to paste transformations horizontally, but also vertically. And of course as before, simultaneous (parallel) application is possible.

9.1.1 Example

To exemplify the method we present a transformation over the TRMCS that moves a **User** from one **Router** to another. But in this case we want to specify that the **User** moves from the first **Router**, to the **Server**, and afterwards to the second **Router**. For this transformation we define in Figure 9.2a. production *Inconsistent User*, that attaches a **User** to the **Server**. The valid transformation is composed of two smaller inconsistent transformations. Figure 9.2b. shows the diagram representation of the whole transformation typed by the corresponding nonterminal. The first transformation in Figure 9.2c. changes the creation of a **User** attached in the first **Router** (*CU*) by the creation of a **User** attached to the **Server** (*IU*) using the non-valid production. And the second transformation takes the creation of the **User** attached to the **Server** and changes it by the creation of a **User** attached to the second **Router**. Figure 9.3 shows the application of the transformation to a derivation (identities are omitted for simplicity).

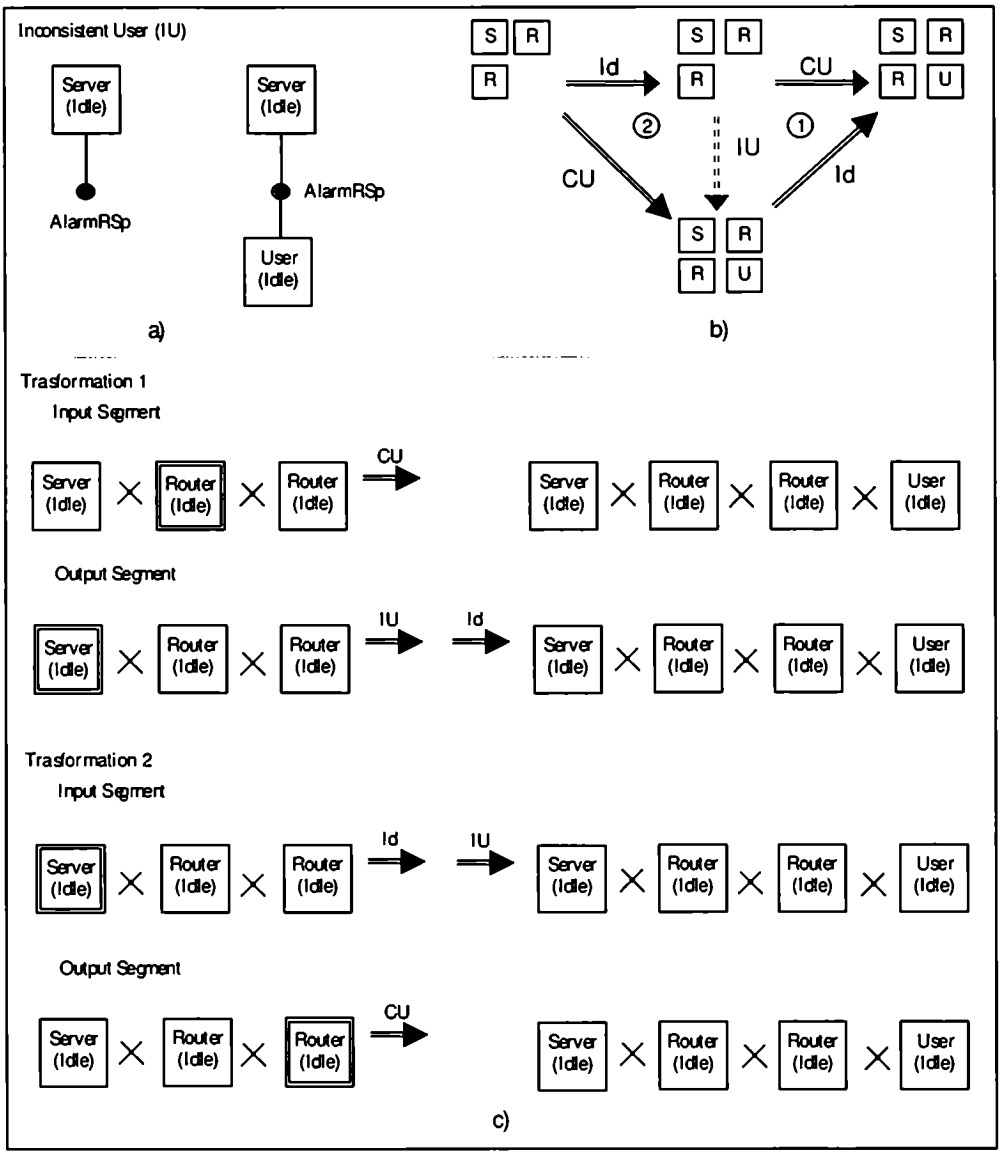

Figure 9.2: Transformation that moves a User through the Server. a) Non-valid Production. b) Complete Transformation. c) Inconsistent Transformations.

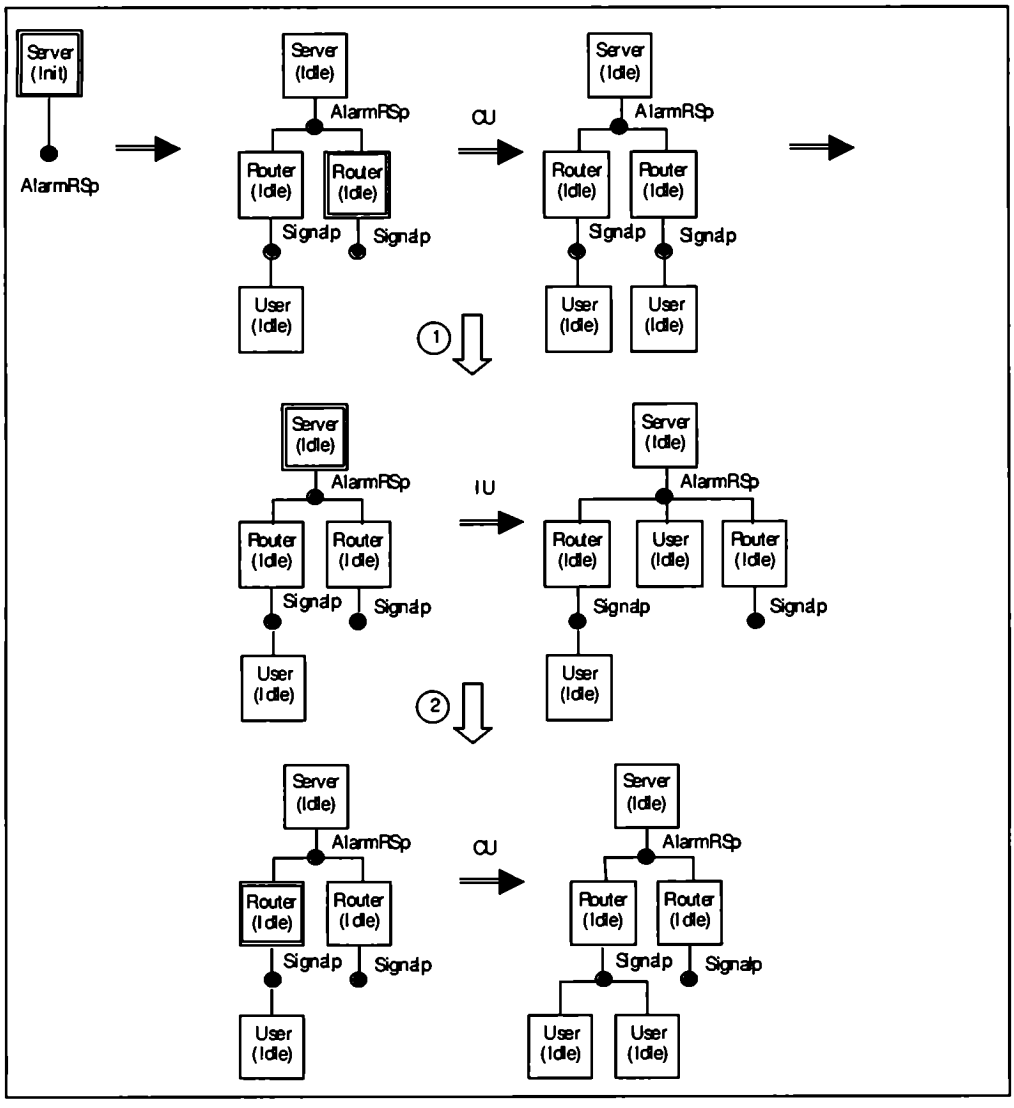

Figure 9.3: Application of the Transformation Rule.

Part VII

Conclusions

Chapter 10

Conclusions and Future Work

The main goal of this thesis was to introduce graph grammars as a general formal framework supporting graphical notations for the description of software architecture styles. Grammars characterize classes of graphs that represent the instances of a style. In this way, the commonalties that identify a style are described as productions of the grammar. Graphs represent the static configuration of systems while the notion of graph transformation is used for the description of the evolution of the architecture instances. Also, the use of grammars allows us specifying not only the general static configuration of the style but it makes possible to include the communication pattern and dynamic reconfiguration at the level of style while achieving a separation of coordination and computation.

Specifically, in Parts III and IV, we have introduced the framework following a self-organising approach using *HR grammars* [Drewes, F. *et al.*, 1997] together with *synchronized rewriting* (SHR systems [Degano, P. and Montanari, U., 1987; Montanari, U. *et al.*, 1999]) and the addition of name mobility. In this way we were able to add the description of interactions and dynamic reconfigurations of software architecture styles. Based on the rewriting system specified by the grammars we describe the style as a set of productions that model the initial structural topology of the architecture, the laws governing the topological changes, and its communication pattern. The use of synchronizing conditions to model coordination of components allows a clear description of component interactions and controlled dynamics and the application of the distributed solutions for the rule-matching problem. Also, context-free rules are a natural way for modeling the behavior of components independently of each other allowing a distributed implementation.

To complement the above ideas, in Part VII we proposed an alternative method to SHR systems where complex reconfigurations are specified as transformations over the derivations of a grammar. In this way, once a transformation is obtained it is assured that it is a consistent reconfiguration with respect to the style. The main difference of this method with respect to the one presented in Part IV using name mobility, is that the method in Part IV is more dynamic in the sense that it applies to running open-ended systems without global control except for synchronization, whereas the approach with

transformations may be useful for working at the level of blueprints, i.e. it rearranges the design steps of the system to produce a different but consistent system. Thus, the latter method can be applied to specify very general kinds of reconfigurations and mobility (as it is shown in the thesis examples), but it requires a global knowledge of system structure.

We represent hypergraphs and SHR systems in textual form using syntactic judgements. This allows the clear separation of rewriting and coordination and the introduction of various synchronization mechanisms as suitable (mobile) synchronization algebras. Specifically, we present the inference rules in the SOS style for *Hoare* (CSP) and *Milner* (CCS, π-calculus) synchronization algebras. However, we extend process algebras in that we allow synchronizations of any number of partners at the same time. Synchronizing conditions for mobility are solved via unification. We have to mention that the initial work of SHR [Degano, P. and Montanari, U., 1987; Montanari, U. and Rossi, F., 1999; Montanari, U. *et al.*, 1999] only uses Hoare synchronization (without mobility).

But also, we realized that the results we have obtained go beyond the domain of software architectures. Specially, the addition of name mobility to SHR systems, their formalization as syntactic judgements that let us introduce various synchronization mechanisms as suitable (mobile) synchronization algebras (for example the inference rules in the SOS style for *Hoare* and *Milner* synchronization algebras in chapter 6), and the correspondence proof of SHR systems with the π-Calculus (see Part V), are a strong support for graphical formal languages as a next step in the high-level description of distributed, concurrent and mobile systems. With respect to this, we have commented the related work (see Section 1.4) in other areas than software architecture that have been derived from our research ([König, B. and Montanari, U., 2001; Ferrari, G.L. *et al.*, 2001; De Nicola, R. *et al.*, 2003; Lanese, I. and Montanari, U., 2002]).

As future work we are interested in various possible directions.

First, one point to note is that we only describe flat architectures, but we think that our approach can be easily extended to cope with hierarchical graphs. In this respect, we can mention specially the research of [Drewes, F. *et al.*, 2000] on hierarchical HR systems as a possible starting point of our work. Also, in the specific area of software architecture, it would be interesting to study the specification of architectural connectors and their possible integration (and corresponding consequences) as primitive entities of the model.

For the formal side of our work, we are interested in continue the study of the expressive power of this model, investigate its abstract semantics and develop new synchronization mechanisms. From the application point of view we want to see the usefulness of the approach for the formalization of more specific domains with an inherent self-orginising strategy, as for example peer-to-peer systems. And obviously, it is clear that a necessary next step includes the implementation of these ideas and investigate techniques to analyze system properties over the graph derivations such as, invariant checking, reachability and static analysis.

With respect to the work on higher-order HR systems it should be noted that the translation from graph representation to terms can be automated and that the lambda representation not only gives a semantics for the graph rewriting systems, but also shows

that the method is implementable. As already mentioned, full formalization of transformations should be investigated combining the use of synchronized rewriting. The synchronized transformations can be used to formalized the ideas introduced in Chapter 9 for specifying consistent transformation with intermediate inconsistent steps. In Chapter 9 we mentioned *Tile logic* [Gadducci, F. and Montanari, U., 1999] as a concrete alternative. The synchronized transformations which can be expressed in tile logic would not require the consistency with the style of the intermediate configurations, and would still make sure that when the transformation is completed, consistency is fully reinstated. Finally, an interesting topic of research is that the use of λ-calculus introduces the idea of *higher order graph rewriting* allowing the possibility of parameterizing the design process with component and connector features which could be specified later, still guaranteeing consistency.

Bibliography

[Allen, R. and Garlan, D., 1992] Allen, R. and Garlan, D. A formal approach to software architectures. In *IFIP Congress*, 1992.

[Allen, R. and Garlan, D., 1994] Allen, R. and Garlan, D. Formalizing architectural connection. In *16th International Conference on Software Engineering*, 1994.

[Bass, L. *et al.*, 1998] Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice*. SEI Series. Addison-Wesley, 1998.

[Booch, G. *et al.*, 1999] Booch, G., Rumbaugh, J., and Jacobson, I. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[Bruni, R. and Montanari, U., 1999] Bruni, R. and Montanari, U. Cartesian closed double categories, their lambda-notation, and the pi-calculus, invited talk. In *14th Symposium on Logic in Computer Science*. IEEE Computer Society, 1999.

[Church, A., 1940] Church, A. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–58, 1940.

[Clavel, M.G. *et al.*, 1998] Clavel, M.G., Duran, F., Eker, S., Lincoln, P., and Meseguer, J. *An Introduction to Maude (Beta Version)*. SRI International, 1998.

[De Nicola, R. *et al.*, 2003] De Nicola, R., Ferrari, G.L., Montanari, U., Rosario, R., and Tuosto, E. A formal basis for reasoning on programmable qos. Lecture Notes in Computer Science. Springer-Verlag, 2003. To appear.

[Degano, P. and Montanari, U., 1987] Degano, P. and Montanari, U. A model of distributed systems based on graph rewriting. *Journal of the ACM*, 34(2):411–449, 1987.

[Drewes, F. *et al.*, 1997] Drewes, F., Kreowski, H-J., and Hable, A. Hyperedge replacement graph grammars. In *[Rozenberg, 1997]*, chapter 2. World Scientific, 1997.

[Drewes, F. *et al.*, 2000] Drewes, F., Hoffmann, B., and Plump, D. Hierarchical graph transformation. In *International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2000)*, volume 1784 of *Lecture Notes in Computer Science*, pages 98–113. Springer-Verlag, 2000.

[Ehrig, H. *et al.*, 1999a] Ehrig, H., Engels G., Kreowski, H-J, and Rozenberg, G., editors. *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools*, volume 2. World Scientific, 1999.

[Ehrig, H. *et al.*, 1999b] Ehrig, H., Kreowski, H.-J., Montanari, U., and Rozenberg, G., editors. *Handbook of Graph Grammars and Computing by Graph Transformation: : Concurrency, Parallelism, and Distribution*, volume 3. World Scientific, 1999.

[Engels, G. and Heckel, R., 2000] Engels, G. and Heckel, R. Graph transformation as unifying formal framework for system modeling and model evolution. In *27th International Colloquium on Automata, Languages and Programming (ICALP'2000). Invited talk*, volume 1853 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[Ferrari, G.L. *et al.*, 2001] Ferrari, G.L., Montanari, U., and Tuosto, E. A lts semantics of ambients via graph synchronization with mobility. In Restivo, A., Ronchi Della Rocca, S., and Roversi, L., editors, *Seventh Italian Conference on Theoretical Computer Science (ICTCS 2001)*, volume 2202 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, October 2001.

[Fradet, P. *et al.*, 1999] Fradet, P., Le Métayer, D., and Périn, M. Consistency checking for multiple view software architectures. In *Joint 7th European Software Engineering Conference (ESEC) and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'7)*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[Gadducci, F. and Montanari, U., 1999] Gadducci, F. and Montanari, U. Proof, language and interaction: Essays in honour of robin milner. ver datos actuales !!!!!!! In Plotkin, G., Stirling, C., and Tofte, M., editors, *[Ehrig, H. et al., 1999b]*, chapter 4. MIT Press, 1999. http://www.di.unipi.it/~ugo/festschrift.ps.

[Garlan, D. and Kompanek, A., 2000] Garlan, D. and Kompanek, A. Reconciling the needs of architectural description with object- modeling notations. In *Third International Conference on the Unified Modeling language (UML'2000)*, October 2000.

[Garlan, D. and Shaw, M., 1996] Garlan, D. and Shaw, M. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[Heckel R., 1998] Heckel R. *Open Graph Transformation Systems: A new Approach to the Compositional Modelling of Concurrent and Reactive Systems*. PhD thesis, Universitt Berlin, 1998.

[Hirsch, D. and Montanari, U., 1999] Hirsch, D. and Montanari, U. Consistent transformations for software architecture styles of distributed systems. In G. Stefanescu, editor, *Workshop on (formal methods applied to) Distributed Systems*, volume 28, pages 23–40, Iasi, Rumania, September 1999. Electronic Notes in Theoretical Computer Science. http://www.elsevier.nl/locate/entcs/volume28.html.

[Hirsch, D. and Montanari, U., 2000] Hirsch, D. and Montanari, U. Higher-order hyperedge replacement systems and their transformations: Specifying software architecture reconfigurations. In Ehrig, H. and Taentzer, G., editors, *Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems (GRATRA 2000). Satellite Event of ETAPS 2000*, Technical Report of Computer Science Department/TU Berlin, No. 2000-02, pages 215–223, March 2000.

[Hirsch, D. and Montanari, U., 2001a] Hirsch, D. and Montanari, U. A graphical calculus for name mobility. In *Workshop on Software Engineering and Mobility (satellite workshop of ICSE2001)*, Toronto, Canada, 2001. http://www.elet.polimi.it/Users/DEI/Sections/Compeng/GianPietro.Picco/ICSE01mobility/.

[Hirsch, D. and Montanari, U., 2001b] Hirsch, D. and Montanari, U. Synchronized hyperedge replacement with name mobility: A graphical calculus for name mobility. In *12th International Conference in Concurrency Theory (CONCUR 2001)*, volume 2154 of *Lecture Notes in Computer Science*, pages 121–136, Aalborg, Denmark, 2001. Springer-Verlag.

[Hirsch, D. and Montanari, U., 2001c] Hirsch, D. and Montanari, U. Synchronized hyperedge replacement with name mobility. Technical Report TR-96-27, Department of Computer Science, Universidad de Buenos Aires, 2001. http://www.dc.uba.ar/people/proyinv/tr.html.

[Hirsch, D. *et al.*, 1998] Hirsch, D., Inverardi, P., and Montanari, U. Graph grammars and constraint solving for software architecture styles. In *Third International Software Architecture Workshop*, volume 1906 of *Lecture Notes in Computer Science*, Orlando, Miami, 1998. Springer-Verlag.

[Hirsch, D. *et al.*, 1999] Hirsch, D., Inverardi, P., and Montanari, U. Modeling software architectures and styles with graph grammars and constraint solving. In *First Working IFIP Conference on Software Architecture*, San Antonio, Texas, February 1999. Kluwer Academic Publishers.

[Hirsch, D. *et al.*, 2000] Hirsch, D., Inverardi, P., and Montanari, U. Reconfiguration of software architecture styles with name mobility. In *4th International Conference, Coordination 2000*, volume 1906 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[Hofmeister, C. *et al.*, 1999a] Hofmeister, C., Nord, R., and Soni, D. *Applied Software Architecture*. Addison-Wesley, 1999.

[Hofmeister, C. *et al.*, 1999b] Hofmeister, C., Nord, R., and Soni, D. Describing software architecture with uml. In *First Working IFIP Conference on Software Architecture*, San Antonio, Texas, February 1999. Kluwer Academic Publishers.

[Inverardi, P. and Muccini, H., 2000] Inverardi, P. and Muccini, H. The teleservices and remote medical care system (trmcs). workshop case study. In *10th International Workshop on Software Specification and Design (IWSSD-10)*, San Diego, California, November 2000. http://www.ics.uci.edu/IRUS/iwssd/cfp.html.

[Inverardi, P. and Wolf, A. L., 1995] Inverardi, P. and Wolf, A. L. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, pages 373–386, April 1995.

[König, B. and Montanari, U., 2001] König, B. and Montanari, U. Observational equivalence for synchronized graph rewriting with mobility. *Submitted*, 2001.

[Kramer, J. and Magee, J., 1990] Kramer, J. and Magee, J. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[Kruchten, P.B., 1995] Kruchten, P.B. The 4+1 view model of architecture. *IEEE Software*, pages 42–50, November 1995.

[Lanese, I. and Montanari, U., 2002] Lanese, I. and Montanari, U. Software architectures, global computing and graph transformation via logic programming. In Leila Ribeiro, editor, *SBES'2002 - 16th Brazilian Symposium on Software Engineering*, pages 11–35, October 2002.

[Le Métayer, D., 1998] Le Métayer, D. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7), July 1998.

[Lukham, D. *et al.*, 1995] Lukham, D., Kenney, J., Augustin, L., Vera, J., Bryan, D., and Mann, W. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4), 1995.

[Mackworth, A.K., 1998] Mackworth, A.K. Constraint satisfaction. In *Encyclopedia of IA*. Springer-Verlag, 1998.

[Magee, J. and Kramer, J., 1996a] Magee, J. and Kramer, J. Dynamic structure in software architectures. In *Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE4)*, ACM Software Engineering Notes, pages 3–14, San Francisco, October 1996.

[Magee, J. and Kramer, J., 1996b] Magee, J. and Kramer, J. Self organising software architectures. In *Second International Software Architecture Workshop*, 1996.

[Magee, J. *et al.*, 1995] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. Specifying distributed software architectures. In *Fifth European Software Engineering Conference (ESEC95)*, Barcelona, September 1995.

[Marx, M. *et al.*, 1996] Marx, M., Pólos L., and Masuch, M., editors. *Arrow Logic and Multi-Modal Logic*. Cambridge University Press, 1996.

[Medvidovic, N. and Rosemblum, D., 1999] Medvidovic, N. and Rosemblum, D. Assesing the suitability of a standard design method. In *First Working IFIP Conference on Software Architecture*, San Antonio, Texas, February 1999. Kluwer Academic Publishers.

[Medvidovic, N. and Taylor, R., 1997] Medvidovic, N. and Taylor, R. A framework for classifying and comparing architecture description languages. In *6th European Software Engineering Conference (ESEC'97)*, 1997.

[Medvidovic, N. *et al.*, 1996] Medvidovic, N., Oreizy, P., Robbins, J., and Taylor, R. Using object-oriented typing to support architectural design in the c2 style. In *SIGSOFT'96: Proceedings of the 4th ACM Symposium on the Foundations of Software Engineering*. ACM Press, October 1996.

[Mens, T., 2000] Mens, T. Conditional graph rewriting as a domain-independent formalism for software evolution. In *International Workshop on Applications of Graph Transformation with Industrial Relevance*, Lecture Notes in Computer Science. Springer-Verlag, 2000.

[Meseguer, J., 1992] Meseguer, J. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[Milner, R., 1999] Milner, R. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, 1999.

[Mitchell, J., 1996] Mitchell, J. *Foundations for Programming Languages*. MIT Press, 1996.

[Montanari, U. and Rossi, F., 1999] Montanari, U. and Rossi, F. Graph rewriting, constraint solving and tiles for coordinating distributed systems. *Applied Categorical Structures*, 7:333–370, 1999.

[Montanari, U. *et al.*, 1999] Montanari, U., Pistore, M., and Rossi, F. Modeling concurrent, mobile and coordinated systems via graph transformations. In *[Ehrig, H. et al., 1999b]*, chapter 4. World Scientific, 1999.

[Moriconi, M. *et al.*, 1995] Moriconi, M., Qian, X., and Riemenschneider, R. Correct architecture refinement. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):356–372, April 1995.

[Périn, M., 2000] Périn, M. *Spécifications Graphiques Multi-Vues: Formalisation et Vérification de Cohérence*. PhD thesis, Université de Rennes, 2000.

[Perry, D. and Wolf, A., 1992] Perry, D. and Wolf, A. Foundations for the study of software architecture. *ACM SIGSOFT*, 17(4):40–52, 1992.

[Rozenberg, 1997] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Foundations*, volume 1. World Scientific, 1997.

[Sangiorgi, D. and Walker, D., 2001] Sangiorgi, D. and Walker, D. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[Sangiorgi, D., 1996] Sangiorgi, D. π-calculus, internal mobility and agent-passing calculi. *Theoretical Computer Science*, 167(2), 1996.

[Taentzer, G. *et al.*, 1998] Taentzer, G., Goedicke, M., and Meyer, T. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In *6th Int. Workshop on Theory and Application of Graph Transformation*, 1998.

[Victor, B., 1998] Victor, B. *The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes*. PhD thesis, Dept. of Computer Science, Uppsala University, 1998.

[Wermelinger, M. and Fiadeiro, J., 2002] Wermelinger, M. and Fiadeiro, J. A graph transformation approach to software architecture reconfiguration. *Science of Computer Programming*, 44:133–155, 2002.

[Wermelinger, M., 1999] Wermelinger, M. *Specification of Software Architecture Reconfiguration.* PhD thesis, Universidade Nova de Lisboa, September 1999. http://ctp.di.fct.unl.pt/ mw/pubs/1999/phd.html.