UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

LUCIANA FOSS

Transactional Graph Transformation Systems

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Profa. Dr. Leila Ribeiro
Advisor

Prof. Dr. Andrea Corradini
Coadvisor

Porto Alegre, July 2008

CIP – CATALOGING-IN-PUBLICATION

Foss, Luciana

Transactional Graph Transformation Systems / Luciana Foss. – Porto Alegre: PPGC da UFRGS, 2008.

123 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2008. Advisor: Leila Ribeiro; Coadvisor: Andrea Corradini.

1. Graph transformation. 2. Transactions. 3. Refinement. 4. Interaction pattern. I. Ribeiro, Leila. II. Corradini, Andrea. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. José Carlos Ferraz Hennemann
Vice-Reitor: Prof. Pedro Cezar Dutra Fonseca
Pró-Reitora de Pós-Graduação: Profª. Valquiria Linck Bassani
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenadora do PPGC: Profª. Luciana Porcher Nedel
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

To my mother Celina, my
brother André and my niece Julia.
In the memory of my father, Luiz Foss.

TABLE OF CONTENTS

LIST OF FIGURES 6

LIST OF TABLES 8

ABSTRACT 9

RESUMO 10

1 INTRODUCTION 11
1.1 Outline 13
1.2 PhD Thesis Motivation and Definition 13
1.2.1 Goals of Thesis 14

2 TYPED GRAPH TRANSFORMATION SYSTEMS 15
2.1 Operational Semantics 18
2.2 Category GTS 23

3 TRANSACTIONAL GRAPH TRANSFORMATION SYSTEMS 27
3.1 Introduction to Transactional GTS 27
3.2 Category TGTS 33
3.3 Abstract GTS associated to a T-GTS 38
3.4 Transactions as graph processes 39
3.4.1 Graph Processes 40
3.4.2 Transactional Processes 44
3.4.3 Abstract GTS for a T-GTS based on process 45
3.5 Implementation morphisms for T-GTSs 47
3.6 Adjunction between GTS and TGTS^{imp} 63

4 TRANSACTIONAL GRAPH TRANSFORMATION SYSTEMS WITH DEPENDENCY RELATION 66
4.1 Categories dGTS and dTGTS 70
4.2 Abstract d-GTS for a T-GTS with dependency relation 72
4.3 Implementation morphism for dT-GTS 77
4.4 Adjunction between dGTS and dTGTS^{imp} 79
4.5 Comparing T-GTS with dT-GTS 81
4.6 Construction of the abstract system associated to a dT-GTS 85
4.7 Refinement of transactional graph transformation systems 87

5 CONCLUSIONS 91
5.1 Theoretical contribution 91
5.2 Software Engineering contribution 92
5.3 Future work 96

REFERENCES 98

APPENDIX A CATEGORICAL DEFINITIONS 105

APPENDIX B PROPER QUOTIENT PRODUCTIONS 108

APPENDIX C TRANSACTIONS OF A DT-GTS 109

APPENDIX D RESUMO ESTENDIDO DA TESE 119
D.1 Contribuições 121
D.1.1 Contribuições para a área teórica 121
D.1.2 Contribuições para a área de Engenharia de Software 122
D.1.3 Trabalhos futuros 122

LIST OF FIGURES

Figure 2.1: Typed graph G. 16
Figure 2.2: A graph production. 17
Figure 2.3: Example of GTS: a pump operator of a gas station system. 18
Figure 2.4: Direct derivation from G to H using ACCEPT based on m. 19
Figure 2.5: Two sequentially independent direct derivations. 20
Figure 2.6: Sequential independence. 20
Figure 2.7: Parallel production. 21
Figure 2.8: Parallel production $q_1 + q_2$ and a proper quotient production q of $q_1 + q_2$. 21
Figure 2.9: Derivation shift-equivalent to derivation in Figure 2.5. 22
Figure 2.10: Isomorphism of derivations. 23
Figure 2.11: Partial morphism f_T. 24
Figure 2.12: Translation of STOP production with respect to f_T: STOP production (left), pulling back along l_{f_T} (center) and STOP' typed over T_1 (right). 24
Figure 2.13: Productions of $\mathcal{G}_1$. 26

Figure 3.1: Transactional GTS $\mathcal{P}umpOper$ for a pump operator of a gas station. . 28
Figure 3.2: Transactional GTS $\mathcal{C}ustomer$ for a customer of a gas station. 29
Figure 3.3: Stabilised graph. 30
Figure 3.4: A transaction of the T-GTS $\mathcal{P}umpOper$ in Example 3.1. 32
Figure 3.5: GTS morphism f does not define a T-GTS morphism. 34
Figure 3.6: Derivation $\mathcal{S}(\rho)$ and its equivalent derivation via proper quotient production. 36
Figure 3.7: Derivations $f_T^{\leftrightarrow}(\rho)$ and $f_T^{\leftrightarrow}(\delta)$. 37
Figure 3.8: Abstract production associated to the transaction in Figure 3.4. 39
Figure 3.9: Type graph (top-left), the maximal and the minimal graphs (bottom-left), and productions (right) of the process associated to derivation in Figure 3.4. 42
Figure 3.10: Isomorphism of processes. 43
Figure 3.11: Class of shift-equivalent derivations (left) and the equivalent graph process (right) 44
Figure 3.12: Abstract GTS associated to the T-GTS $\mathcal{P}umpOper$, depicted in Figure 3.1. 46
Figure 3.13: Transactional process ϕ implements production p. 47
Figure 3.14: Implementation morphism from the abstract GTS $A(\mathcal{C}ustomer)$ to the T-GTS $\mathcal{C}ustomer$. 48
Figure 3.15: Transactional processes of customer system. 48
Figure 3.16: Composition of implementation T-GTS morphisms is associative. . . . 55

Figure 3.17: Universality of $\epsilon_{\mathcal{Z}}$ in $\mathbf{TGTS}^{imp}$. 64

Figure 4.1: Transactions implementing STOP production depicted in Figure 3.1. . 67
Figure 4.2: A dep-production. 68
Figure 4.3: A T-GTS $\mathcal{D}epPumpOper$ with dependency relation for gas station system. 70
Figure 4.4: dep-productions p_1 and p_2. 72
Figure 4.5: Transactional process of dT-GTS $\mathcal{D}epPumpOper$. 73
Figure 4.6: dep-production associated to the process in Example 4.4. 76
Figure 4.7: Abstract d-GTS $\mathcal{Z}_1$ associated to the dT-GTS $\mathcal{D}epPumpOper$, described in Example 4.2. 76
Figure 4.8: Universality of $\epsilon_{\mathcal{Z}}$ in $\mathbf{dTGTS}^{imp}$. 80
Figure 4.9: Transaction constructed from dep-production STOP in Figure 4.2. . . 83
Figure 4.10: The dT-GTS $\mathcal{Z}_2$. 84
Figure 4.11: Productions ACCEPT and FINISH for the dT-GTS $\mathcal{Z}_2'$. 84
Figure 4.12: A refinement dT-GTS $\mathcal{Z}$ for $\mathcal{D}epPumpOper$ (Figure 4.3). 90
Figure 4.13: An unstable transactional process associated to STOP production. . . 90

LIST OF TABLES

Table 4.1: Dependency relations of productions of transactional process ϕ_1. 74
Table 4.2: Transitive closure of dependency relations of production of ϕ_1. 75
Table 4.3: Dependency relation associated to transactional process ϕ_1. 75
Table 4.4: Transitive closure of dependencies of productions of ϕ depicted in Figure 4.9. 82

ABSTRACT

Reactive systems, in contrast to transformational systems, are characterised by having to continuously react to stimuli from its environment. If, in addition to reactiveness, we consider that for many applications the specification method should provide a way to describe the spatial distribution of states, graph transformation seems to be a suitable specification technique. Some applications with these characteristics are mobile systems and biological pathways. However, the approaches provided for graph transformations so far are not adequate to explicitly describe interaction patterns.

Furthermore, several approaches to specify reactive systems propose to use asynchronous languages to specify communication between components and define mechanisms to describe a set (or sequence) of activities that are performed atomically. However, scarce attention has been devoted to the idea of extending GTSs in order to allow the specification of atomic activities.

Inspired by the ideas of zero-safe Petri nets, an extension of graph transformation systems (GTSs) – called *transactional* GTS (T-GTS) – was defined, equipping them with a transaction notion. A transaction, in this approach, describes a set of actions that are executed in an atomic way and it is defined by distinguishing the resources that are visible or invisible from an external point of view, where the last ones are considered temporary and are forgotten at a more abstract level.

In this thesis, we give a more theoretical foundation to T-GTS defining a notion of implementation morphisms between T-GTSs (associating graph productions of a system with transactions of other system) and using this notion we demonstrate the existence of an adjunction between categories of GTSs and T-GTSs with implementation morphisms. Moreover, we extends transactional GTSs with a mechanism to describe interaction patterns of reactive systems, by means of dependency relations included in the graph productions. The idea is that a system interacts with its environment by consuming and creating elements visible to this environment, obeying a causal dependency. Finally, we propose a notion of glass-box refinement for T-GTSs with dependency relations, where some internal aspects are preserved. In an abstract level, the system is specified by productions describing (in an atomic way) complete reactions, where the dependency relations give some constraints on the internal structure of these reactions. A refinement of a system is given by a total implementation morphism, that associates each (abstract) production to a transaction. Hence, the refined system preserves all external behaviour of the original system and the internal constraints given by the dependency relations.

Keywords: Graph transformation, transactions, refinement, interaction pattern, Graph transformation, transactions, refinement, interaction pattern.

Sistemas de Transformação de Grafos Transacionais

RESUMO

Em contraste aos sistemas transformacionais, sistemas reativos são caracterisados por reagir continuamente a estímulos provinientes seu ambiente. Além da reatividade, se considerarmos que muitas aplicações requerem métodos de especificação que possibilitam descrever a distribuição espacial dos estados, sistemas de transformação de grafos parecem ser uma técnica de especificação bastante adequada. Algumas aplicações com essas características são sistemas móveis e vias biológicas.

Além disso, diversas abordagens para especificação de sistemas reativos propõem usar linguagens assíncronas para especificar a comunicação entre componentes e definem mecanismos para descrever um conjunto (ou seqüência) de atividades que são realizadas atomicamente. Porém, pouca atenção tem sido dada à idéia de estender sistemas de transformação de grafos para permitir a especificação de atividades atômicas.

Recentemente, inspirada nas idéias das redes de Petri "zero-safe" foi definida uma extensão de sistemas de transformação de grafos (GTS) – denominada GTS *transacional* (T-GTS) – equipando-os com uma noção de transação. Uma transação, nesta abordagem, descreve um conjunto de ações que são executadas de um modo atômico e é definida através de uma distinção entre os recursos visíveis e invisíveis de um ponto de vista externo, onde os últimos são considerados temporários e "esquecidos" em um nível abstrato.

Nesta tese é dada uma fundamentação mais teórica para T-GTSs definindo uma noção de morfismos de implementação T-GTS (associando produções de um sistema com transações de outro) e, usando essa noção, é demonstrada a existência de uma adjunção entre as categorias de GTSs e T-GTSs com morfismos de implementação. Além disso, GTSs transacionais são estendidas com um mecanismo para descrever padrões de interação de sistemas reativos através de relações de dependência incluídas nas produções. A idéia é que um sitema interage com seu ambiente consumindo e criando elementos visíveis para à esse ambiente, uma relação de causalidade. Finalmente, propomos uma noção de refinamento para T-GTSs com relação de dependência caracterizada por uma visão "caixa-de-vidro", onde alguns aspectos internos são preservados. Em um nível abstrato, o sistema é especificado por produções que descrevem (de uma maneira atômica) reações completas, onde a relação de dependência determina algumas restrições na estrutura interna dessas reações. Um refinamento de um sistema é definido por um morfismo total de implementação que associa cada produção (abstrata) a uma transação. Assim, o sistema refinado preserva todo o comportamento externo do sistema original e as restrições da estrutura interna determinadas pelas relações de dependência.

Palavras-chave: Transformação de grafos, transações, refinamento, padrão de interação.

1 INTRODUCTION

The complexity of today systems requires the use of development methods that guarantee correctness and quality. Formal specification is an important instrument used to achieve these goals. A specification is a description of the behaviour of a system and/or its structure.

Graph transformation systems (GTSs) are a flexible formalism for the specification of complex systems, that may take into account aspects such as object-orientation, concurrency, mobility and distribution (EHRIG et al., 1999a). In fact, graphs can be naturally used to provide a structured representation of the states of a system, which highlights its subcomponents and their logical or physical interconnections. Then, the events occurring in the system, which are responsible for the evolution from one state into another, are modelled by applications of suitable transformation rules, called (graph) productions. Such a representation is precise enough to allow the formal analysis of the system under scrutiny, as well as amenable of an intuitive, visual representation, which can be easily understood also by a non-expert audience.

Along the years several enrichments of the original framework have been introduced, extending GTSs with structuring concepts that are needed to master the complexity of large specifications. Several modularity and refinement notions have been proposed, providing basic mechanisms for encapsulation, abstraction and information hiding – see (HECKEL et al., 1999; SCHÜRR; WINTER, 2000; KREOWSKI; KUSKE, 1999), (DREWES et al., 2000; HECKEL et al., 1998; TAENTZER; SCHÜRR, 1995; GROSSE-RHODE; PARISI-PRESICCE; SIMEONI, 1999; EHRIG; ENGELS, 1993, 1996). However, to our knowledge, scarce attention has been devoted to the idea of extending GTSs in order to allow the specification of transactional activities. Abstractly, a transaction is an activity, involving the execution of a group of events, which can either bring the system to a successful state or fail. In the last case the partial execution of the transaction is discarded and has no effect on the system. In concrete implementations this is achieved with a roll-back mechanism which restores the starting state when a failure is detected.

In an introductory work (BALDAN et al., 2008), inspired by the ideas of zero-safe Petri nets, we define an extension of GTSs, called *transactional graph transformation systems* (T-GTS), equipping them with a transaction notion. Roughly speaking, states (graphs) are partitioned into a stable and a non-stable (unstable) parts, and a transaction is defined as a computation that starts and ends in states consisting only of stable items, in which all intermediate states have some unstable parts. This approach is motivated by the understanding of graph transformation as a *data-flow* formalism, where the productions of a system are applied non-deterministically, and any form of control on the application of productions has to be encoded in the graphs. Thus, transactions are more naturally defined indirectly, by identifying parts of the state which represent temporary (or "unstable")

resources, only visible within a transaction. Transactions can be seen in two different levels of abstraction. In a lower level, both stable and unstable items, and thus also the internal structure of transaction, are visible. But at a more abstract level, the unstable items can be forgotten and only the complete transactions are observable. Intuitively, this gives rise to another GTS, where abstract transactions of the original T-GTS become productions which rewrite directly the source stable state into the target stable state.

Reactive systems, in contrast to transformational systems, are characterised by continuous having to react to stimuli from its environment. If, in addition to reactiveness, we consider that for many applications the specification method should provide a way to describe the spatial distribution of states, graph transformation seems to be a suitable specification technique. Some applications with these characteristics are mobile systems and biological pathways.

Several methods for design and analysis of reactive systems propose synchronous languages as specification formalism (BERRY, 2000; HALBWACHS et al., 1991; LEGUERNIC et al., 1991), where the time of reaction to an event is null. Other methods (SECELEANU; SECELEANU, 2004; MAIA; IORIO; BIGONHA, 1998; RIESCO; TUYA, 2004) propose to use asynchronous languages to specify communication between components and define mechanisms to describe a set (or sequence) of activities that are performed atomically. Therefore, we can use the notion of transactions to describe, at an abstract level, these atomic activities. Moreover, in (MAIA; IORIO; BIGONHA, 1998) an approach to specify interaction pattern explicitly using Abstract State Machines (ASM)s was proposed, where the designer, besides specifies the operations of component, can describe signals that are sent to and received from environment. The designer can also partially specify the environment showing only the interaction specification.

However, the approaches for graph transformations do not provide a mechanism to explicitly specify patterns of interaction between a system and its environment. Some of them restrict the interaction pattern to functions (and thus actually describe transformational systems), and others just allow very restricted forms of interactions. In (HECKEL, 1998), it is proposed to use graph transformation systems to specify reactive systems: the interaction between system and its environment it not explicitly specified, instead, it is described in the semantical level, where the states of the system show effects that are not determined by rule applications. Therefore, we extend the transactional graph transformations systems to explicitly express interaction.

In this thesis, we firstly develop a more elaborated work on transactional GTS: we define a more manageable characterisation of transactions as graph processes, which simplifies categorical definitions; based on the characterisation of transactions as processes, we show how the internal structure of transactions can be abstracted away, by considering an abstract GTS associated to a T-GTS; and, we demonstrate that the concrete T-GTS and its associated abstract GTS have the same behaviour in terms of transactions by characterising this construction as a functor from the category of T-GTSs to the category of GTS and showing that it is the right adjoint to the inclusion functor in the opposite direction. In order to do this, we define appropriated morphisms in both categories, specially a notion of implementation morphism for the category of T-GTSs. Using the notion of implementation morphism it is possible to describe the behaviour of a system at an abstract level and assure that the refined system really executes according to this abstract behaviour. Preliminary results of this work was published in (BALDAN et al., 2006).

Next, we extend T-GTS to a version of T-GTS with dependency relations (dT-GTS), by providing a mechanism to describe a weaker dependency among created and consumed

items of a production. We extend, too, all concepts and results of the original version to dT-GTSs. The notion of dependency is extended to transactions and it reflects the dependencies generated by internal items of the transaction. Moreover, this relation restricts the definition of implementation morphisms giving additional constraints to possible refinements for each production. Preliminary results of this work was published in (FOSS; MACHADO; RIBEIRO, 2007). Finally, we propose to use dT-GTS to specify reactive systems. The idea, in our proposal, is that a component interacts with its environment by consuming and creating elements visible to this environment. These actions may be described as (abstract) graph productions in the abstract specification that are implemented by a series of other productions in the more concrete specification. Defining a relationship between these different levels of abstraction we can have a notion of refinement. Additionally, the dependency relation associated to each abstract production can describe a complex pattern of interaction between a system and its environment.

1.1 Outline

In Chapter 2, some concepts of double-pushout approach for typed graph transformation systems (GTS) will be presented. An extension of this formalism, called transactional graph transformation systems (T-GTS), will be presented in Chapter 3. We review all definitions of transactions and abstract system associated to a T-GTS. In this chapter, we also introduce a characterisation of transactions as graph processes that is used to define implementation morphisms and to demonstrate the existence of an adjunction between categories of GTS and T-GTS with implementation morphism. A further extension of T-GTSs is given in Chapter 4, where all definitions and results obtained in Chapter 3 are reproduced for T-GTSs with dependency relations. In the last section of this chapter, we propose a notion of refinement for specification of reactive systems. And, in the last chapter (5), we summarise the contributions of this work, give an overview about related work and describe the future directions of this research. In the appendices, some of the formal definitions and theorems used/proved along of this proposal can be found.

1.2 PhD Thesis Motivation and Definition

This research is inserted in a cooperation project between Brazil and Italy, called IQ-Mobile, which has as main goal to improve the software quality for open environments, specially mobile and distributed applications, using formal methods. Within the scope of this project, some aspects of Java language and graph transformation systems were compared: the object model; the support and treatment of concurrency; systems composed of several classes. After analysing these aspects some translations between Java and GTS were considered. The interest in these translations has two aspects: on the one hand, the possibility of to map GTSs in a programming language; and on the other hand, if Java constructions can be translated in GTS, one can analyse Java systems by means of formal techniques. Therefore, some work was developed in this sense: GTSs were mapped into Java classes (DOTTI et al., 2005) and vice-versa (CORRADINI et al., 2004). Some questions arose from the last translation:

- in this mapping we had to consider several details of Java implementation, thus the concept of transaction became an important mechanism to allow the abstraction of these details;

- the Java language, with its class concepts, allows a modular development of large systems. An interest in leading these modularity concepts into GTS arose as a result of comparison between the languages.

Thus, as a first step, a transaction notion was defined (BALDAN et al., 2008), where a transaction is a computation with some special properties. Moreover, a transaction can be seen at an abstract level, where some intermediate steps can be forgotten. This introductory work presented some basic definitions without further formal foundations. It would be nice to express the abstraction construction as a functor from the category of transactional GTSs to the category of GTSs, giving raise, as in the case of Petri nets, to the right adjoint to the inclusion functor in the opposite direction. This allows us to relate the abstract and concrete systems by means of morphisms and prove that they are equivalent from an external observer point of view. For this, it seems that the appropriate choice of morphisms in the category of T-GTSs must allow to map a production into a transaction. Therefore, the following step in this research was to give a categorical foundations for T-GTSs.

Other researchers proposed, in the scope of IQ-Mobile project, an approach to verify partial systems using restricted GTSs, called object-based graph grammars (DOTTI et al., 2006). This work was based on the assume-guarantee approach for verification (PNUELI, 1985). The basic idea is to see each part of a system as an open system - a system whose behaviour is not fully specified and that depends on interactions with its environment. In order to verify a component alone, one can assume an environment behaviour and, based on this assumption, try to prove the desired properties of the component. In their approach, they describe several steps to allow this verification and the first two are: (1) define the interface and (2) define interaction pattern (assume). In the second step, a description of the desired interaction pattern between the system and its environment is required. Inspired in (DOTTI et al., 2006), we, finally, define an extension of T-GTSs incorporating a notion of dependency relation and propose it to specify reactive systems.

1.2.1 Goals of Thesis

The main goals of this thesis are:

- to define a notion of atomic activity for GTSs;
- to demonstrate that atomic activities are preserved in a higher level of abstraction;
- to propose a mechanism to describe interaction pattern by means of graph productions to specify reactive systems;
- to define a notion of refinement which take in account the interaction pattern.

2 TYPED GRAPH TRANSFORMATION SYSTEMS

In this chapter, the basic definitions of typed graph transformation systems in the algebraic approach are reviewed. Typing discipline for graphs will allow us to distinguish between stable and unstable items in a given graph. Typing for graphs (see (HECKEL et al., 1996; CORRADINI; MONTANARI; ROSSI, 1996) for more details) can be seen as a labelling technique, which allows to label each graph over a structure that is itself a graph (called the *type graph*). The base of this approach is the category of graphs and graph morphisms.

(Typed) Graph and graph morphisms. A *graph* is composed by vertices and edges connecting source and target vertices. We can relate two graphs if they are structurally compatible. This relation is given by means of a *graph morphism*: a mapping between graphs that respects the source and the target of each edge, i.e., each edge mapped into another must have its source and target vertices mapped into the source and the target vertices of the other, respectively. Besides, we can label each item of a graph mapping it into other graph, called type graph. A *typed graph* is a graph equipped with a *typing morphism* from it into a type graph. The compatibility between graphs typed over the same type graph is determined by *typed graph morphisms*, that are mappings respecting the type of each item of the graph, i.e., the vertices and edges of a graph can be mapped only into vertices and edges of another if they have the same type. If the typing mapping of a graph is injective, the graph is called *injective*, as well.

Definition 2.1 ((Typed) Graph and graph morphisms) *A* graph *is a tuple $G = \langle V_G, E_G, s^G, t^G\rangle$, where V_G and E_G are sets of vertices and edges, and $s^G, t^G : E_G \to V_G$ are the source and target function. A* (total) graph morphism *$f : G \to G'$ is a pair of functions $(f_V : V_G \to V_{G'}, f_E : E_G \to E_{G'})$ such that $f_V \circ s^G = s^{G'} \circ f_E$ and $f_V \circ t^G = t^{G'} \circ f_E$. The category of graph and total graph morphisms is called* $\mathbf{Graph}$.
Let $T \in \mathbf{Graph}$ be a fixed graph, called type graph, a T-typed graph *G^T is given by a graph G and a graph morphism $t_G : G \to T$. When type graph is obvious we will write G instead of G_T. A morphism of T-typed graphs $f : G^T \to G'^T$ is a graph morphism $f : G \to G'$ that satisfies $t_{G'} \circ f = t_G$. A typed graph G^T is called* injective *if the typing morphism t_G is injective. The category of T-typed graphs and T-typed graph morphisms is the comma category $(\mathbf{Graph} \downarrow T)$, shorted by T-$\mathbf{Graph}$.*

Example 2.1 ((Typed) Graph and graph morphisms) *Figure 2.1 shows two graphs T and G and a morphism f between them. The vertices are represented by round squares and circles, for example* Operator *and* Free*; and the edges are represented by arrows with names (inscribed in polygons or not), for example,* Busy *and* Start*. When source and target of an edge are the same vertex, we omit the source in the pictures, for example,*

the edge Busy has the pump vertex as source and target. Morphism f maps the vertices and edges of G into vertices and edges of T, respectively. This mapping is defined by the numbering. Graph G and morphism f compose a typed graph $G^T = \langle G, f \rangle$.

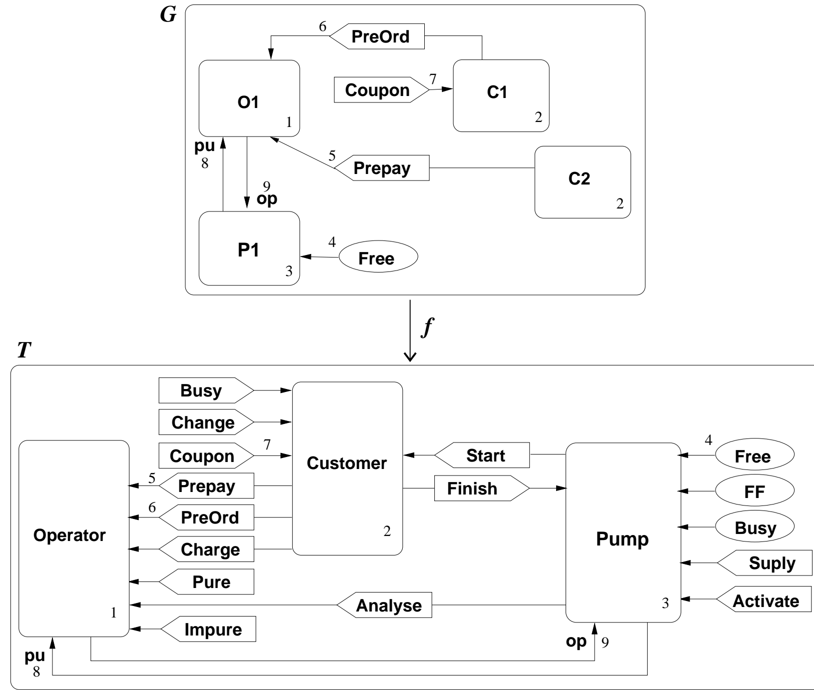


Figure 2.1: Typed graph G.

⌟

The behaviour of a graph transformation system is determined by the application of rewriting rules, also called graph productions (ROZENBERG, 1997).

Productions. At an abstract level, a *production* is composed by three graphs: the *left-hand side* L, the *right-hand side* R, and an *interface* K which represents the parts that L and R have in common. It specifies that, once an occurrence of the graph L is found in the current state[1], it can be replaced with the graph R, preserving K.

Definition 2.2 (Productions) *A T-typed (graph) production is a tuple $q : L_q \overset{l_q}{\hookleftarrow} K_q \overset{r_q}{\rightarrowtail} R_q$, where q is the name of the production, L_q, K_q and R_q are T-typed graph, l_q is an inclusion and r_q is an injective morphism. The class of all T-typed graph production is denoted by T-$\mathbf{Prod}$.*

In this work we will only consider consuming productions, i.e., productions must consume something.

Example 2.2 (Productions) *Figure 2.2 shows a production q whose left-hand side, right-hand side and interface are graphs L_q, R_q and K_q, respectively. Graph K_q is mapped into L_q and R_q by two morphisms: l and r. This production can be applied in a state containing graph L_q (i.e. containing vertices typed as Customer and Operator, and edge typed as `Prepay` from the Customer to the Operator), resulting in a graph where `Prepay` was deleted; Customer and Operator were preserved; and `Coupon` and `PreOrd` were created.*

[1] The graph representing the current state of the system.

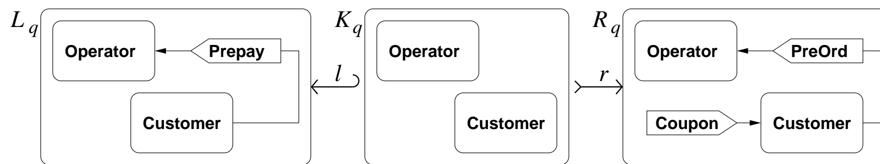


Figure 2.2: A graph production.

┘

Graph transformation systems. A *graph transformation system* is defined by a collection of graph productions typed over a fixed type graph. Its definition is given by a type graph, a set of production names, and a function associating each production name to a typed production.

Definition 2.3 (Graph transformation systems - GTS) *A* (typed) graph transformation system *is a tuple $\mathcal{G} = \langle T, P, \pi \rangle$, where T is a type graph, P is a set of production names, π is a function mapping production names to productions in T-$\mathbf{Prod}$.*

Example 2.3 (Graph transformation systems) *An example will be used to illustrate the main presented concepts. We will model an adapted version of a gas station system presented in (DOTTI* et al.*, 2006) using* GTS*s. In this example, a customer prepays a certain amount of money to the operator of a gas station for the gas that will be supplied by the pump. If the operator is free, the pump supplies the gas and then the operator returns the change to the customer, based on the real amount of money expended by the customer. Otherwise, the customer receives a message advising him to retry. Another activity realised in the gas station is the analysis of the pump in order to test the purity or impurity of its gas.*

In Figure 2.3, the GTS *modelling the system of a pump operator of a gas station system is shown. The type graph of the system is depicted at the bottom of the figure. We have, in this system, three entities modelled by vertices: Customer, Operator, and Pump. Each entity can have attributes (for example FF or $Free$) and receive messages (for example `Prepay` or `Supply`). Each entity can have a reference to other one, that represents an attribute as well. Both, attributes and messages are modelled as edges. So, by type graph we can see the Pump entity with their attributes: the operator (op) that operates it and three flags indicating if it is free ($Free$), or activated (FF), or busy ($Busy$). The Customer entity has only a reference to the operator (op) as attribute; and the Operator has as attributes the pump that it operates (pu).*

*The behaviour of this system is described by the productions at the top of Figure 2.3 – the interface graphs of the productions were omitted but they can be built as the intersection between the left-hand side and the right-hand side of each production. The Operator entity controls the access to the Pump when a customer tries to use it. If pump is not being used, the operator accepts (generating a `PreOrd` message) the prepayment and gives to customer a coupon (*ACCEPT *production) and it prepares the pump to be activated (*SERVE *production). The Pump maybe activated, changing its flag from $Free$ to FF (*ACTIVATE *production) and it may start supplying gas changing its flag from FF to $Busy$ (*START *production). Eventually, the Pump finishes supplying the gas – changing its flag from $Busy$ to $Free$ again – and it indicates to the operator (generating `Charge` message) the real amount that the customer expended (*STOP *production). Finally, the*

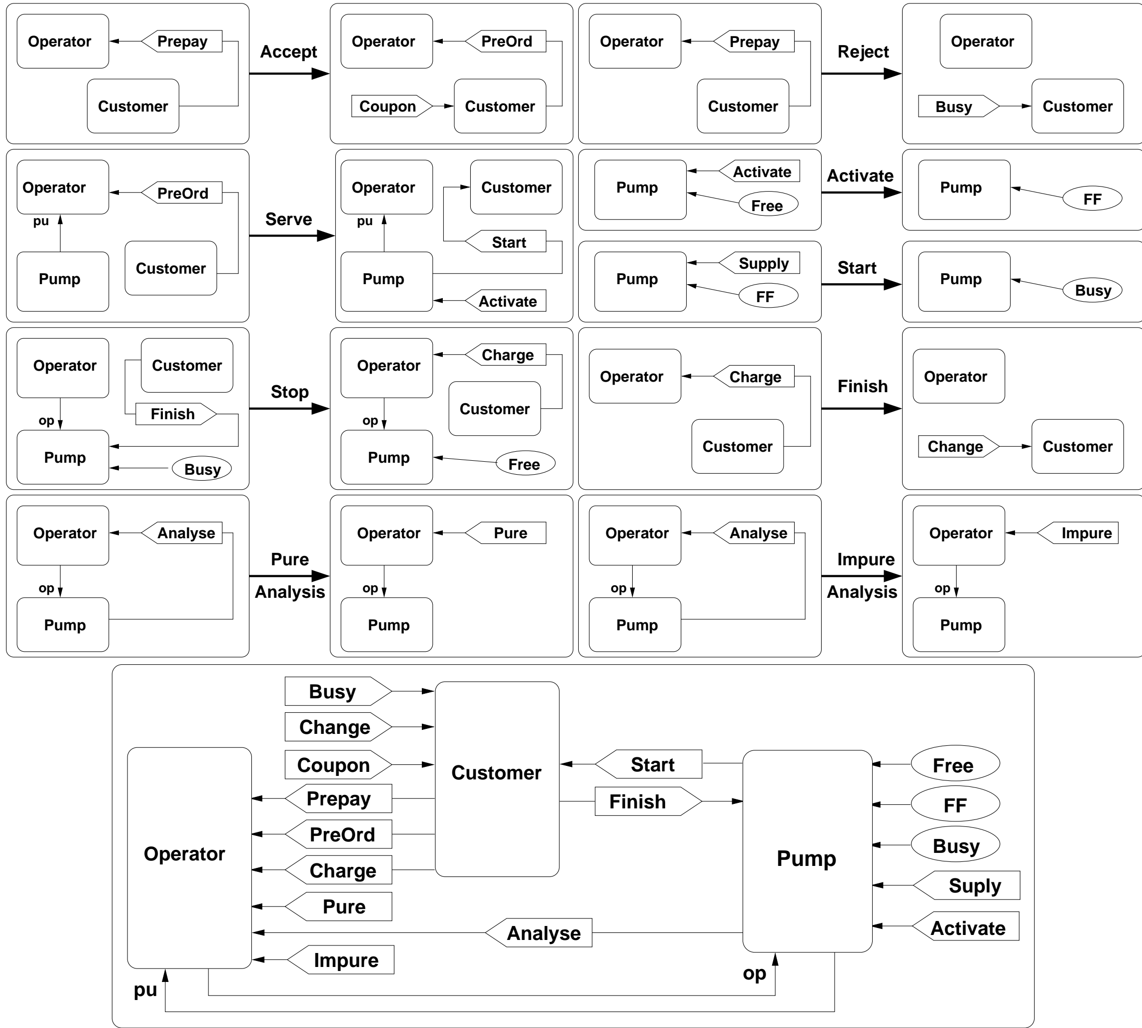


Figure 2.3: Example of GTS: a pump operator of a gas station system.

*operator can return the change to the customer (*FINISH *production). If pump is being used, the operator advises customer (sending a message* `Busy`*) to try again (*REJECT *production). Moreover, the* Operator *entity can perform an analysis of the pumps, verifying if the provided gas is pure or not (*PUREANALYSIS *or* IMPUREANALYSIS *productions, respectively).*

⌟

2.1 Operational Semantics

The operational semantics of a T-typed GTS is given by derivations. A derivation describes the application of a production to a graph representing a state of the system. In the double-pushout approach, a derivation is given in terms of pushouts in T-**Graph**. Intuitively, the pushout of two graphs with respect to another one, called interface graph, is given by the gluing of these two graphs together, identifying the items in the interface (see appendix A).

Direct derivation and derivations. If there exists an *occurrence* of the left-hand side of a *production* in a graph, this production can be applied to it. An application of a production

is given by a direct derivation in the double pushout approach. In this approach, a *direct derivation* is defined by two pushouts: the first deletes all items that shall be consumed and the second includes all items that shall be created. Figure 2.4 shows a direct derivation using the production ACCEPT.

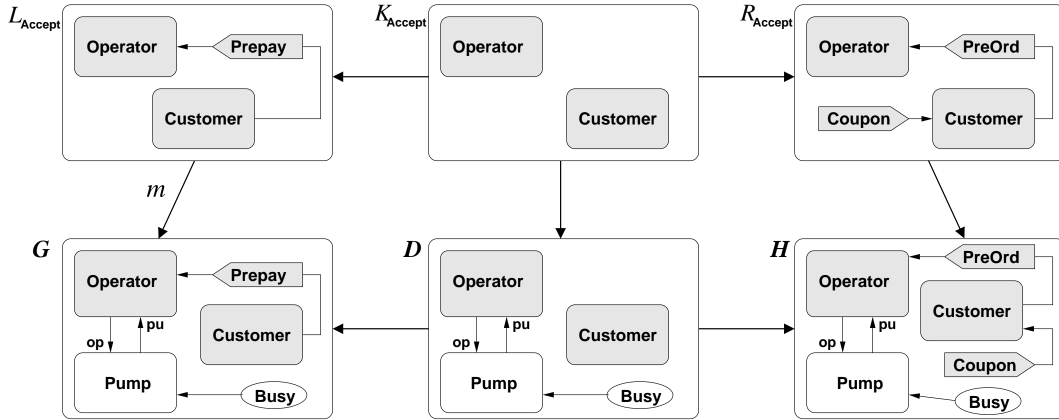


Figure 2.4: Direct derivation from G to H using ACCEPT based on m.

A *derivation* is a finite or infinite sequence of direct derivations where the final graph of one is the start graph of another.

Definition 2.4 (Direct derivation and derivations) *Given a T-typed graph G, a T-typed graph production $q = L_q \xleftarrow{l} K_q \xrightarrow{r} R_q$ and a match (i.e. an injective T-typed graph morphism) $m : L_q \rightarrow G$, a* direct derivation *from G to H using q (based on m) exists if and only if the diagram below can be constructed, where both squares are pushouts in T-$\mathbf{Graph}$. In this case the direct derivation is denoted by $\delta : G \overset{q,m}{\Rightarrow} H$ or $\delta : G \overset{q}{\Rightarrow} H$ if we do not make explicit m.*

$$\begin{array}{ccccc} L_q & \xleftarrow{l} & K_q & \xrightarrow{r} & R_q \\ {\scriptstyle m}\downarrow & (1) & {\scriptstyle k}\downarrow & (2) & \downarrow{\scriptstyle m^*} \\ G & \xleftarrow[l^*]{} & D & \xrightarrow[r^*]{} & H \end{array}$$

Given a GTS *$\mathcal{G} = \langle T, P, \pi \rangle$, a* derivation *$\rho : G_0^T \overset{p_1,m_1}{\Rightarrow} G_1^T \overset{p_2,m_2}{\Rightarrow} G_2^T \cdots$ of $\mathcal{G}$ is a finite or infinite of direct derivations $\delta_i : G_i^T \overset{p_i,m_i}{\Rightarrow} H_i^T$, where $G_{i+1}^T = H_i^T$ and $i \geq 0$. If a derivation $\rho : G_0^T \overset{p_1,m_1}{\Rightarrow} \cdots \overset{p_n,m_n}{\Rightarrow} G_n^T$ is finite we call G_0^T and G_n^T of initial and final graphs, respectively. The semantics of $\mathcal{G}$ is the class of all derivations in $\mathcal{G}$, denoted by* $\mathbf{Der}(\mathcal{G})$.

The construction of the diagram above depends on the existence of the pushout complement D. In order to guarantee this existence, m must satisfy the *gluing condition* with respect to l. This condition is partitioned in two: *dangling condition*, i.e., if a vertex is deleted, there are not edges that are incident to it; and the *identification condition*, i.e., two vertices can be identified by m, only if they are preserved. Since the considered matches are injective, the identification condition is always satisfied, so it remains to verify the dangling condition (see appendix A).

Sequential independence. If two direct derivations ρ: $G^T \overset{q_1,m_1}{\Rightarrow} X^T \overset{q_2,m_2}{\Rightarrow} H^T$ overlap only on items preserved by both, they are called *sequential independent*. For example,

the Figure 2.5 shows two independent direct derivations. We can see in this example (in a lighter colour) that the items used by both productions are preserved by them.

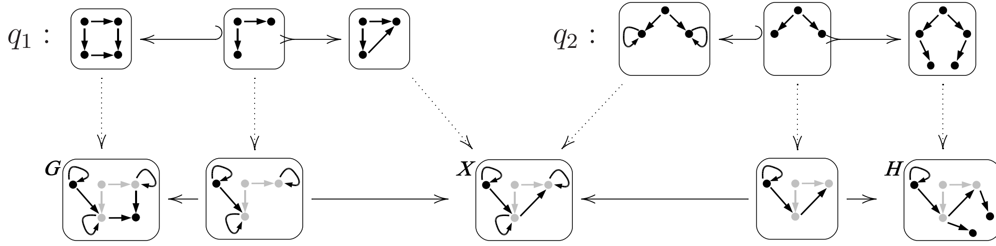


Figure 2.5: Two sequentially independent direct derivations.

Definition 2.5 (Sequential independence) *(CORRADINI et al., 1997) Let* $\delta_1 : G \overset{q_1,m_1}{\Rightarrow} X$ *and* $\delta_2 : X \overset{q_2,m_2}{\Rightarrow} H$ *(as in Figure 2.6) be two direct derivations. They are* sequentially independent *if* $m_2(L_2) \cap m_1^*(R_1) \subseteq m_2(l_2(K_2)) \cap m_1^*(r_1(K_1))$*, i.e., if the images in* X *of the* L_2 *and* R_1 *overlap only on items that are preserved by both derivation steps. In categorical terms, this condition can be expressed by requiring the existence of two arrows* $s : L_2 \to D_1$ *and* $u : R_1 \to D_2$ *such that* $r_1^* \circ s = m_2$ *and* $l_2^* \circ u = m_1^*$*. In this case* $\langle s, u\rangle$ *is said to be an independence pair of* δ_1 *and* δ_2*.*

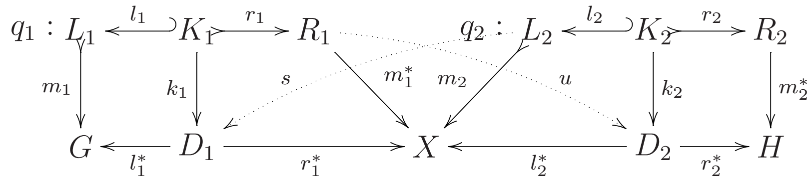


Figure 2.6: Sequential independence.

Parallel productions. A *parallel production* associated to a set P of productions is given by the disjoint union of them.

Definition 2.6 (Parallel productions) *(CORRADINI et al., 1997) Let* $q_1, \ldots, q_n$ *be productions. A* parallel production *associated with* $q_1, \ldots, q_n$ *is the production* $q_1 + \ldots + q_n$ $= \langle (q_1, in_1), \ldots, (q_n, in_n)\rangle : L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ *(depicted in Figure 2.7), where* $n \leq 0$*,* $q_i : L_i \overset{l_i}{\leftarrow} K_i \overset{r_i}{\rightarrow} R_i \in P$*, for each* $i \in \{1, \ldots, n\}$*,* L*,* K *and* R *are the coproduct objects of the graphs in* $\langle L_1, \ldots, L_n\rangle$*,* $\langle K_1, \ldots, K_n\rangle$ *and* $\langle R_1, \ldots, R_n\rangle$*, respectively. Moreover,* l *and* r *are uniquely determined by the families of arrows* $\{l_1, \ldots, l_n\}$ *and* $\{r_1, \ldots, r_n\}$*, respectively. Finally, for each* $i \in \{1, \ldots, n\}$*,* in_i *is the triple of injections* $\langle in_i^L : L_i \to L, in_i^K : K_i \to K, in_i^R : R_i \to R\rangle$*.*

By the classical parallelism theorem (CORRADINI et al., 1997) for the double-pushout approach of GTS, two sequentially independent direct derivation $G \overset{p_1}{\Rightarrow} H_1 \overset{p_2}{\Rightarrow} H$ may be replaced by a single parallel direct derivation $G \overset{p_1+p_2}{\Rightarrow} H$. But if we are using injective matches, this is not always true: if p_1 and p_2 preserve the same item, there is no injective match that allows to construct the corresponding parallel direct derivation. This situation

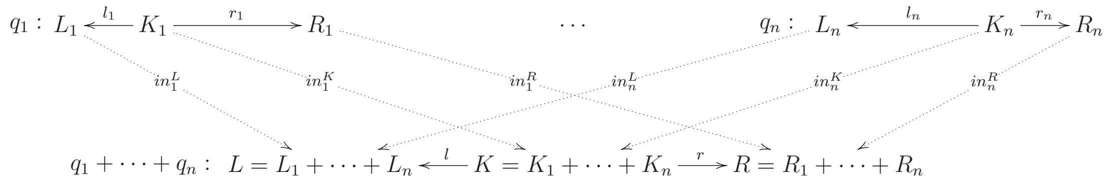


Figure 2.7: Parallel production.

makes necessary the notion of proper quotient production of a parallel production. In the following, we present the definition of proper quotient productions and the parallelism theorem considering proper quotient productions as introduced in (HABEL; MÜLLER; PLUMP, 2001).

Proper quotient productions. Because we are using injective matches, sometimes we cannot find a match from a parallel production into a graph such that this derivation is equivalent to another, where productions in P are applied in a sequential way. This is because, if we have two independent direct derivations sharing at least one preserved item, the parallel production associated to these derivations will have two different items for this shared preserved item. Since we use injective matches, we can have two different matches to pointing to shared item, but we cannot have one match from parallel production to the same shared item. So we will use *proper quotient productions* of parallel productions in order to have an equivalent production for which there is an injective match (see appendix B for formal definitions). In (HABEL; MÜLLER; PLUMP, 2001), a method for constructing the set of all proper quotient productions of $q_1 + q_2$ is presented. It is based on the idea of "gluing" L_1^T and L_2^T, K_1^T and K_2^T, R_1^T and R_2^T according to a graph S which relates the productions q_1 and q_2, based on its relationship in a derivation. For example, in Figure 2.8 we have the parallel production of q_1 and q_2 and its proper quotient production (at the bottom line of the figure) – the lighter colour in $q_1 + q_2$ indicates the items that are identified in q.

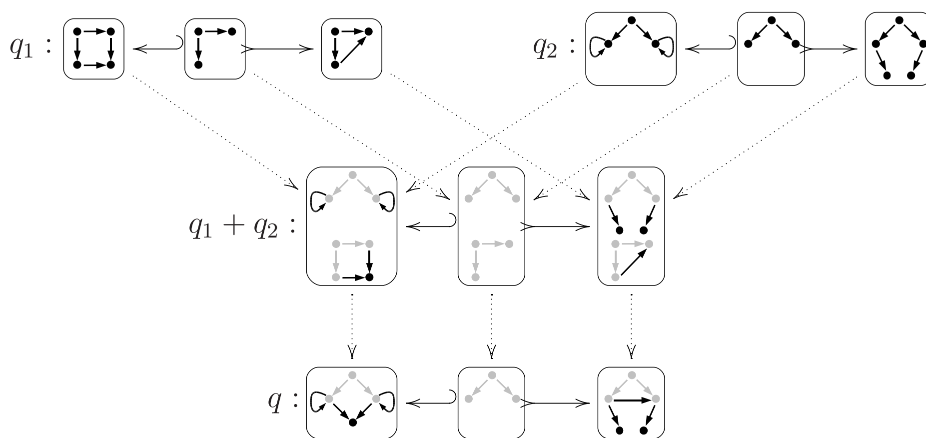


Figure 2.8: Parallel production $q_1 + q_2$ and a proper quotient production q of $q_1 + q_2$.

Parallelism theorem. Considering the above sequential independent derivations, by the Parallelism Theorem (Theorem 7.8 in (HABEL; MÜLLER; PLUMP, 2001)), we can obtain an equivalent direct derivation, applying a suitable production $q \in PQ(q_1 + q_2)$ – a proper quotient of the parallel production $q_1 + q_2$ – via an injective match. This means

that q_1 and q_2 can be applied in a reverse order resulting in the same graph H, i.e., there is a derivation $\rho'\colon G^T \overset{q_2,m'_2}{\Rightarrow} X'^T \overset{q_1,m'_1}{\Rightarrow} H^T$ where the two derivation steps are "switched".

Theorem 2.1 (Parallelism theorem) *(HABEL; MÜLLER; PLUMP, 2001) Given two productions q_1 and q_2, the following statements are equivalent:*

1. *There is a parallel direct derivation $G \overset{q}{\Rightarrow} H$, for some production $q \in PQ(q_1+q_2)$;*
2. *There are sequentially independent derivations $G \overset{q_1,m_1}{\Rightarrow} H_1 \overset{q_2,m_2}{\Rightarrow} H$;*
3. *There are sequentially independent derivations $G \overset{q_2,m'_2}{\Rightarrow} H_2 \overset{q_1,m'_1}{\Rightarrow} H$;*

Shift-equivalence. The equivalence on derivations induced by switchings of sequential independent direct derivations is called *shift-equivalence*.In Figure 2.9, a derivation shift-equivalent to the derivation in Figure 2.5 is shown. In this derivation, q_2 and q_1 are applied in reverse order. It is easy to see that we can apply the proper quotient production q (see Figure 2.8) to G^T obtaining H^T in a single step derivation, i.e., there exists the direct derivation $G^T \overset{q,m}{\Rightarrow} H^T$.

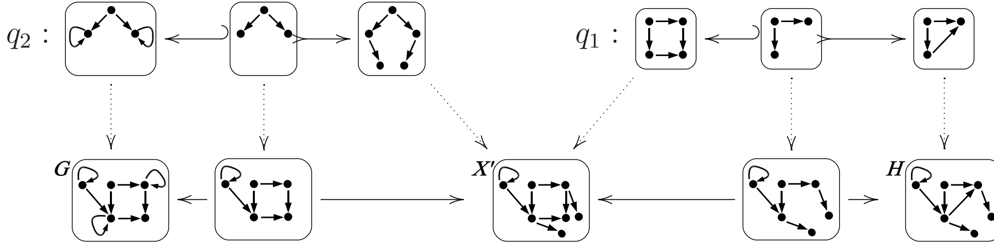


Figure 2.9: Derivation shift-equivalent to derivation in Figure 2.5.

Definition 2.7 (Shift equivalence) *(BALDAN; CORRADINI; MONTANARI, 1998a) Given a derivation $\rho = G \overset{q_1,m_1}{\Rightarrow} X \overset{q_2,m_2}{\Rightarrow} H$, consisting of two sequentially independent direct derivations, $\rho' = G \overset{q_2,m'_2}{\Rightarrow} X' \overset{q_1,m'_1}{\Rightarrow} H$ is obtained as in the Theorem 2.1, where productions q_1 and q_2 are applied in the reverse order. ρ' is called a switching of ρ and is denoted by $\rho \sim^{sh} \rho'$. The* shift equivalence *$\equiv^{sh}$ on derivations is the transitive and "context" closure of $\sim^{sh}$, i.e., the least equivalence, containing $\sim^{sh}$, such that if $\rho \equiv^{sh} \rho'$ then $\rho_1; \rho; \rho_2 \equiv^{sh} \rho_1; \rho'; \rho_2$.*

Abstract traces. If we want to abstract the concrete identities of the items of typed graphs involved in a derivation, i.e., considering graphs with the same structure as being the same (abstract) graph, we can consider classes of *abstract equivalent derivations*. Combining shift-equivalence with abstraction equivalence we obtain the so-called *abstract truly-concurrent equivalence*. Equivalence classes of derivations, with respect to abstract truly-concurrent equivalence, are denoted as $[_]_a$ and are called *abstract traces*.

Definition 2.8 (Abstraction equivalence) *Let $\rho : G_0 \overset{q_1,m_1}{\Rightarrow} \cdots \overset{q_n,m_n}{\Rightarrow} G_n$ and $\rho' : G'_0 \overset{q'_1,m'_1}{\Rightarrow} \cdots \overset{q'_n,m'_n}{\Rightarrow} G'_n$ (whose i^{th} step is depicted in the low arrows of the diagram in Figure 2.10) be two derivations. Then they are* abstraction equivalent*, denoted by $\rho \equiv^{abs} \rho'$, if $n = n'$,*

$q_i = q'_i$ for $1 \leq i \leq n$ and there exists a family of isomorphisms $\{\theta_{X_{q_i}} : X_{q_i} \mapsto X'_{q_i} | X \in \{G, D\}, 0 \leq i \leq n\}$, between corresponding graphs in the two derivations, such that the resulting diagram (step i depicted in Figure 2.10) commutes. Equivalence classes of decorated derivations with respect to $\equiv^{abs}$are called abstract derivations *and are denoted by $[\rho]_{abs}$, where ρ is an element of the class (BALDAN; CORRADINI; MONTANARI, 1998a).*

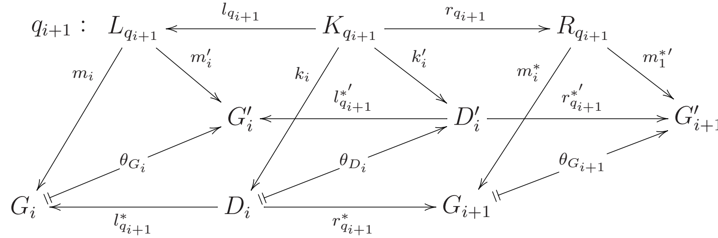


Figure 2.10: Isomorphism of derivations.

Definition 2.9 (Abstract truly-concurrent equivalence) *The* abstract truly-concurrent equivalence $\equiv^a$ *on derivations is the transitive closure of the union of the relations $\equiv^{abs}$ and $\equiv^{sh}$. Equivalence classes of derivations with respect to $\equiv^a$ are denoted as $[\rho]_a$ and are called* abstract traces *(BALDAN; CORRADINI; MONTANARI, 1998a).*

2.2 Category GTS

We can relate two GTSs by mapping the source type graph and production names into the target ones. This mapping is given by morphisms in category of GTS. This definition will be used as the base for definitions of morphisms introduced in the next chapters.

Various notions of morphisms for graph transformation systems have been introduced in literature (CORRADINI et al., 1996; GROSSE-RHODE; PARISI-PRESICCE; SIMEONI, 1999; HECKEL et al., 1996; RIBEIRO, 1996; BALDAN, 2000). Our morphisms on GTSs are a slight variation of morphisms in (BALDAN, 2000). The mapping between type graphs must allow to forget some items, therefore we will use partial morphisms to relate them.

Partial graph morphisms A partial relation between two graphs maps only part of the source graph to the target one. Here, we use the definition that simulates a partial graph morphism using a pair of total graph morphisms, with origin in a subgraph $dom(f)$ of the source graph (domain of the partial graph morphism). The first morphism maps $dom(f)$ into source graph and the second one maps $dom(f)$ into target graph.

Definition 2.10 (Partial graph morphisms) *A* partial graph morphism $f : G_1 \rightharpoonup G_2$ *is a total graph morphism from a subgraph of G_1, called $dom(f)$, to G_2, and is depicted as $G_1 \stackrel{l_f}{\hookleftarrow} dom(f) \stackrel{r_f}{\rightarrow} G_2$.*

Example 2.4 (Partial graph morphisms) *In Figure 2.11, a partial morphism f_T from the type graph of the pump operator GTS to a graph T_1 is depicted. The mapping is described by numbering, where the items without a number are not in the range of the morphism. It means, that items in $T_{PumpOper}$, that are not in $dom(f_T)$ are all forgotten by morphism f_T.*

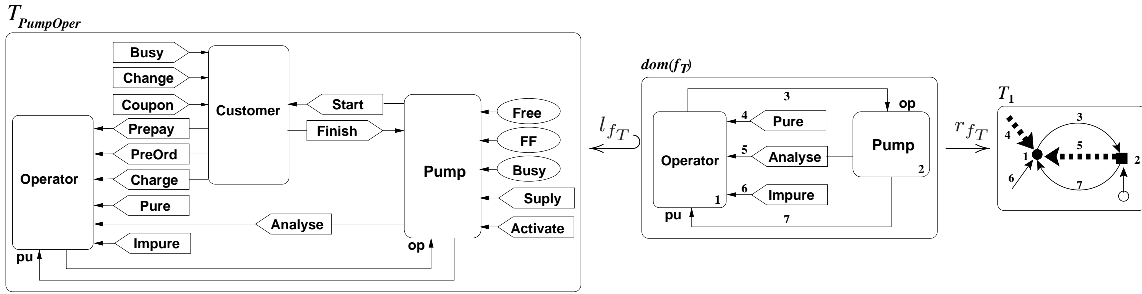


Figure 2.11: Partial morphism f_T.

┘

Proposition 2.1 *Graphs and partial graph morphisms form a category, denoted by* $\mathbf{Graph^P}$ *(see (HECKEL et al., 1996)).*

The mapping between productions must preserve them, so we will require that the target production be a translation of the source one, with respect to the mapping between the type graphs. Formally, this is given by a pullback functor, that, intuitively, forgets all items whose type is not preserved by the type graph mapping and then by typing the production over the target type graph. For example, the translation of the production STOP by means the pullback functor, with respect to f_T is shown on the center of Figure 2.12 and, on the right, it is typed over T_1. One can see that production STOP' $\downarrow T_1$ is not a consuming production, since it defines an isomorphism.

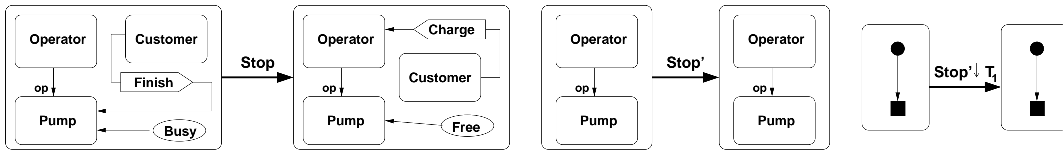


Figure 2.12: Translation of STOP production with respect to f_T: STOP production (left), pulling back along l_{f_T} (center) and STOP' typed over T_1 (right).

Definition 2.11 (Pullback functor) *Given an object A of a category $\mathcal{C}$, the* slice category *$\mathcal{C}{\downarrow}A$ has all $\mathcal{C}$-arrows with target A as objects; an arrow $h\colon f \to g$ in $\mathcal{C}{\downarrow}A$ is a $\mathcal{C}$-arrow h such that $g \circ h = f$.*[2]

Let $m\colon A \to B$ be an arrow in a category $\mathcal{C}$ with pullbacks. Chosen a pullback square as (1) below for any $f\colon D \to B$, the pullback functor *along $m\colon A \to B$, denoted $m^*\colon \mathcal{C}{\downarrow}B \to \mathcal{C}{\downarrow}A$, maps an object $(f\colon D \to B) \in \mathcal{C}{\downarrow}B$ to $(m^*(f)\colon m^*(D) \to A) \in \mathcal{C}{\downarrow}A$.*

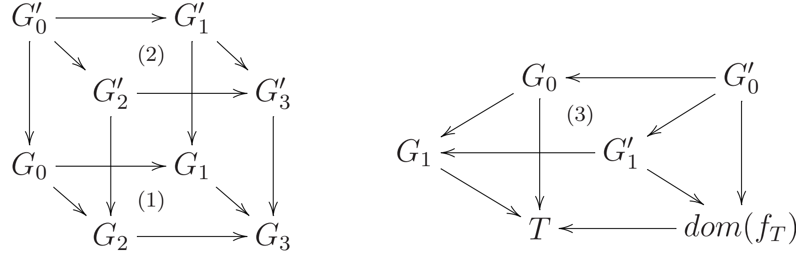
Given arrows $m\colon A \to B$ and $f\colon D \to B$ of $\mathcal{C}$, we write $g \cong_{C\downarrow A} m^*(f)$ if there exists an arrow $C \to D$ such that square (2) below is a pullback.

$$\begin{array}{ccc} m^*(D) & \longrightarrow & D \\ {\scriptstyle m^*(f)}\downarrow & (1) & \downarrow{\scriptstyle f} \\ A & \xrightarrow[m]{} & B \end{array} \qquad\qquad \begin{array}{ccc} C & \longrightarrow & D \\ {\scriptstyle g}\downarrow & (2) & \downarrow{\scriptstyle f} \\ A & \xrightarrow[m]{} & B \end{array}$$

[2]Thus, for example, T-$\mathbf{Graph} = \mathbf{Graph}{\downarrow}T$ as in Definition 2.1.

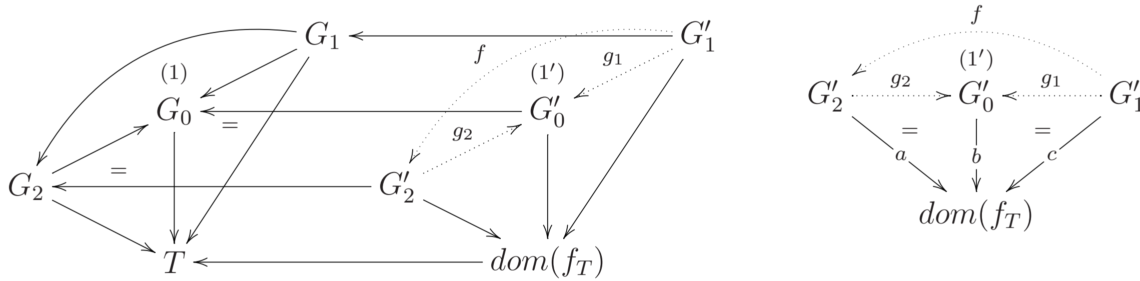
In the following lemmas we prove that pullback functor preserve pushouts and commutativity.

Lemma 2.1 (pullback functor preserves pushouts) *Let the square (1) be a pushout in T-$\mathbf{Graph}$. Let G'_i be a graph obtained from G_i, applying the pullback functor. Then the square (2) is a pushout in $dom(f_T)$-$\mathbf{Graph}$.*



Proof: By definition of pullback functor, all arrows $g : G_0 \to G_1$ in T-$\mathbf{Graph}$, we have $g' : G'_0 \to G'_1$ in $dom(f_T)$-$\mathbf{Graph}$, such that the square (3) above is a pullback. It holds by definition of pullback functor and Lemma A.5, since $\langle G'_0, G_0, dom(f_T), T\rangle$ and $\langle G'_1, G_1, dom(f_T), T\rangle$ are pullbacks. Therefore, the right, front, left and back squares, in cube diagram above, are pullbacks. Thus, by Lemma A.6, the top square is a pushout, as well. □

Lemma 2.2 (pullback functor preserves commutativity) *Let* (1) *be a commuting diagram in T-$\mathbf{Graph}$. Let G'_i be a graph obtained from G_i, applying the pullback functor. Then* $(1')$ *is a commuting diagram in $dom(f_T)$-$\mathbf{Graph}$.*



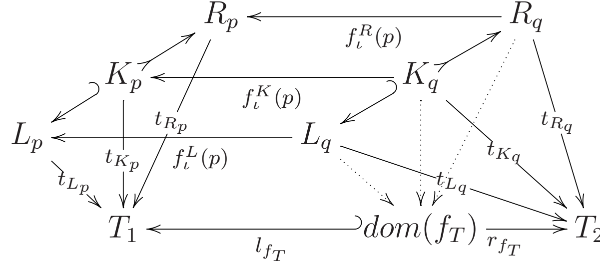
Proof: We must prove that $a \circ f = g_1$. By Lemma A.5, there exist $g_1 : G'_1 \to G'_0$, $g_2 : G'_2 \to G'_0$ and $f : G'_1 \to G'_2$ such that $b \circ g_1 = c$, $b \circ g_2 = a$ and $a \circ f = c$. Thus, we have $a \circ f = b \circ g_1 = b \circ g_2 \circ f$. Since b is injective, then $a \circ f = g_1$. □

GTS morphism A GTS *morphism* $f : \mathcal{G}_1 \to \mathcal{G}_2$ is given by two components. The first, f_T, is a partial and non injective morphism between the type graphs, that besides allowing to forget some type item, allows to identify two items of the first GTS into one of the second. The second component, f_P, is a partial mapping between the production names such that the mapped the productions are preserved.

Definition 2.12 (GTS morphism) *Let $\mathcal{G}_1 = \langle T_1, P_1, \pi_1\rangle$ and $\mathcal{G}_2 = \langle T_2, P_2, \pi_2\rangle$ be* GTS*s. A* GTS morphism *$f : \mathcal{G}_1 \to \mathcal{G}_2$ is a pair $f = \langle f_T, f_P\rangle$, where*

- *$f_T : T_1 \rightharpoonup T_2$ is a partial graph morphism;*
- *$f_P : P_1 \to P_2$ is a partial function on production names, such that for all $p \in P_1$:*

- *if $f_P(p) = q$, then there are morphisms $f_\iota^L(p)$, $f_\iota^K(p)$ and $f_\iota^R(p)$ such that the diagram below commutes, and $f_\iota^X(p) \cong_{\mathbf{Graph}\downarrow X_p} t^*_{X_p}(l_{f_T})$ (See Definition 2.11) for $X \in \{L, K, R\}$;*
- *if $f_P(p)$ is not defined, then morphisms $l^*_{f_T}(K_p) \to l^*_{f_T}(L_p)$ and $l^*_{f_T}(K_p) \to l^*_{f_T}(R_p)$ are isomorphisms.*



Example 2.5 (GTS morphism) *If we consider a* GTS *$\mathcal{G}_1$, where the type graph is T_1 in Figure 2.11 and the productions in Figure 2.13, we can have a* GTS *morphism f from the* GTS *of the pump operator system to $\mathcal{G}_1$. The component on the type graphs f_T is defined as in Figure 2.11 and the component on the production sets maps* PUREANALISYS *and* IMPUREANALISYS *into q_1 and q_2, respectively. The other productions are not mapped, because their translations result in isomorphisms, as we can see for the production* STOP *in Figure 2.12.*

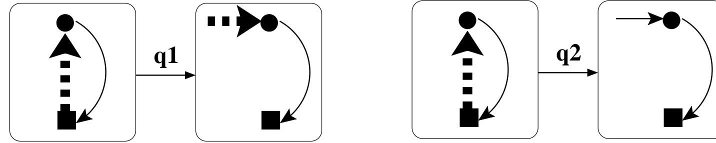


Figure 2.13: Productions of $\mathcal{G}_1$.

┘

Then in the following proposition we show that GTSs and GTS morphisms form a category, proving that the composition and identities are well defined GTS morphisms, i.e., they preserve the existence of a mapping between components of productions, required by Definition 2.12.

Proposition 2.2 GTS*s and* GTS *morphisms form a category, denoted by* $\mathbf{GTS}$*, in which composition and identities are defined componentwise. (see (HECKEL et al., 1996))*

Chosen a pullback functor $l^*_{f_T}$, as in Definition 2.12, the partial morphism $f_T : T_1 \rightharpoonup T_2$ induces a *retyping functor* $f_T^{\leftrightarrow} : T_1\text{-}\mathbf{Graph} \to T_2\text{-}\mathbf{Graph}$, defined on objects as $f_T^{\leftrightarrow}(t_G : G \to T_1) = r_{f_T} \circ l^*_{f_T}(t_G)$. The condition on morphisms involving the pullback squares ensures that all the items in X_p whose type is preserved by f_T occur in $X_{f_P(p)}$. Thus, GTS morphisms are simulations, meaning that, for a derivation δ in $\mathcal{G}_1$, (any choice of) the retyped diagram $f_T^{\leftrightarrow}(\delta)$ is a derivation in $\mathcal{G}_2$. This fact is showed in the following proposition.

Proposition 2.3 (GTS morphisms preserve derivations) *Let $f : \mathcal{G}_1 \to \mathcal{G}_2$, and let $\delta_1 : G_1 \Rightarrow_{p_1} H_1$ be a direct derivation in $\mathcal{G}_1$. Then there exists corresponding direct derivation $f_T^{\leftrightarrow}(\delta_1) = \delta_2 : f_T^{\leftrightarrow}(G_1) \Rightarrow_{f_T(p_1)} f_T^{\leftrightarrow}(H_1)$ in $\mathcal{G}_2$ (see (HECKEL et al., 1996)).*

3 TRANSACTIONAL GRAPH TRANSFORMATION SYSTEMS

In this chapter, we present the basic definitions of a GTS extension with support to the notion of transactions. A transaction in this approach describes a set of actions that must be executed in an atomic way. The notion of transaction has been originally defined and studied in the realm of database management systems, and only later it has been considered in programming and specification formalisms, like process calculi, programming languages and Petri nets. A transaction represents a unit of interaction with the management system, that is treated in a coherent and reliable way, independently of other transactions, and that must be either entirely completed or aborted.

Transactions can be introduced in different ways in a modelling, specification or programming formalism. In *control-centered formalisms*, like process calculi and programming languages, where the execution of computations is ruled by expressive control mechanisms, typically new control structures are introduced for starting/committing transactions. In *data-centered formalisms* (see (SHIELDS, 2007) and Section 2.3 of (GAJSKI et al., 2000) for data vs. control flows), like rewriting formalisms and Petri nets, where the control structures are typically poor and the emphasis is on the structure of the state that evolves during a computation, transactions are more naturally defined indirectly, by identifying parts of the state which represent temporary (or "unstable") resources, only visible within a transaction. This is the approach that has been defined for *zero-safe nets* (BRUNI; MONTANARI, 2001, 2000).

3.1 Introduction to Transactional GTS

Inspired by the work on zero-safe nets, *transactional graph transformation systems* (T-GTSs), introduced in (BALDAN et al., 2008), are an extension to the double-pushout (DPO) approach to graph transformation, providing a simple way of expressing transactional activities. The basic tool is a typing mechanism for graphs which induces a distinction between *stable* and *unstable* graph items. Given a typed graph, representing a system state, we can identify a subgraph which represent its "stable" part, i.e., the fragment of the state which is visible for an external observer. Transactions in a T-GTS are thus abstract, "minimal" computations starting from a completely stable graph, evolving through graphs with unstable items and eventually ending up in a new stable state. Definitions and propositions of this section, as well as corresponding proofs, can be found in (BALDAN et al., 2008).

Transactional GTS. A *transactional* GTS is defined as a collection of productions, typed over a fixed type graph – i.e. a graph transformation system – with a subgraph of this type

graph determining the stable items, i.e., the items that are visible for an external observer. The other items are considered temporary, so, they are invisible, or unstable.

Definition 3.1 (Transactional GTS) *A* transactional GTS *is a pair* $\langle \mathcal{G}, T_s \rangle$*, where* $\mathcal{G}$ *is a* T*-typed* GTS *and* T_s *is a subgraph of the type graph* T *of* $\mathcal{G}$*, called the* stable type graph.

Example 3.1 (transactional GTS) *One can consider some activities of the system described in Example 2.3 being executed as transactions. In order to do this, we must define the temporary items of the system, i.e., the items that cannot be seen at a more abstract level. For example, when the operator receives a prepayment we can be interested only in the supplied gas, without considering the series of steps to achieve this goal. To do this, the messages* `PreOrd`*,* `Activate` *and* `Charge` *will become unstable in the transactional* GTS $\mathcal{P}umpOper$*, as shown in Figure 3.1. Thus, when an operator receives a prepayment and accepts it (*ACCEPT *and* SERVE *productions), a* `Start` *message (signing that gas can be supplied) is sent to customer, but the activation of the pump becomes unobservable (*ACTIVATE *production). In the type graph (depicted at the bottom of the figure) we marked the temporary (or unstable) messages, that are used to perform some steps of the pump operator activities, with dashed lines, while the stable ones are depicted with solid lines.*

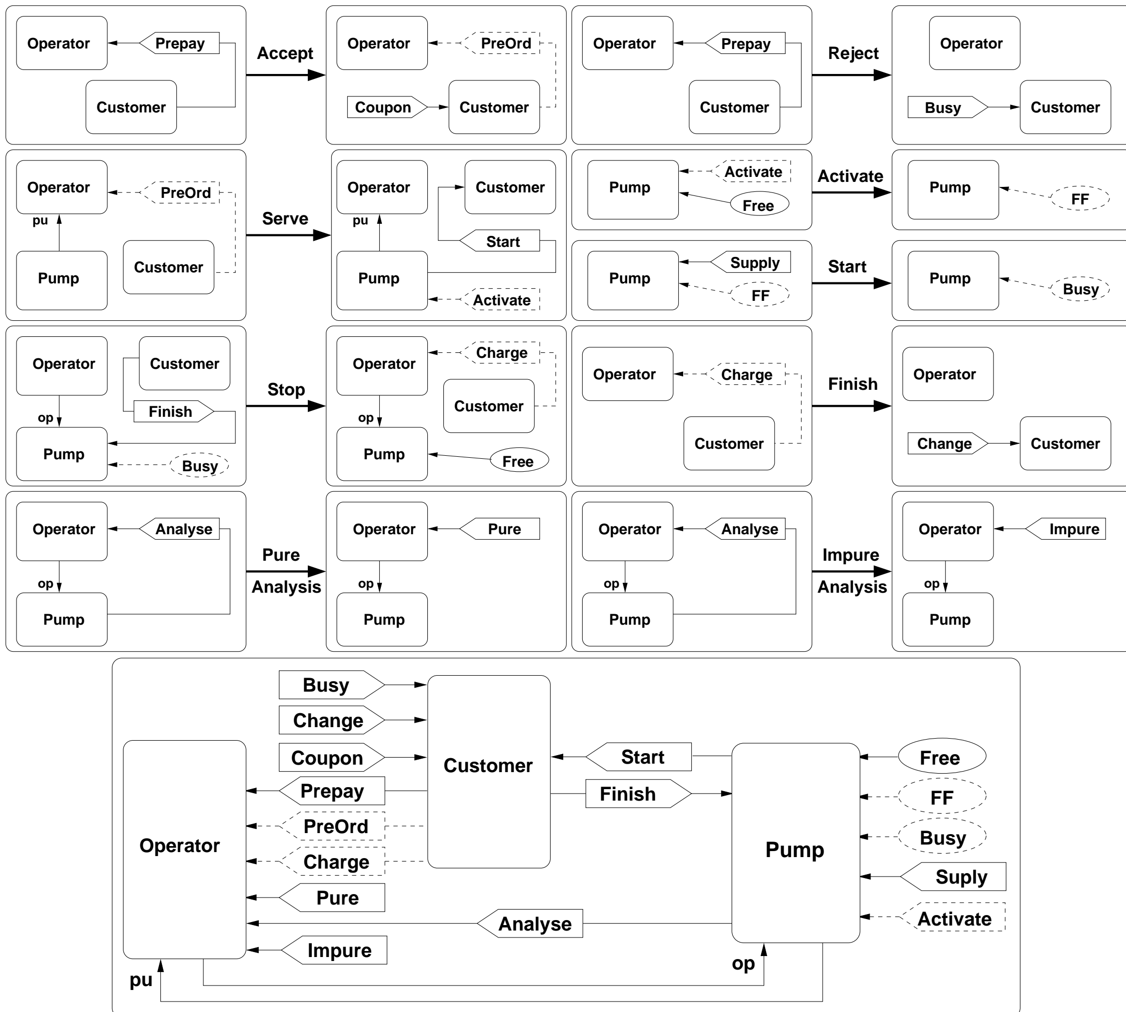


Figure 3.1: Transactional GTS $\mathcal{P}umpOper$ for a pump operator of a gas station.

In Figure 3.2, the T-GTS $\mathcal{C}ustomer$*, modelling the customer system, is shown. The type graph (bottom) shows us the same three entities in the pump operator system and its behaviour is described by the productions at the top of this figure. In this system, a* Customer *entity has a cyclic behaviour. It initiates prepaying an amount of money in order to get some gas (*INIT *production). If the pump is being used the customer is advised by the operator (*REJECT *production), then it retries to be supplied (with a* `Busy` *message). Otherwise, the operator accepts the prepaying and the customer can initiate to supply itself with gas (*STARTSUPPLY *production). It goes until the wanted volume of gas was reached, when the customer can stop the pump (*ENDSUPPLY *production) and receive the change and a coupon from the operator. At this moment the pump is set free and then the customer is ready to start all the process again (*RESTART *production).*

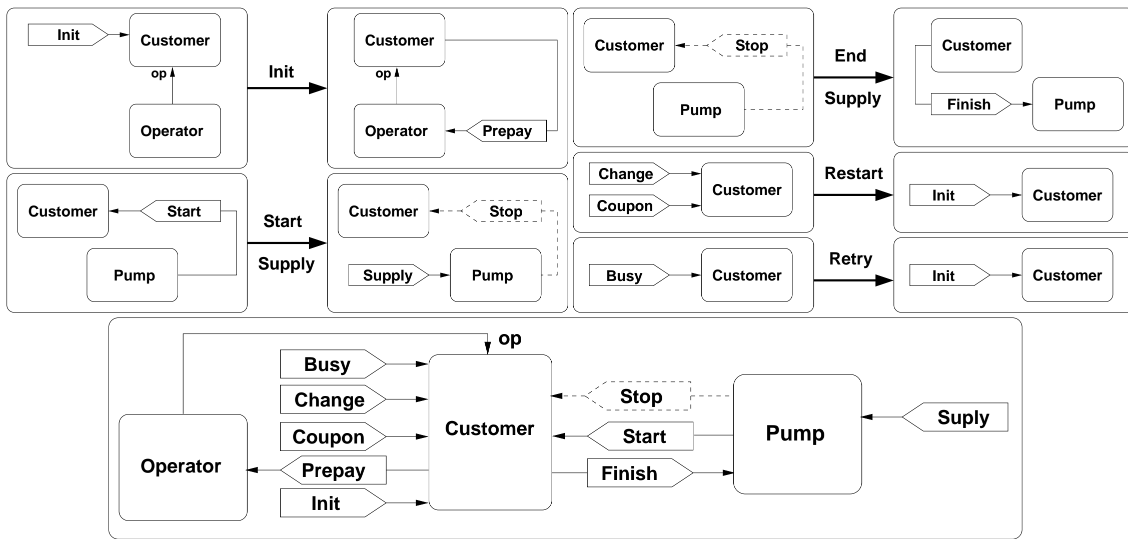


Figure 3.2: Transactional GTS $\mathcal{C}ustomer$ for a customer of a gas station.

┘

Inspired by the approach for Petri nets proposed in (BRUNI; MONTANARI, 2000) and extended to nets with read arcs in (BRUNI; MONTANARI, 2001), stable steps, transactions and abstract transactions are introduced. In order to give these definitions, we need a notion of stable graphs.

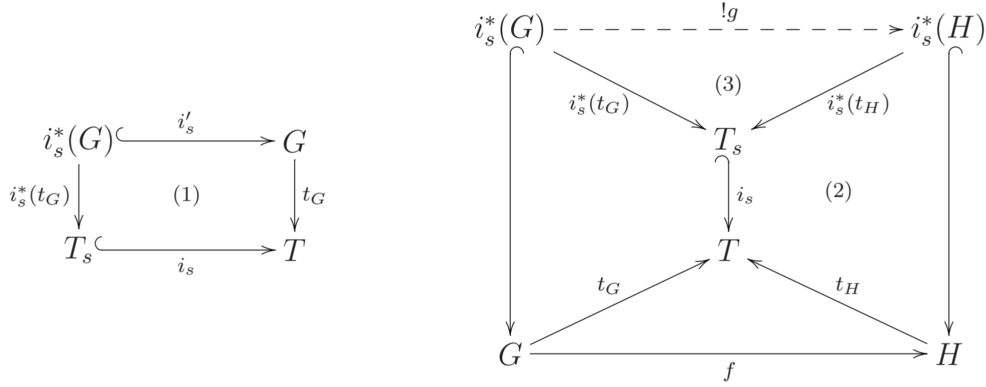
Stabilising functor. We can get the stable part of a graph considering only the items typed over the stable type graph. Formally, this operation is defined by a functor. We denote by $\mathcal{S}\colon T\text{-}\mathbf{Graph} \to T_s\text{-}\mathbf{Graph}$ the functor that maps each graph G, typed over T, to its subgraph consisting of its stably-typed items only, and each morphism to its restriction to stable items: thus $\mathcal{S}$, called the *stabilising functor*, is a concrete choice for the pullback functor induced by i_s, where the morphism $\mathcal{S}(G) \hookrightarrow G$ is an inclusion.

Definition 3.2 (Stabilising functor) *Let* $\mathcal{Z} = \langle\langle T, P, \pi\rangle, T_s\rangle$ *be a* T-GTS *and* $i_s\colon T_s \hookrightarrow T$ *be the inclusion morphism determined by subgraph* T_s *of* T*. The* stabilising functor $\mathcal{S} : T\text{-}\mathbf{Graph} \to T_s\text{-}\mathbf{Graph}$ *is defined by:*

- *on objects: for each* G_T*,* $\mathcal{S}(G^T) = \langle i_s^*(G), i_s^*(t_G)\rangle$*, where* i_s' *(see diagram* (1) *below) is an inclusion, and*

- *on morphisms: for each morphism* $f : G_T \to H_T$, $\mathcal{S}(f) = g$*, where* $g : i_s^*(G) \to i_s^*(H)$ *is the morphism uniquely determined by universal property of pullback* (2) *below, such that* (3) *commutes.*

Sometimes, when it does not cause confusion, we will use $\mathcal{S}(G^T)$ *to denote the* T-**Graph** *graph* $\langle i_s^*(G), i_s \circ i_s^*(t_G)\rangle$.



In Figure 3.3, we can see a stabilised graph, where $\mathcal{S}(G)$ is the stable part of G.

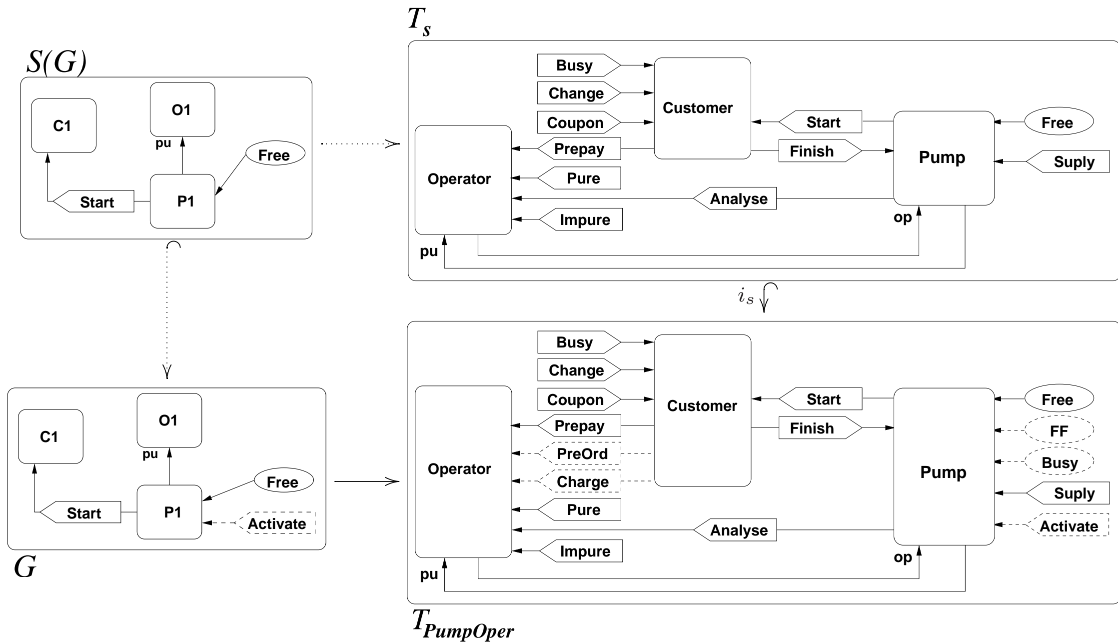


Figure 3.3: Stabilised graph.

If a graph has only visible items, i.e., it is type over T_s, it is called *stable*, otherwise, it is *unstable*.

Definition 3.3 (Stable graph) *A* T-**Graph** G^T *is called* stable *if* $\mathcal{S}(G^T) = G^T$ *(i.e., if the morphism* i'_s *– in diagram* (1) *above – is the identity). It is called* unstable *otherwise.*

Stabilised GTS. The stabilising functor can be applied pointwise to any production of a given T-GTS, producing a *stabilised* GTS – a GTS typed over the stable type graph.

Definition 3.4 (Stabilised GTS) *Given a T-typed* T-GTS $\mathcal{Z} = \langle\langle T, P, \pi\rangle, T_s\rangle$, *the* stabilised GTS $\mathcal{S}(\mathcal{Z})$ *is given by* $\langle T_s, P, \pi'\rangle$, *where* $\pi'(q) = \mathcal{S}(\pi(q))$ *for any* $q \in P$.

The functor $\mathcal{S}$, when applied to a derivation in a given T-GTS $\mathcal{Z} = \langle\mathcal{G}, T_s\rangle$, produces a derivation in $\mathcal{S}(\mathcal{Z})$. An indirect proof of this fact can be obtained by observing that there exists a typed GTS morphism $\langle id_{T_s} : T \rightharpoonup T_s, id_P\rangle$, in the sense of (BALDAN; CORRADINI; MONTANARI, 1998b), which essentially forgets about the unstable items. Then, using the fact that GTS morphisms are simulations (Proposition 2.3), one can conclude the mentioned fact.

Proposition 3.1 *Let* $\mathcal{Z} = \langle\mathcal{G}, T_s\rangle$ *be a* T-GTS *and let* $\rho = G_0 \overset{q_1,m_1}{\Rightarrow} G_1 \overset{q_2,m_2}{\Rightarrow} \ldots \overset{q_n,m_n}{\Rightarrow} G_n$ *be a derivation in* $\mathcal{Z}$. *Then*

$$\mathcal{S}(\rho) = \mathcal{S}(G_0) \overset{q_1,\mathcal{S}(m_1)}{\Rightarrow} \mathcal{S}(G_1) \overset{q_2,\mathcal{S}(m_2)}{\Rightarrow} \ldots \overset{q_n,\mathcal{S}(m_n)}{\Rightarrow} \mathcal{S}(G_n)$$

is a derivation in $\mathcal{S}(\mathcal{Z})$.

Now, we are ready to define stable steps and transactions. In the following, $\mathcal{Z} = \langle\mathcal{G}, T_s\rangle$ is a fixed T-GTS.

Stable step and transaction. A *stable step* is, intuitively, a computation which (1) starts and ends in stable states. Moreover, stable items which are generated are "frozen", in the sense that they cannot be preserved nor consumed by other productions inside the same step; similarly, stable items which are deleted cannot be preserved by other productions. Therefore, (2) the dependencies between productions occurring in a step are induced by unstable items: this implies that at the abstract level, where unstable items are forgotten, all such productions are applicable in parallel. A *transaction* is a stable step where (3) the start graph contains exactly what it needs to reach a successful end – so, in a computation, it can be embedded into a larger context – and (4) none of its sub-derivations is a transaction as well, so all intermediated graphs must be unstable. The formal definition is given as follows, where each above restrictions are defined by conditions with corresponding numbers.

Definition 3.5 (Stable step and transaction) *A* stable step *is a derivation* $\rho = G_0 \overset{q_1,m_1}{\Rightarrow} G_1 \overset{q_2,m_2}{\Rightarrow} \ldots \overset{q_n,m_n}{\Rightarrow} G_n$ *which satisfies the following properties:*

1. G_0 *and* G_n *are stable graphs;*

2. *the derivation* $\mathcal{S}(\rho)$ *is equivalent in* $\mathcal{S}(\mathcal{G})$ *to a direct derivation via a proper quotient of the production* $q_1 + \ldots + q_n$ *and a suitable match* m, *i.e.,* $\mathcal{S}(G_0) \overset{PQ(q_1+\ldots+q_n),m}{\Longrightarrow} \mathcal{S}(G_n)$ *is a derivation in* $\mathcal{S}(\mathcal{G})$.

A transaction *is a stable step additionally satisfying*

3. *the match* m *is an isomorphism;*

4. *any intermediate graph* G_i *(*$i \neq 0, n$*) is not stable.*

Remark 3.1 Note that by restriction (2) (a stable item created by a transaction cannot be used within this transaction) a stable connected graph must be created by a single production, this is because we need to preserve a vertex in order to add an edge over it. Therefore, one cannot define a T-GTS which generates stable connected graphs of arbitrary size.

Example 3.2 (transaction) *Figure 3.4 shows the derivation* $G_0 \overset{\text{ACCEPT},m_1}{\Longrightarrow} \cdots \overset{\text{FINISH},m_6}{\Longrightarrow} G_6$ *of the* T-GTS $\mathcal{P}umpOper$ *(Example 3.1).* G_0 *and* G_6 *are the initial and the final graphs, respectively, and the morphisms are described by numbering, i.e., two related items have the same number. We can see that this derivation corresponds to a transaction,*

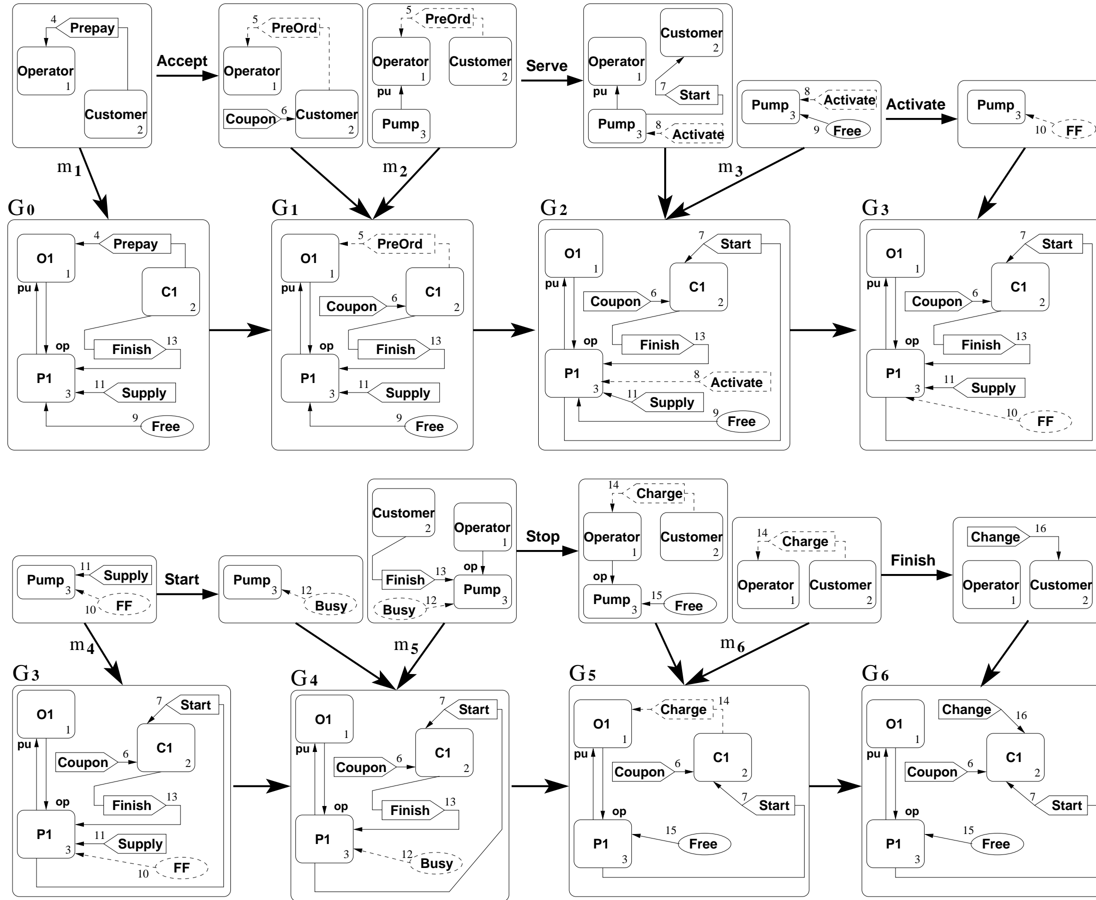


Figure 3.4: A transaction of the T-GTS $\mathcal{P}umpOper$ in Example 3.1.

since it respects the requirements of Definition 3.5, i.e., 1) *the initial (*G_0*) and final (*G_6*) graphs are stable;* 2) *dependency between direct derivation is given only on stable items, since the unique items created by one production and consumed by another are unstable (e.g.* ACCEPT *production creates* `Preord` *which is consumed by* SERVE *production);* 3) *all items in* G_0 *are used (i.e. at least one production preserves or consumes items in* G_0*); and* 4) *intermediate states* G_1 *to* G_5 *are not stable.*

⌟

Actually, since we are considering a concurrent model of computation, the fact that all the intermediate graphs are not stable should not be related to the specific order in which productions are applied. Rather, this property should still hold for any derivation which is obtained from the original one by exchanging independent steps of computation, i.e., shift-equivalent derivations. Moreover, we also want to abstract the concrete identities of items in the involved graphs, i.e., considering graphs up to isomorphism.

Abstract transaction. An *abstract transaction* is defined as an abstract trace (a class of shift-equivalent derivations where the involved graphs are considered up to isomorphism) containing only transactions.

Definition 3.6 (Abstract stable transaction) *An* abstract stable transaction *is an abstract derivation trace $[\rho]_a$, such that for any $\rho' \in [\rho]_a$ the derivation ρ' is a stable transaction. The class of all abstract stable transaction of a* T-GTS $\mathcal{Z}$ *is denoted by* $\mathbf{absTrans}(\mathcal{Z})$.

It follows from the definition that if two abstract transactions can be applied in parallel to a stable graph, then all direct derivations of one are independent of the direct derivations of the other one. Thus, as desired, the transactions can be interleaved in an arbitrary way.

In order to use transactions to define relations between transactional graph transformations systems – e.g., a refinement relation, where a production is associated to a transaction – we must be able to compose transactions. This composition is defined as an operation in an appropriated category. The definition of transactions as derivation traces can be hard to cope with, since traces are equivalence classes and their composition is not simple. Therefore, a more manageable characterisation of abstract stable transactions would be desirable. This characterisation can be given by using the idea of graph processes (BALDAN; CORRADINI; MONTANARI, 1998a), where one graph process represents a whole class of shift-equivalent derivation, i.e., a derivation trace. In following sections we will present this characterisation and use it to define a notion of refinement or implementation morphism and abstract GTS.

3.2 Category TGTS

In this section we introduce a definition T-GTS morphism, which relates productions of source T-GTS to productions of target one, and prove that T-GTSs and these morphisms form a category. These results will be used to characterise transactions as graph processes in subsection 3.4.1.

Two T-GTSs are related by a special GTS morphism between their underlying GTSs. This morphism must preserve transactions, i.e., each transaction of the source T-GTS must be "translated" to a transaction in the target T-GTS. Thus, we impose two restrictions to this GTS morphism to guarantee that transactions are preserved: the first one requires that it is total on unstable items and that these items are preserved – each unstable item is mapped into another unstable item; and the second one requires that the stable items are preserved – each mapped stable item must be related to another stable item. Preserving stable items we avoid that a stable graph in the source becomes an unstable graph in the target. In addition, requiring preservation of and totality on unstable items we avoid to turn unstable intermediary states of a transaction into stable graphs. For example, Figure 3.5 illustrates what can happen if the restrictions are not imposed: in the left, it is showed a mapping between type graphs (given by numbers) that is not total on unstable items (the unstable arrow 9 of T_1 is not mapped) and it does not preserve the stable items (the stable arrow 8 if T_1 is mapped into an unstable arrow in T_2); in the right, it is showed (at the top) a transaction in the source T-GTS and (at the bottom) its translation, which is not a transaction, since the intermediated graph G'_1 is not unstable and the initial graph G'_0 is not stable.

T-GTS morphisms. A T-GTS *morphism* between two T-GTSs is a GTS morphism between their underlying GTSs, such that the type component preserves the stable and unstable items and it is total on unstable ones.

Definition 3.7 (T-GTS morphism) *Let $\mathcal{Z}_1 = \langle \mathcal{G}_1, T_{1s} \rangle$ and $\mathcal{Z}_2 = \langle \mathcal{G}_2, T_{2s} \rangle$ be* T-GTS*s. A* T-GTS morphism *$f\colon \mathcal{Z}_1 \to \mathcal{Z}_2$ is a* GTS *morphism $f\colon \mathcal{G}_1 \to \mathcal{G}_2$ between the underlying* GTS*s, such that*

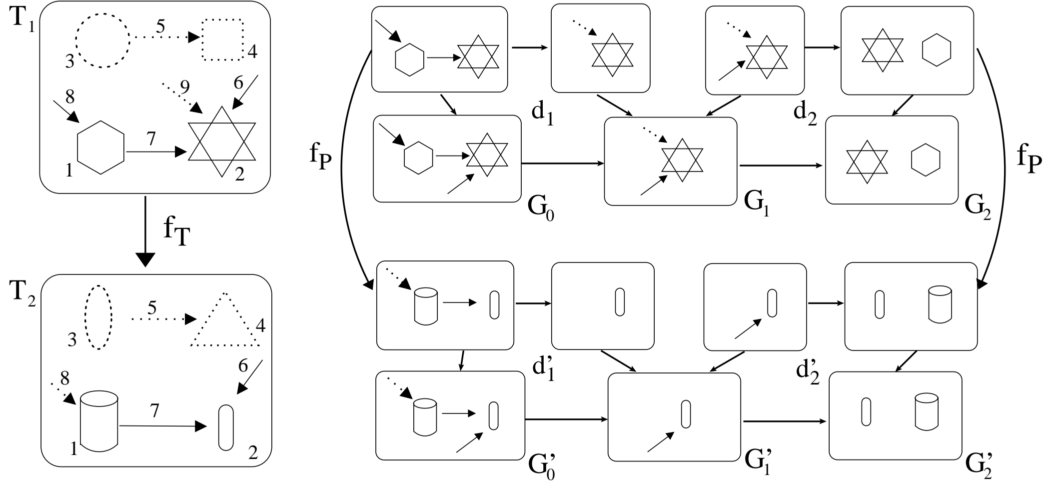


Figure 3.5: GTS morphism f does not define a T-GTS morphism.

1. *for all $z \in T_1 \setminus T_{1s}$, we have that $f_T(z)$ is defined and $f_T(z) \in T_2 \setminus T_{2s}$;*

2. *for all $z \in T_{1s}$, if $f_T(z)$ is defined then $f_T(z) \in T_{2s}$.*

Proposition 3.2 T-GTS*s and* T-GTS *morphisms form a category, denoted by* $\mathbf{TGTS}$*, in which composition and identities are defined as in* $\mathbf{GTS}$*.*

Proof:

1. *Composition is well-defined*: Let $f : \mathcal{Z}_1 \to \mathcal{Z}_2$ and $g : \mathcal{Z}_2 \to \mathcal{Z}_3$ be T-GTS morphisms and $g \circ f$ be their composition. By definition of T-GTS morphism $\forall z_1 \in T_1 \setminus T_{1s} \centerdot f_T(z_1) = z_2 \in T_2 \setminus T_{2s}$; $\forall z_2 \in T_2 \setminus T_{2s} \centerdot g_T(z_2) = z_3 \in T_3 \setminus T_{3s}$; $\forall z_1 \in T_{1s} \centerdot f_T(z_1) = z_2 \in T_{2s} \vee f_T(z_1) = \text{undefined}$; and $\forall z_2 \in T_{2s} \centerdot g_T(z_2) = z_3 \in T_{3s} \vee f_T(z_2) = \text{undefined}$. Therefore, by composition of graph morphisms:

   (a) $\forall z_1 \in T_1 \setminus T_{1s}$ we have that $g_T(f_T(z_1)) = z_3$, with $z_3 \in T_3 \setminus T_{3s}$;

   (b) $\forall z_1 \in T_{1s}$ if $f_T(z_1)$ and $g_T(z_2)$ are defined, then $g_T(f_T(z_1)) = z_3 \in T_{3s}$, else $g_T(f_T(z_1)) = \text{undefined}$.

2. *Identities are well-defined morphisms*: Let $id_{\mathcal{Z}} = \langle id_T, id_P \rangle$ be the identity on $\mathcal{Z} = \langle \langle T, P, \pi \rangle, T_s \rangle$. By definition of identity on $\mathbf{GTS}$, we have that $\forall z \in T \centerdot id_T(z) = z$, therefore:

   (a) $\forall z \in T \setminus T_s \centerdot id_T(z) \in T \setminus T_s$;

   (b) $\forall z \in T_s \centerdot id_T(z) \in T_s$.

3. Neutrality of identity and associativity of composition follow from these properties in $\mathbf{GTS}$.

$\square$

In order to ensure that morphisms are simulations in this more general framework, we prove that T-GTS morphisms preserve abstract transactions, i.e., if two T-GTSs are related by a T-GTS morphism, each transaction in the source T-GTS when retyped using the retyping functor induced by the morphism between the type graphs gives raise to a transaction in the target T-GTS. The following proposition describes this property.

Proposition 3.3 (T-GTS morphisms preserve transactions) *Let $f\colon \mathcal{Z}_1 \to \mathcal{Z}_2$ be a* T-GTS *morphism and let $[\rho]_a$ be an abstract transaction in $\mathcal{Z}_1$. Then $[f_T^{\leftrightarrow}(\rho)]_a$ is an abstract transaction in $\mathcal{Z}_2$.*

Proof: Assume that the derivation ρ in $\mathcal{Z}_1$ has the following shape:

$$\begin{array}{ccccccccccc} L_{p_1} \hookleftarrow K_{p_1} \to R_{p_1} & & L_{p_2} \hookleftarrow K_{p_2} \to R_{p_2} & & \cdots & & L_{p_n} \hookleftarrow K_{p_n} \to R_{p_n} \\ \downarrow \quad\; \downarrow \qquad \searrow & \swarrow & \downarrow \qquad \searrow & & & \swarrow & \downarrow \qquad \downarrow \\ G_0 \hookleftarrow D_1 \longrightarrow G_1 & \hookleftarrow & D_2 \longrightarrow G_2 & \cdots & G_n & \hookleftarrow & D_n \longrightarrow G_n \end{array}$$

By 2.3, we have that $f_T^{\leftrightarrow}(\rho)$ is a derivation in $\mathcal{Z}_2$. Let us consider H_i be the graph obtained from G_i, by applying the pullback functor. We have to prove that:

1. *H_0 and H_n are stable graphs*: we have to show that $\mathcal{S}(H_0) \approx H_0$ and $\mathcal{S}(H_n) \approx H_n$. As H_0 is obtained from G_0, by applying the pullback functor, the square (1), in diagram below, is a pullback. Since f_T preserves the stable items, (4) and (5) are pullbacks, as well. Moreover, by definition of $\mathcal{S}$, (2) and (3) are pullbacks.

$$\begin{array}{ccccccc} \mathcal{S}(H_0) & \xrightarrow{\iota_{\mathcal{S}(H_0)}} \; (\xleftarrow{!t}) & H_0 & \xrightarrow{f} & G_0 & \xleftarrow{\iota_{G_0}} \; (\xrightarrow{\iota'_{G_0}}) & \mathcal{S}(G_0) \\ & & \downarrow g \;\; (!s) & (1) & \downarrow i & & \\ \downarrow k & (2) \quad T_2 & \xleftarrow{l} dom(f_T) & \xrightarrow{h} & T_1 \quad (3) & & \downarrow j \\ & \nearrow \iota_{T_2} & (4) \quad \uparrow o & (5) & & \nwarrow \iota_{T_1} & \\ T_{2s} & \xleftarrow{m} & X & \xrightarrow{n} & & & T_{1s} \end{array}$$

Since G_0 is stable, i.e. $\mathcal{S}(G_0) \approx G_0$, then there exists $\iota'_{G_0} : G_0 \to \mathcal{S}(G_0)$ such that $\iota'_{G_0} \circ \iota_{G_0} = id_{\mathcal{S}(G_0)}$ and $\iota_{G_0} \circ \iota'_{G_0} = id_{G_0}$. Moreover, we have:

   a. by universal property of pullback (5), $\exists! s : H_0 \to X$, such that:

      i. $o \circ s = g$ and
      ii. $n \circ s = j \circ \iota'_{G_0} \circ f$.

   b. by (a.i), $l \circ g = l \circ o \circ s$ and by commutativity of (4), $l \circ g = \iota_{T_2} \circ m \circ s$;

   c. by universal property of pullback (2), $\exists! t : H_0 \to \mathcal{S}(H_0)$, such that:

      i. $\iota_{\mathcal{S}(H_0)} \circ t = id_{H_0}$ and
      ii. $k \circ t = m \circ s$.

   Since $\iota_{\mathcal{S}(H_0)}$ is surjective (by $(c.i)$) and injective (because of (2) is a pullback and ι_{T_2} is injective), then it is an isomorphism. Therefore, we have $\mathcal{S}(H_0) \approx H_0$. By symmetry we also have $\mathcal{S}(H_n) \approx H_n$, as we want to prove.

2. *$\mathcal{S}(\delta') = \mathcal{S}(H_0) \overset{p',m'}{\Longrightarrow} \mathcal{S}(H_n)$ is a derivation in $\mathcal{S}(\mathcal{Z}_2)$.* Let $\mathcal{S}(\delta) = \mathcal{S}(G_0) \overset{p,m}{\Longrightarrow} \mathcal{S}(G_n)$ be the derivation via $p : L \hookleftarrow K \to R$, the proper quotient production of $p_1 + \ldots + p_n$. The derivation $\mathcal{S}(\rho)$ is depicted by solid lines in Figure 3.6 and $\mathcal{S}(\delta)$ is depicted by dashed ones. The dotted part of represents the construction of proper quotient production p. Note that, for the sake of clearness, we will consider $n = 2$, but the proof can be trivially extended for derivation with any length. Moreover, as all graphs in the diagram is stabilised we omitted the "$\mathcal{S}$", e.g., we use G_0 to refer $\mathcal{S}(G_0)$.

   By definition of proper quotient production, we have that the squares (6) and $\langle X, R_{p_1}, L_{p_2}, G_1\rangle$ are a pullbacks, and the squares (1), (2), (3), (4), (5), $\langle D_1^*, L, D_1, G_0\rangle$, $\langle D_1^*, D^*, D_1, G_1\rangle$, $\langle D_2^*, D^*, D_2, G_1\rangle$, $\langle D_2^*, R, D_2, G_2\rangle$, $\langle K_{p_2}, L_{p_2}, K, D_1^*\rangle$, $\langle K_{p_1}, R_{p_1}, K, D_2^*\rangle$ and

$\langle X, K_{p_1}, K_{p_2}, K\rangle$ are pushouts. By definition of transactions $\mathcal{S}(\rho) \approx \mathcal{S}(\delta)$, then we have that the squares (7) and (8) are pushouts and (9) is a pullback. Moreover, since $\langle K, L, D, G_0\rangle$ and $\langle D_1^*, L, D_1, G_0\rangle$ are pushouts then $\langle K, D_1^*, D, D_1\rangle$ is a pushout (by Lemma A.4). By symmetry, $\langle K, D_2^*, D, D_2\rangle$ is a pushout, as well. Since f_T preserves the

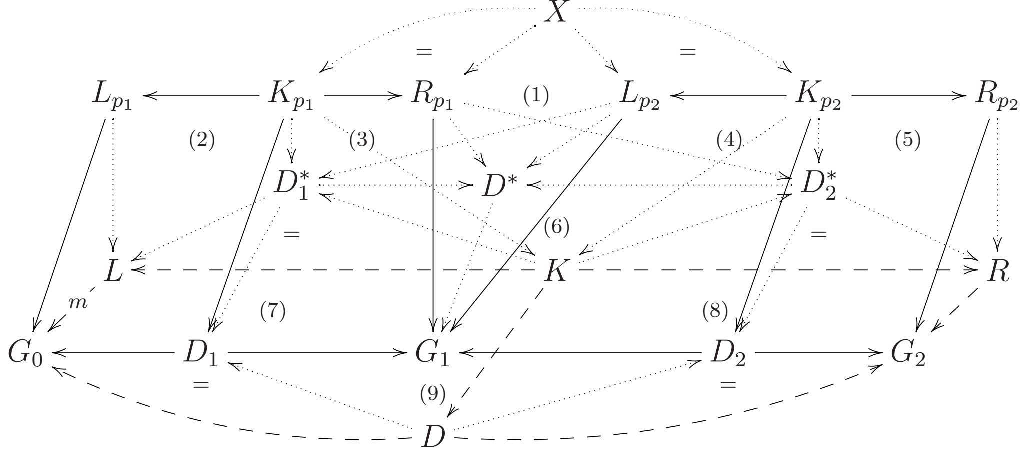
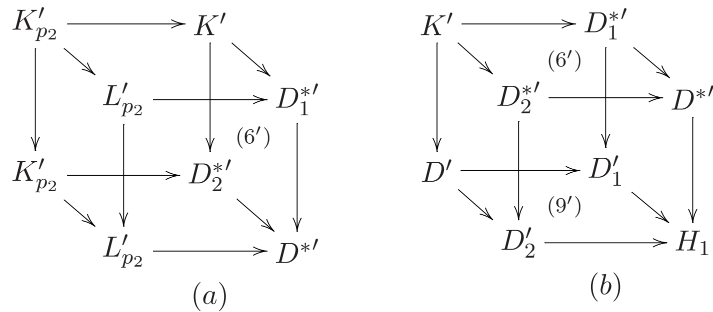


Figure 3.6: Derivation $\mathcal{S}(\rho)$ and its equivalent derivation via proper quotient production.

stable and unstable items and by propositions 2.3 and 3.1, we have that $f_T^{\leftrightarrow}(\mathcal{S}(\rho)) = \mathcal{S}(\rho')$ and $f_T^{\leftrightarrow}(\mathcal{S}(\delta)) = \mathcal{S}(\delta')$ are derivations in $\mathcal{S}(\mathcal{Z}_2)$. Therefore, it remains to prove that $p' : L' \hookleftarrow K' \rightarrow R'$ is the proper quotient of $f_P(p_1) + \ldots + f_P(p_2)$ and that $\rho' \approx \delta'$.

Figure 3.7 depicts the derivations ρ' and δ' obtained from ρ and δ', applying the pullback functor on each graph . Thus we have that the squares $\langle K_1', L_1', D_1', H_0\rangle$, $\langle K_1', R_1', D_1', H_1\rangle$, $\langle K_2', L_2', D_2', H_1\rangle$, $\langle K_2', R_2', D_2', H_2\rangle$, $\langle K', L', D', H_0\rangle$ and $\langle K', R', D', H_2\rangle$ are pushouts. Then we must prove that:

- the squares (1′), (2′), (3′), (4′), (5′), $\langle D_1^{*\prime}, L', D_1', H_0\rangle$, $\langle D_1^{*\prime}, D^{*\prime}, D_1', H_1\rangle$, $\langle D_2^{*\prime}, D^{*\prime}, D_2', H_1\rangle$, $\langle D_2^{*\prime}, R', D_2', H_2\rangle$, $\langle K_{p_2}', L_{p_2}', K', D_1^{*\prime}\rangle$, $\langle K_{p_1}', R_{p_1}', K', D_2^{*\prime}\rangle$, $\langle X', K_{p_1}', K_{p_2}', K'\rangle$, $\langle K, D_1^*, D, D_1\rangle$ and $\langle K, D_2^*, D, D_2\rangle$ are pushouts. It holds by Lemma 2.1;
- the squares (6′) and (9′) are pullbacks. In the following diagram (a) we have that the top and bottom squares are pushouts and the back and left squares are pullbacks (by Lemma A.1). Then, by Lemma A.6, (6′) is a pullback.



In the diagram (b) above we have that the top square is a pullback and the back, left, front and right are pushouts. Then, by Lemma A.6, (9′) is a pullback.

- the diagrams (=), in Figure 3.7 commute: it holds by Lemma 2.2.

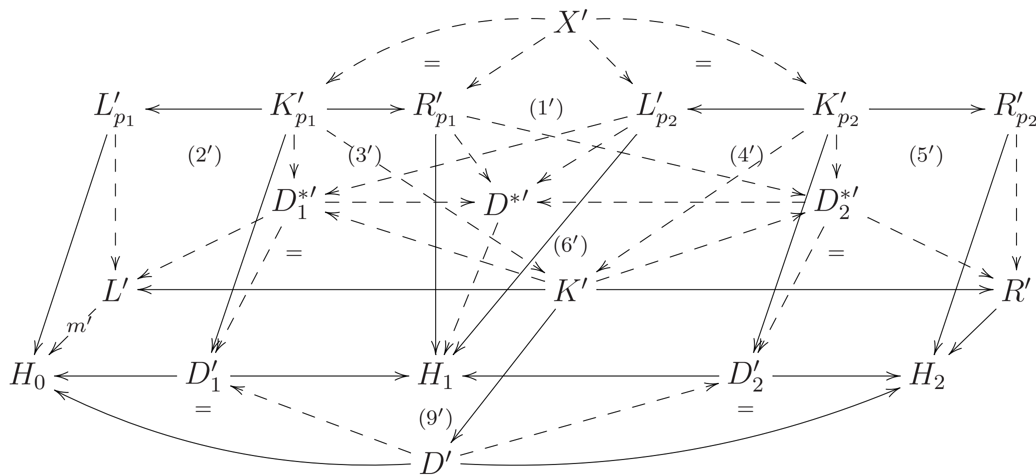
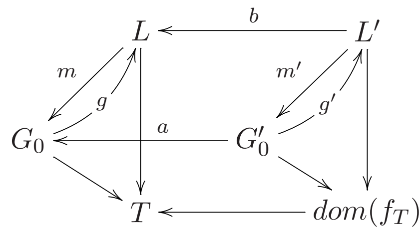


Figure 3.7: Derivations $f_T^{\leftrightarrow}(\rho)$ and $f_T^{\leftrightarrow}(\delta)$.

3. *the match m' is an isomorphism*: considering the Figure 3.6, since $m : L \to G_0$ is an isomorphism, there exists an morphism $g : G_0 \to L$ as depicted in diagram below, such that $g \circ m = id_L$ and $m \circ g = id_{G_0}$.



G'_0 and L' are obtained from G_0 and L, respectively, by applying the pullback functor. We must prove that $g' \circ m' = id_{L'}$ and $m' \circ g' = id_{G'_0}$. By commuting diagram above, we have that:

(1) $m \circ g \circ a = id_{G_0} \circ a$;

(2) $m \circ b \circ g' = a$;

(3) $a \circ m' \circ g' = a$;

(4) $a \circ m' \circ g' = a \circ id_{G'_0}$, by property of identity;

(5) $m' \circ g' = id_{G'_0}$, since a is injective; and

(6) $g \circ m \circ b = id_L \circ b$;

(7) $g \circ a \circ m' = b$;

(8) $b \circ g' \circ m' = b$;

(9) $b \circ g' \circ m' = b \circ id_{L'}$, by property of identity;

(10) $g' \circ m' = id_{L'}$, since b is injective.

Therefore, by (5) and (10), m' is an isomorphism, as well.

4. *any intermediate graph H_i ($i \neq 0, n$) is not stable*: we have to prove that $\exists x \in H_i$.

$r_{f_T}(d(x)) \in T_2 \setminus T_{2s}$. Considering the diagram below we have:

$$\begin{array}{ccccc} G_i & \xleftarrow{h} & H_i & & \\ {\scriptstyle g}\downarrow & PB & \downarrow{\scriptstyle d} & \searrow & \\ T_1 & \xleftarrow[l_{f_T}]{} & dom(f_T) & \xrightarrow[r_{f_T}]{} & T_2 \end{array}$$

(a) By definition of T-GTS morphisms:

i. f_T is total on unstable items (l_{f_T} is surjective on unstable items): $\forall x \in T_1 \setminus T_{1s} \centerdot x \in dom(f_T) \setminus \mathcal{S}(dom(f_T))$;

ii. f_T preserves unstable items: $\forall x \in dom(f_T) \setminus \mathcal{S}(dom(f_T)) \centerdot r_{f_T}(x) \in T_2 \setminus T_{2s}$

(b) By definition of $\mathcal{S}$ and graph morphisms: $\forall y \in G_i \setminus \mathcal{S}(G_i) \centerdot \exists x \in T_1 \setminus T_{1s} \centerdot g(y) = x$;

(c) By (a.i) and (b): $\forall y \in G_i \setminus \mathcal{S}(G_i) \centerdot \exists x \in dom(f_T) \setminus \mathcal{S}(dom(f_T)) \wedge g(y) = x$;

(d) By definition of pullbacks in **Graph**:

i. Since l_{f_T} is surjective on unstable (a.i), then $\forall y \in G_i \setminus \mathcal{S}(G_i) \centerdot \exists z \in H_i \centerdot h(z) = y \wedge g(h(z)) = l_{f_T}(d(z))$;

ii. By (d.i) and since l_{f_T} is an inclusion: $\forall y \in G_i \setminus \mathcal{S}(G_i), z \in H_i \centerdot h(z) = y \centerdot g(h(z)) = d(z)$.

(e) By definition of transactions: $\exists y \in G_i \setminus \mathcal{S}(G_i)$;

(f) By (d.i) and (e): $\exists y \in G_i \setminus \mathcal{S}(G_i) \centerdot \exists z \in H_i \centerdot h(z) = y$;

(g) By (c) and (f): $\exists y \in G_i \setminus \mathcal{S}(G_i) \centerdot \exists z \in H_i \centerdot \exists x \in dom(f_T) \setminus \mathcal{S}(dom(f_T)) \centerdot h(z) = y \wedge g(y) = x$;

(h) By (d.ii) and (g): $\exists \in H_i \centerdot \exists x \in dom(f_T) \setminus \mathcal{S}(dom(f_T)) \centerdot g(h(z)) = x = d(z)$;

(i) By (a.ii) and (h): $\exists z \in H_i \centerdot d(z) \in dom(f_T) \setminus \mathcal{S}(dom(f_T)) \wedge r_{f_T}(d(z)) \in T_2 \setminus T_{2s}$

Therefore, by (i), $\exists z \in H_i \centerdot r_{f_T}(d(z)) \in T_2 \setminus T_{2s}$.

$\square$

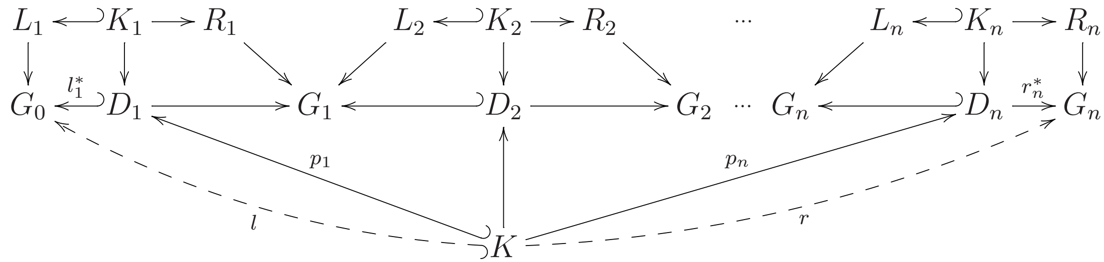
3.3 Abstract GTS associated to a T-GTS

As mentioned in the introduction, a T-GTS can be seen at two different levels of abstraction. It can be viewed as a standard graph transformation system, where both stable and unstable items of states are visible. But we can also abstract away from the unstable states and observe only complete transactions. Formally, this gives rise to another GTS, where the productions are all transactions of the original T-GTS. This definition requires the notion of the production induced by a derivation sequence, a known construction in the literature (CORRADINI et al., 1997).

Induced production of a derivation. The *production induced* by a derivation $\rho : G_0 \Rightarrow^* G_n$ has G_0 as left-hand side and G_n as right-hand side. The interface graph is the subgraph of G_0 which, intuitively, consists of all the items which are preserved by all the direct derivations occurring in the sequence.

Definition 3.8 (Induced production of a derivation) *Given a derivation* $\rho : G_0 \stackrel{p_1,m_1}{\Rightarrow} \cdots \stackrel{p_n,m_n}{\Rightarrow} G_n$ *for a* T-GTS $\mathcal{Z}$, *the* induced production of ρ *is* $G_0 \stackrel{l}{\hookleftarrow} K \stackrel{r}{\rightarrow} G_n$, *where* K *is*

the limit object obtained as in diagram below, $l = l_1^* \circ p_1$ *and* $r = r_n^* \circ p_n$.



Without loss of generality, we will assume a concrete choice for K*, by imposing that the morphism* p_1 *in the limit diagram above is an inclusion.*

Example 3.3 (Induced production of a derivation) *Figure 3.8 shows the abstract production associated to the transaction in Figure 3.4.*

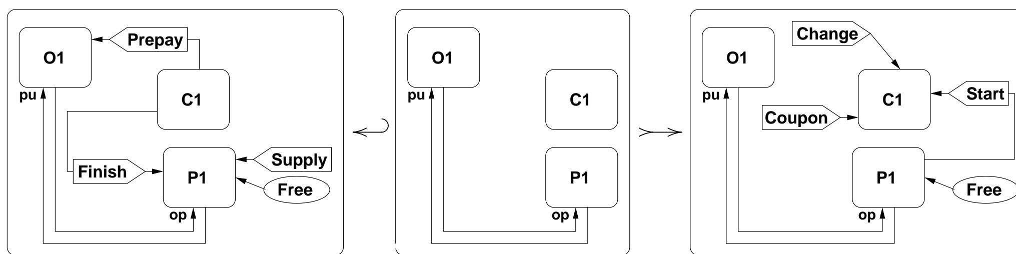


Figure 3.8: Abstract production associated to the transaction in Figure 3.4.

┘

Abstract GTS. The *abstract* GTS associated to the given T-GTS $\mathcal{Z}$, denoted by $A(\mathcal{Z})$, is a GTS where the type graph is the stable type graph of $\mathcal{Z}$, the set of production names contains all abstract stable transactions of $\mathcal{Z}$ and each abstract stable transaction is associated to the production induced by it.

Definition 3.9 (Abstract GTS) *Let* $\mathcal{Z} = \langle \mathcal{G}, T_s \rangle$ *be a* T-GTS*. Given an abstract stable transactions* $[\rho]_a$*, a production induced by* ρ *is called* abstract production *for the transaction* $[\rho]_a$*. The* abstract GTS *associated to the given* T-GTS*, denoted by* $A_{\mathcal{Z}}$*, is the* GTS $\langle T_s, P', \pi' \rangle$ *where* P' *is the set of abstract stable transactions* $[\rho]_a$ *and* $\pi'([\rho]_a)$ *is an abstract production for the transaction* $[\rho]_a$.

3.4 Transactions as graph processes

Inspired by the classical *non-sequential processes* for Petri nets (GOLTZ; REISIG, 1983), *graph processes* have been proposed in (CORRADINI; MONTANARI; ROSSI, 1996; BALDAN; CORRADINI; MONTANARI, 1998a) as a faithful representation of the derivations of a GTS up to shift-equivalence. Since abstract transactions are defined as abstract traces we will introduce an equivalent, and yet more manageable, definition of transaction based on graph processes – this presentation is used to provide a universal characterisation for the class of transactions of a given T-GTS.

A *graph process* for a T-GTS $\mathcal{Z}$ is defined as an “occurrence grammar” $\mathcal{O}$, i.e., a grammar satisfying suitable acyclicity constraints, equipped with a T-GTS morphism from $\mathcal{O}$ to $\mathcal{Z}$. This definition is given in terms of structural properties, but we can use a more concrete definition where a graph process for $\mathcal{Z}$ is constructed from a derivation in $\mathcal{Z}$ by means of an explicitly colimit construction.

An occurrence grammar is a graph grammar (a T-GTS with start graph) that represents a class of derivations, so, it contains the items used in the derivation and the applied productions. Each item in the type graph of this kind of grammar represents an occurrence of this item in the derivation and can be created, consumed or preserved by its productions. Therefore, an occurrence grammar must satisfy some restrictions: all reachable graphs from the start graph, applying some sequence of productions, have an injective typing morphism; the type graph items and the productions have an acyclic dependence among them; the start graph coincides with the graph obtained restricting the type graph to the items that are not created by any production; all productions can be applied; and each type graph item is created or deleted at most once.

For any graph process, we can obtain a derivation applying all productions exactly once, in any compatible order, from its initial graph. This allows to realise that a process defined as above can be seen as a representative of a class of derivations of the original T-GTS, where only the independent steps may be switched, that is, an abstract trace. Thus, we can construct a graph process from a trace: since a process represents a class of derivations, any derivation in this class can be chosen to construct the equivalent process.

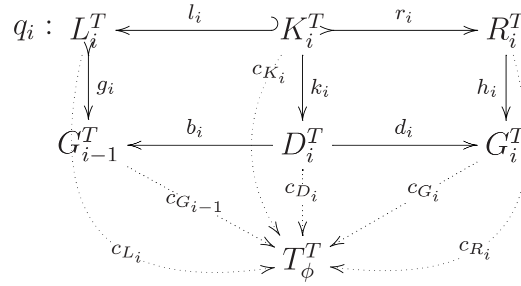
3.4.1 Graph Processes

In this subsection, we review some definitions about graph processes (CORRADINI; MONTANARI; ROSSI, 1996; BALDAN, 2000), (BALDAN; CORRADINI; MONTANARI, 1998a).

Graph process from a derivation. Intuitively, the colimit construction applied to a derivation of $\mathcal{Z}$ essentially constructs the type graph as a copy of the source graph plus the items created during the rewriting process. Productions are instances of production applications and are related to the original ones by means of identities in the category of typed graphs over T (the type graph of $\mathcal{Z}$).

Definition 3.10 (process from a derivation) *Let $\mathcal{Z} = \langle\langle T, P, \pi\rangle, T_s\rangle$ be a* T-GTS*, and let $\rho = G_0 \overset{q_1,m_1}{\Rightarrow} G_1 \overset{q_2,m_2}{\Rightarrow} \ldots \overset{q_n,m_n}{\Rightarrow} G_n$ be a derivation in $\mathcal{Z}$. A* process *ϕ* associated to *ρ is a* T-GTS *morphism $\phi = \langle \phi_T, \phi_P\rangle \colon \mathcal{O}_\phi \to \mathcal{Z}$, where $\mathcal{O}_\phi = \langle\langle T_\phi, P_\phi, \pi_\phi\rangle, T_{\phi_s}\rangle$ is obtained as follows*

- *$T_\phi^T = \langle T_\phi, t_{T_\phi}\rangle$ is a colimit object (in T-**Graph**) of the diagram representing derivation ρ, as depicted (for a single derivation step) in the diagram below, where $c_{X_i} \colon X_i^T \to T_\phi^T$ is the induced injection for $X \in \{D, G, L, K, R\}$;*
- *$T_{\phi_s} \hookrightarrow T_\phi = t^*_{T_\phi}(T_s \hookrightarrow T)$;*
- *$P_\phi = \{\langle q_i, i\rangle \mid i \in \{1, \ldots, n\}\}$;*
- *$\pi_\phi(\langle q_i, i\rangle) = \langle L_i, c_{L_i}\rangle \overset{l_i}{\leftarrow} \langle K_i, c_{K_i}\rangle \overset{r_i}{\rightarrow} \langle R_i, c_{R_i}\rangle$ (see the diagram below);*
- *$\phi_T = t_{T_\phi}$*
- *$\phi_P(\langle q_i, i\rangle) = q_i$, for all $i \in \{1, \ldots, n\}$.*



Since in a derivation all matches are assumed to be injective, in the associated process all productions are injectively typed, so we can say that a production consumes, preserves or creates elements of the type graph. Considering the diagram above, if $x \in T_\phi^T$ and $q = \langle q_i, i\rangle$, we say that the production q *consumes* x if x is in the image of c_{L_i} and not in that of c_{K_i}; that q *creates* x if x is in the image of c_{R_i} and not in that of c_{K_i}; and that q *preserves* x if it is in the image of c_{K_i}. This leads to the following net-like notation

$$^\bullet q = c_{L_i}(L_i \setminus l_i(K_i)) \qquad q^\bullet = c_{R_i}(R_i \setminus r_i(K_i)) \qquad \underline{q} = c_{K_i}(K_i)$$

We say that q *consumes*, *creates* and *preserves* items in $^\bullet q$, $q^\bullet$ and $\underline{q}$, respectively. Similarly, the sets of productions which consume, create and preserve $x \in T_\phi$ are denoted by $^\bullet x$, $x^\bullet$ and $\underline{x}$, respectively.

Minimal and maximal graphs. Considering the sets defined above, we can obtain the minimal and the maximal graphs of a graph process, i.e, the initial and final graphs of the derivation associated to it. The minimal graph is a subgraph of the type graph whose items are not created by any production of the process and the maximal is a subgraph of the type graph whose items are not consumed by any production.

Definition 3.11 (minimal and maximal graphs) *Let $\phi : \langle\langle T_\phi, P_\phi, \pi_\phi\rangle, T_{\phi_s}\rangle \to \mathcal{G}$ be a process of a* GTS *$\mathcal{G}$. The* minimal graph *of ϕ, denoted by $Min(\mathcal{O}_\phi)$, is the subgraph of T_ϕ consisting of the items x such that $^\bullet x = \varnothing$, and $^\bullet \phi$ is the same graph typed over T by the restriction of ϕ_T. The maximal graph of ϕ, denoted by $Max(\mathcal{O}_\phi)$, is the subgraph of T_phi consisting of the items x such that $x^\bullet = \varnothing$, and $\phi^\bullet$ is the same graph typed over T.*

Example 3.4 (graph process) *As an example, in Figure 3.9 (top-left), we show the type graph of the process ϕ_1 associated to the derivation of Figure 3.4 – it was constructed as a gluing of all graphs in the derivation. The injections from the graphs of the derivation are implicitly represented by using an index pair for the items with a* creation index *on the first position of the pair, and a* deletion index *on the last one. The creation index is missing $(-)$ in the items that are not created, i.e., that belong to the start graph, and symmetrically for the deletion index. The image of graph G_i of the derivation, with $i \in \{0, \ldots, 6\}$, contains all items with creation index, if any, smaller than i, and deletion index, if any, larger than or equal to i. The minimal and the maximal graphs of this process are shown at the bottom-left side of the figure. The productions (see Figure 3.9 (right)) of the process are all productions in Figure 3.4 typed over the process type graph.*

┘

If we consider two productions in a graph process, we can have different situations concerning their application: if they do not use items that are common to both or only preserve them, these productions can be applied in parallel, but if one of the productions

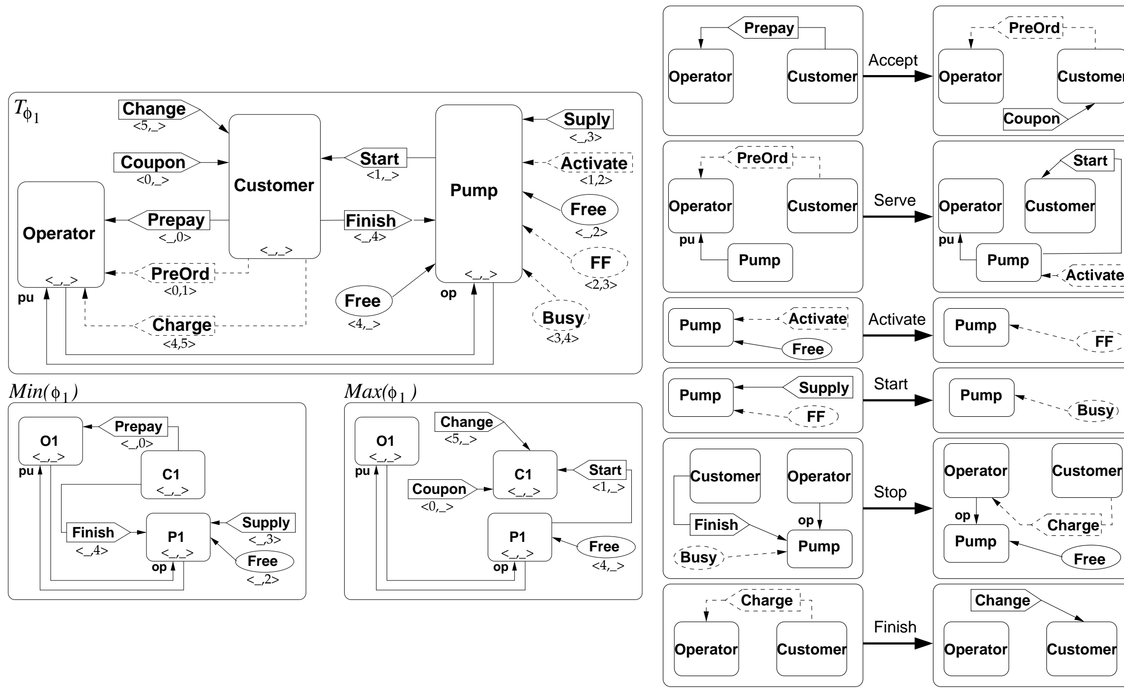


Figure 3.9: Type graph (top-left), the maximal and the minimal graphs (bottom-left), and productions (right) of the process associated to derivation in Figure 3.4.

consumes (or preserves) the same item that is preserved (or created) by the other, the latter must be applied before the former. Thus we can define a causal relation between all elements (items of the type graph and productions) of a graph process, considering the relation described above.

Definition 3.12 (causal relation) *The* causal relation *of a process ϕ is the least transitive and reflexive relation $\leq_\phi$ over $T_\phi \uplus P_\phi$ such that for all $x, y \in T_\phi \uplus P_\phi$ and $q_1, q_2 \in P_\phi$:*

i) $x \leq_\phi y$ *if* $x \in {}^\bullet y$ *and*

ii) $q_1 \leq_\phi q_2$ *if* $((q_1{}^\bullet \cap \underline{q_2}) \cup (\underline{q_1} \cap {}^\bullet q_2)) \neq \varnothing$.

Reachable sets. Reachable sets of a process are subsets of elements of type graph of these processes. Each of them represents a graph reachable from minimal graph, applying a subset of productions of a process. A reachable set is obtained based on the causal relation and contains all items that are created/preserved and are not consumed by considered productions.

Definition 3.13 (Reachable sets) *Let ϕ be a process. For any $\leq_\phi$-left-closed $P' \subseteq P_\phi$, the reachable set associated to P' is the set $S_{P'} \subseteq T_\phi$ defined by*

$$x \in S_{P'} \text{ iff } \forall q \in P_\phi \,.\, (x \leq_\phi q \Rightarrow q \notin P') \wedge (q \leq_\phi x \Rightarrow q \in P').$$

Graph processes which have the same structure but, for example, different identities of vertices, edges and productions should be considered as being equivalent.

Abstract process. An abstract process is a class of isomorphic processes of a T-GTS. Two processes of a T-GTS $\mathcal{Z}$ are isomorphic if their type graphs are isomorphic (they

have the same structure) and their productions are instances of the same productions of $\mathcal{Z}$. Since each item in the type graph of a process represents an occurrence of this item, the typing morphisms of productions indicate, for each item, the productions that created, consumed and/or preserved it. Therefore the isomorphism between the type graphs must be compatible with the typing of productions.

Definition 3.14 (Abstract process) *Let $\langle \mathcal{O}_{\phi_1}, \phi_1 \rangle$ and $\langle \mathcal{O}_{\phi_2}, \phi_2 \rangle$ be two graph processes of a* GTS *$\mathcal{G}$. Then ϕ_1 and ϕ_2 are isomorphic if only if there exists a pair $\langle f_T, f_P \rangle$, where:*

- *$f_T : \langle T_{\phi_1}, \phi_{1T} \rangle \to \langle T_{\phi_2}, \phi_{2T} \rangle$ is an isomorphism in T-**Graph***
- *$f_P : P_{\phi_1} \to P_{\phi_2}$ is a bijection, such that $\phi_{1P} = \phi_{2P} \circ f_P$*
- *for each $q_1 : L_{q1}^{T_{\phi_1}} \overset{l_1}{\leftarrow} K_{q1}^{T_{\phi_1}} \overset{r_1}{\to} R_{q1}^{T_{\phi_1}} \in P_{\phi_1}$, $q_2 = f_P(q_1) : L_{q2}^{T_{\phi_2}} \overset{l_2}{\leftarrow} K_{q2}^{T_{\phi_2}} \overset{r_2}{\to} R_{q2}^{T_{\phi_2}} \in P_{\phi_2}$, if $q = \phi_{1P}(q_1) = \phi_{2P}(q_2) : L_q^T \overset{l}{\leftarrow} K_q^T \overset{r}{\to} R_q^T \in P$, the diagram in Figure 3.10 commutes.*

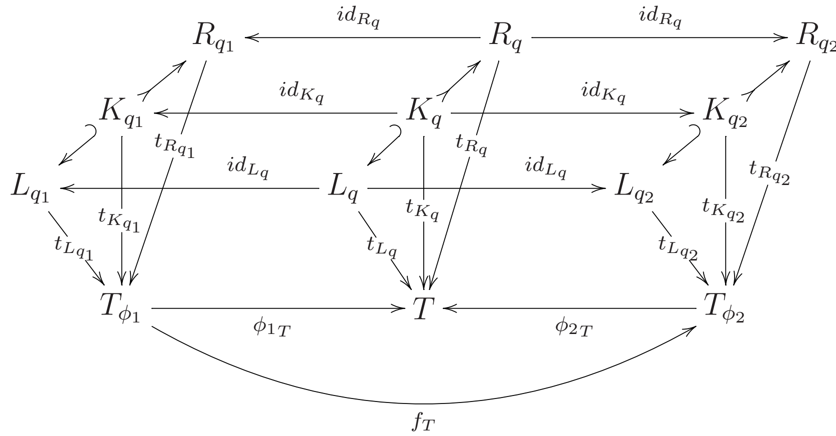


Figure 3.10: Isomorphism of processes.

If two processes ϕ_1 and ϕ_2 are isomorphic, then we will write $\phi_1 \cong \phi_2$. Moreover, given any process ϕ, we will write $[\phi]$ to denote $\{\phi' | \phi' \cong \phi\}$, and we will call $[\phi]$ an abstract process.

From the theory of graph processes (see (BALDAN; CORRADINI; MONTANARI, 1998a)) we know that the abstract processes of a T-GTS $\mathcal{Z}$ are in one-to-one correspondence with the abstract traces of $\mathcal{Z}$. More precisely, if $[\rho]_a$ is an abstract trace of $\mathcal{Z}$ and $\rho', \rho'' \in [\rho]_a$ are two derivations, then the processes associated to ρ' and ρ'' are isomorphic. This defines a function $\mathcal{TP}_{\mathcal{Z}}$ mapping the abstract traces of $\mathcal{Z}$ to abstract processes for $\mathcal{Z}$. Vice versa, if ϕ is a process for $\mathcal{Z}$, and ρ, ρ' are two derivations of $\mathcal{O}_\phi$, then the retyped derivations $\phi_T^{\leftrightarrows}(\rho)$ and $\phi_T^{\leftrightarrows}(\rho')$ of $\mathcal{Z}$ (see the observation after Proposition 2.2) are abstract truly-concurrent equivalent, and thus belong to the same abstract trace. This defines a function $\mathcal{PT}_{\mathcal{Z}}$ mapping the abstract processes for $\mathcal{Z}$ to abstract traces of $\mathcal{Z}$. Moreover, it can be proved that functions $\mathcal{TP}_{\mathcal{Z}}$ and $\mathcal{PT}_{\mathcal{Z}}$ are inverse to each other. By Proposition 3.4 an isomorphism between abstract traces and abstract processes is established: hence, the latter provides an alternative, equivalent characterisation of the former ones.

Proposition 3.4 *Let $\mathcal{Z}$ be a* T-GTS. *Then $[\phi]$ is an abstract t-process of $\mathcal{Z}$ iff $\mathcal{PT}_{\mathcal{Z}}([\phi])$ is an abstract transaction (see (BALDAN; CORRADINI; MONTANARI, 1998a)).*

Figure 3.11 shows an abstract representation of the relationship between derivation traces and graph processes: in the left, we have a class of shift-equivalent derivations containing three direct derivations (δ_1, δ_2 and δ_3) occurring in different order; and, in the right, we have the correspondent process, where the direct derivations are not ordered.

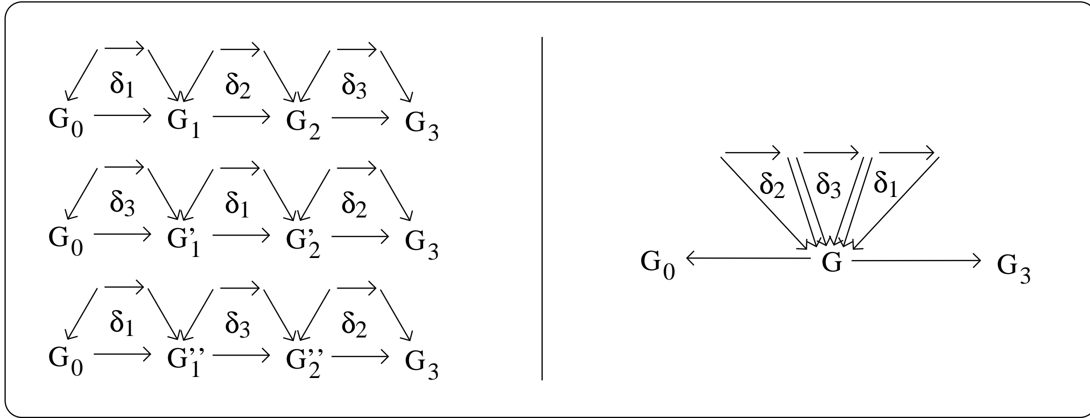


Figure 3.11: Class of shift-equivalent derivations (left) and the equivalent graph process (right)

3.4.2 Transactional Processes

Since we can build a process from any derivation, we can characterise the processes that can be obtained from transactions, these processes are called *transactional processes*.

Transactional process. A transactional process is a process having the minimal and the maximal graphs stable and the remaining reachable graphs are unstable; moreover, all items in the minimal graph must be used and the stable type items can be or consumed, or created, or preserved by the productions in the process.

Since transactional processes are used to define implementation morphisms, where productions are mapped into processes, we need to consider also a wider class of processes, the *unstable transactional processes*, which may start and end in unstable states. These processes are used to define implementations of unstable productions.

Definition 3.15 (transactional process) *Let $\mathcal{Z} = \langle\langle T, P, \pi\rangle, T_s\rangle$ be a* T-GTS. *An* unstable transactional process (ut-process) *is a process ϕ of $\mathcal{Z}$ such that*

1. *for any $x \in T_{\phi_s}$, at most one of the sets ${}^\bullet x$, $x^\bullet$, $\underline{x}$ is not empty;*
2. *for any $x \in Min(\mathcal{O}_\phi)$, there exists $q \in P_\phi$ such that either $x \in {}^\bullet q$ or $x \in \underline{q}$;*
3. *for any reachable set $S_{P'}$ associated to a non-empty $P' \subset P_\phi$, there exists $x \in S_{P'}$ such that $x \notin Min(\mathcal{O}_\phi) \cup Max(\mathcal{O}_\phi)$.*

If $Min(\mathcal{O}_\phi) \cup Max(\mathcal{O}_\phi) \subseteq T_{\phi_s}$, then ϕ is called transactional process (t-process). *The family of abstract ut-processes of $\mathcal{Z}$ is denoted by $\mathbf{utProc}(\mathcal{Z})$ and $\mathbf{tProc}(\mathcal{Z}) \subseteq \mathbf{utProc}(\mathcal{Z})$ denotes the class of all abstract t-processes of $\mathcal{Z}$.*

Note that if a representative of an abstract process is a(n unstable) transactional one, then all the other members of the equivalence class are transactional processes, as well. Condition 1 implies that each stable item is either in the source or in the target state of the process. Additionally, each stable item that is preserved by at least one production cannot be generated nor consumed in the process itself: this would induce a dependency between productions, violating the defining requirements for transactions (see Definition 3.5). By condition 2, any item in the source state is used in the computation. Condition 3 ensures that the process is not decomposable into "smaller pieces". It tells that by executing only an initial, non-empty subset P' of the productions of the process, we end up in a graph $S_{P'}$ which is not entirely contained in $Min(\mathcal{O}_\phi) \cup Max(\mathcal{O}_\phi)$, i.e., which contains at least one unstable item. Finally, in a transactional process the source and target states are required to be stable. For example, the process described in Example 3.4 is transactional.

3.4.3 Abstract GTS for a T-GTS based on process

Since the abstract transaction and the abstract t-processes of a T-GTS have an one-to-one relationship, we can redefine abstract GTSs associated to a T-GTSs using this more manageable representation for the transactions. This new definition will be used to characterise the operation that associates an abstract GTS to a T-GTS as an adjunction.

In this chapter, we consider a class of equivalent transactional processes which does not take in account the causal relation. Since an abstract process is defined as a class of isomorphic processes, two processes with same productions, isomorphic type, minimal and maximal graphs may not be in the same abstract process (class of equivalence) if there are two elements with the same type but with different causal relation. Therefore, we define a notion of equivalence of processes which is weaker than isomorphic process, allowing to consider the mentioned process as equivalent, called *weak equivalence*. Thus, we will use all classes of weak-equivalent processes to represent all transactions of a T-GTS. This definition allows us to characterise the abstraction operation as an adjunction: if we had used abstract processes (class of isomorphic processes) as transactions, this characterisation could not be established, because the causal relation is lost in abstraction operation.

Weak processes. Two abstract processes are weak-equivalent if there are isomorphisms between their type graphs, minimal graphs and maximal graphs, and they have the same instances of productions. Note that here, the condition on typing of productions is disregarded.

Definition 3.16 (Weak-equivalence, wut-processes) *Let ϕ_1 and ϕ_2 be two ut-processes. Then, ϕ_1 and ϕ_2 are* weak-equivalent*, written $\phi_1 \approx^w \phi_2$, if only if there exists a pair $\langle f_T, f_P \rangle$, where:*

- *$f_T : \langle T_{\phi_1}, \phi_{1T} \rangle \rightarrow \langle T_{\phi_2}, \phi_{2T} \rangle$ is an isomorphism in T-$\mathbf{Graph}$, such that $Min(\mathcal{O}_{\phi_1}) \approx Min(\mathcal{O}_{\phi_2})$ and $Max(\mathcal{O}_{\phi_1}) \approx Max(\mathcal{O}_{\phi_2})$*

- *$f_P : P_{\phi_1} \rightarrow P_{\phi_2}$ is a bijection, such that $\phi_{1P} = \phi_{2P} \circ f_P$*

A weak abstract ut-process (wut-process) *is defined as an equivalence class of ut-processes with respect to weak equivalence, denoted as $[\phi]_w$ for a representative ϕ. The set of wut-processes of a* T-GTS *$\mathcal{Z}$ is denoted by $\mathbf{wutProc}(\mathcal{Z})$. The set of* weak t-processes (wt-processes) *$\mathbf{wtProc}(\mathcal{Z})$ is defined in an analogous way.*

Underlying span. Like for the traces, we can associate to each transactional process a production, i.e., an *underlying span*, where the left- and right-hand side are the minimal and the maximal graphs of the process, respectively, and the interface is the intersection between these graphs. In order to associate a concrete span to an abstract process, we need to assume a chosen representative for any equivalence class of processes.

Definition 3.17 (span underlying abstract process) *Given a process ϕ for a* T-GTS $\mathcal{Z}$, *we have $\Pi(\phi) = {}^{\bullet}\phi \hookleftarrow {}^{\bullet}\phi \cap \phi^{\bullet} \hookrightarrow \phi^{\bullet}$ (intersection is taken component-wise).*
Let us assume for each T-GTS $\mathcal{Z}$ *a choice function $\mathsf{ch}_{\mathcal{Z}}$, mapping each wut-process $[\phi]_w$ to a concrete representative $\mathsf{ch}_{\mathcal{Z}}([\phi]_w) \in [\phi]_w$. The* underlying span *of a wut-process $[\phi]_w$ is defined as $\Pi_{\mathcal{Z}}([\phi]_w) = \Pi(\mathsf{ch}_{\mathcal{Z}}([\phi]_w))$.*

We are now able to define the abstract system associated with a T-GTS using transactional processes. This definition differs from that in Definition 3.9 to use the weak equivalence, therefore, some different transactions in Definition 3.9 can be the same in the following definition.

Abstract GTS. An *abstract system associated to a* T-GTS $\mathcal{Z}$ is a graph transformation system $A(\mathcal{Z})$ that have as type graph the stable type graph of $\mathcal{Z}$ and the set of production names contains all wt-processes of $\mathcal{Z}$, where each one is associated to its underlying production.

Definition 3.18 (Abstract GTS) *Let $\mathcal{Z} = \langle \mathcal{G}, T_s \rangle$ be a* T-GTS. *The* abstract GTS *associated to $\mathcal{Z}$, denoted by $A_{\mathcal{Z}}$, is the* GTS $\langle T_s, \mathbf{wtProc}(\mathcal{Z}), \Pi_{\mathcal{Z}} \rangle$ *where $\mathbf{wtProc}(\mathcal{Z})$ is the set of wt-processes of $\mathcal{Z}$ and $\Pi_{\mathcal{Z}}$ is as in Definition 3.17.*

Example 3.5 (Abstract GTS) *As an example, we can see in Figure 3.12 the abstract system associated to $\mathcal{P}umpOper$ (see Example 3.1).*

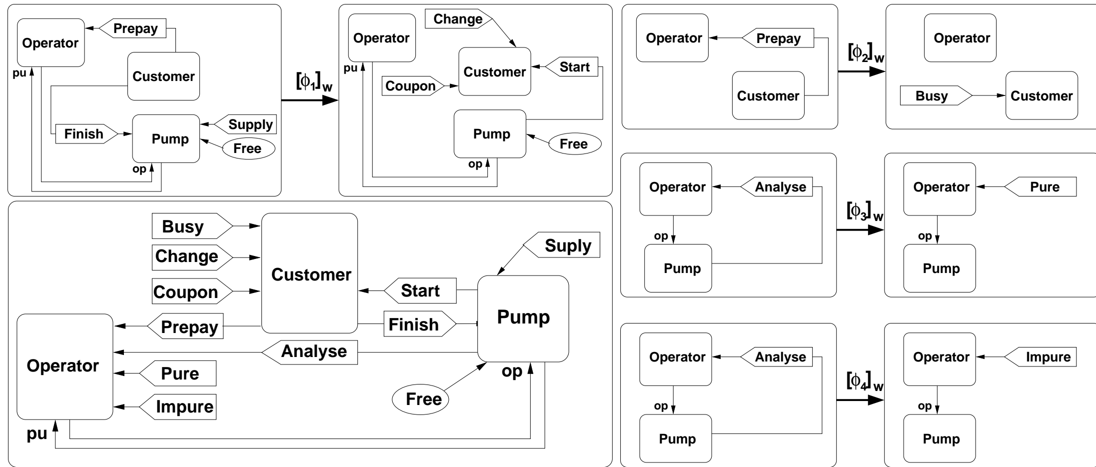


Figure 3.12: Abstract GTS associated to the T-GTS $\mathcal{P}umpOper$, depicted in Figure 3.1.

It is easy to see that the type graph of this system is the stable type graph of $\mathcal{P}umpOper$. The productions of this GTS *are all transactional process of the original* T-GTS*:*

- *the process $[\phi_1]_w$ is shown in Example 3.4;*
- *the processes $[\phi_2]_w$, $[\phi_3]_w$ and $[\phi_4]_w$ are those which have as the only productions* REJECT, PUREANALYSIS *and* IMPUREANALYSIS, *respectively.*

┘

3.5 Implementation morphisms for T-GTSs

As described in the previous section, a T-GTS can be viewed as a standard GTS, where the unstable states are abstracted away and only the complete transactions are observable. This transformation defines a mapping from the objects of the category $\mathbf{TGTS}$ to those of $\mathbf{GTS}$. Interestingly, equipping the category of transactional GTSs with a more general notion of morphism, – called *implementation morphism* – this mapping can be turned into a functor, which is the right adjoint to the inclusion functor in the opposite direction.

Then, we equip T-GTSs with a suitable notion of implementation morphism, allowing to relate two systems, mapping the productions of one into transactions of the other – this notion is also used to relate the components of a module: the body is the implementation of the interface. Figure 3.13 shows a schematic representation of mapping of a production into a transaction: a production is depicted at the top; and a transactional process implementing this production is depicted at the bottom. The mapping from production to process must preserve the left- and right-hand sides, i.e., all items in these graphs, whose types are preserved, must be in minimal and maximal graphs of the process, respectively.

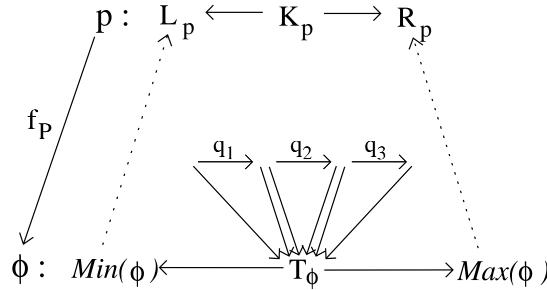


Figure 3.13: Transactional process ϕ implements production p.

T-GTS implementation morphisms. An *implementation morphism* is a T-GTS morphism that maps each given production of the source system to a weak unstable *transactional* process of the target system.

Definition 3.19 (T-GTS implementation morphisms) *Given* T-GTS $\mathcal{Z}_i = \langle\langle T_i, P_i, \pi_i\rangle, T_{is}\rangle$*, let* $\widehat{\mathcal{Z}_i} = \langle\langle T_i, \mathbf{wutProc}(\mathcal{Z}_i), \Pi_{\mathcal{Z}_i}\rangle, T_{is}\rangle$ *be a* T-GTS *having all weak ut-processes of* $\mathcal{Z}_i$ *as productions for* $i \in \{1, 2\}$*. An* (T-GTS) implementation morphism $f \colon \mathcal{Z}_1 \to \mathcal{Z}_2$ *is the* T-GTS *morphism* $\langle f_T, f_P\rangle \colon \mathcal{Z}_1 \to \widehat{\mathcal{Z}_2}$*.*

Example 3.6 ((T-GTS) implementation morphism) *In Figure 3.14 we can see an implementation morphism between the abstract* GTS *of the customer system to the concrete* T-GTS *of this system.*

The type component e_T *is the obvious inclusion and the production component* e_P *maps each production* $[\varphi_i]_w$ *of* $A(\mathcal{C}ustomer)$ *into the corresponding weak t-process* $[\varphi_i]_w$ *of* $\mathcal{C}ustomer$*. In Figure 3.15, the type graphs and the set of name productions of each wt-process of* $\mathcal{C}ustomer$ *system are shown.*

┘

To provide a correct definition of the category having T-GTSs as objects and implementation morphisms as arrows, we have to explain how implementation morphisms compose. This is summarised by the next propositions. In order to compose implementation

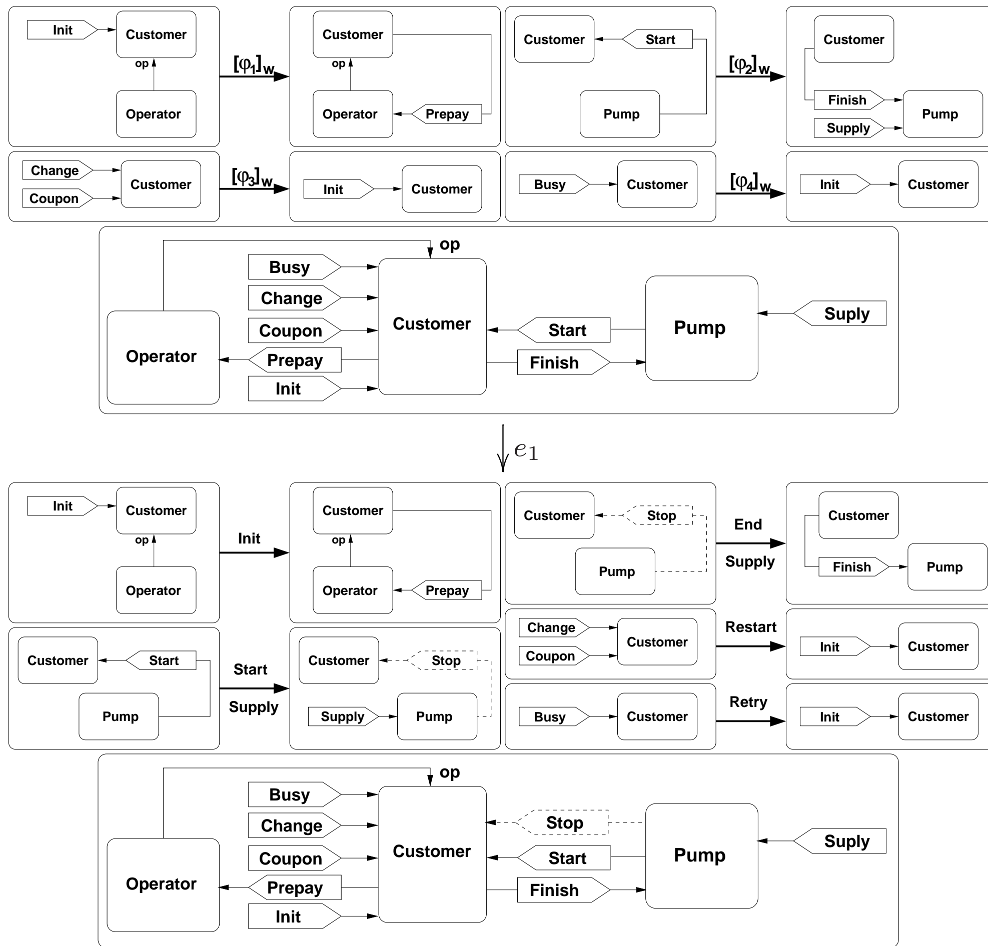


Figure 3.14: Implementation morphism from the abstract GTS $A(\mathcal{C}ustomer)$ to the T-GTS $\mathcal{C}ustomer$.

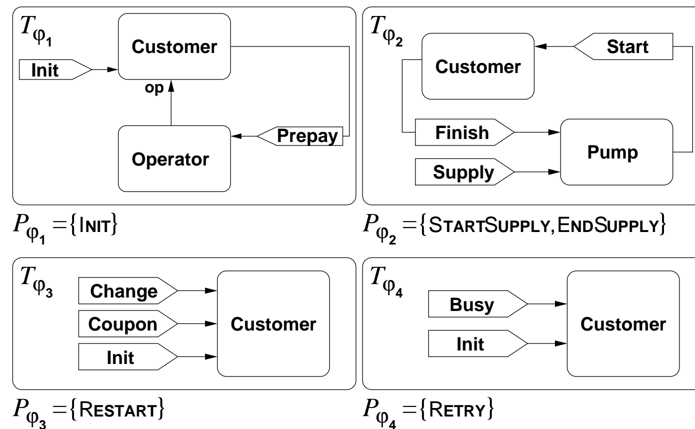


Figure 3.15: Transactional processes of customer system.

morphisms, we first have to know how to map wut-processes of source T-GTS into wut-processes of target one. Proposition 3.5 shows how to obtain this mapping extending the mapping of productions of source (T-GTS) into wut-processes of target one. The proof of this proposition is divided in two parts: (1) the definition of the extension; and (2) the

proof that the extension is well-defined, i.e., it results in a unique implementation morphism (the productions component maps each wut-process into a unique class of weak ut-processes, independently of the concrete choice for the mapping between productions and t-processes).

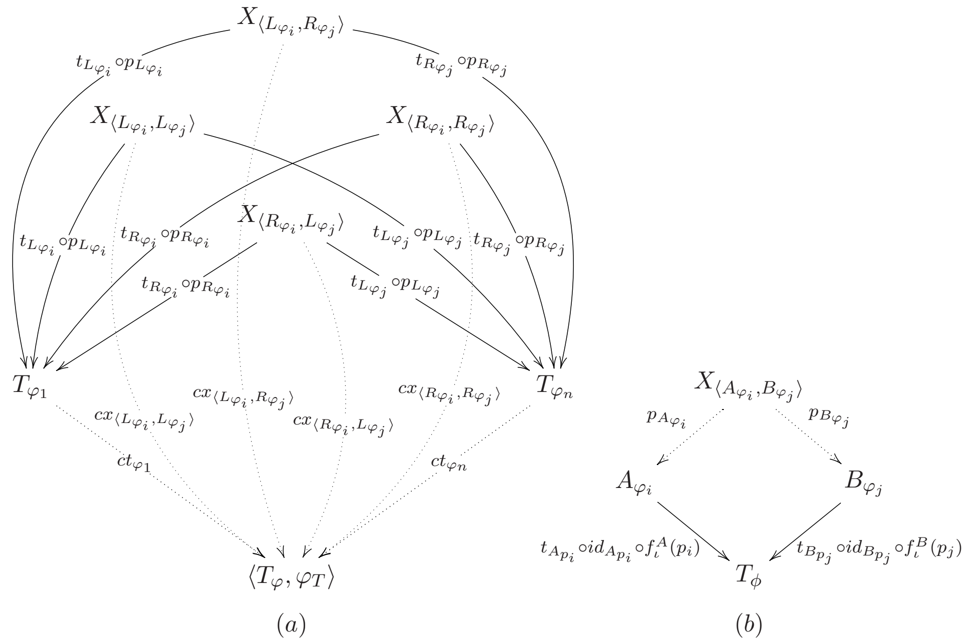
Given a T-GTS $\mathcal{Z}$ and a production p in $\mathcal{Z}$, below we denote by ϕ_{id_p} the process associated (see Definition 3.10) to the one-step derivation which applies p to its left-hand side L_p with the identity match.

Proposition 3.5 *Given a* T-GTS $\mathcal{Z}_i$, *let* $\widehat{\mathcal{Z}_i}$ *be as in Definition 3.19, for* $i \in \{1,2\}$. *Then any* T-GTS *morphism* $f\colon \mathcal{Z}_1 \to \widehat{\mathcal{Z}_2}$ *extends to a* T-GTS *morphism* $\widehat{f}\colon \widehat{\mathcal{Z}_1} \to \widehat{\mathcal{Z}_2}$.

Proof:

1. $\widehat{f}$ is defined by $\langle f_T, \widehat{f_P}\rangle$, where for all weak ut-process $[\phi]_w \in \mathbf{wutProc}(\mathcal{Z}_1)$, $\widehat{f_P}([\phi]_w) = [\varphi]_w$, with $[\varphi]_w \in \mathbf{wutProc}(\mathcal{Z}_2)$ and $\varphi : \mathcal{O}_\varphi \to \mathcal{Z}_2 = \langle \varphi_T, \varphi_P\rangle$ defined as follows:

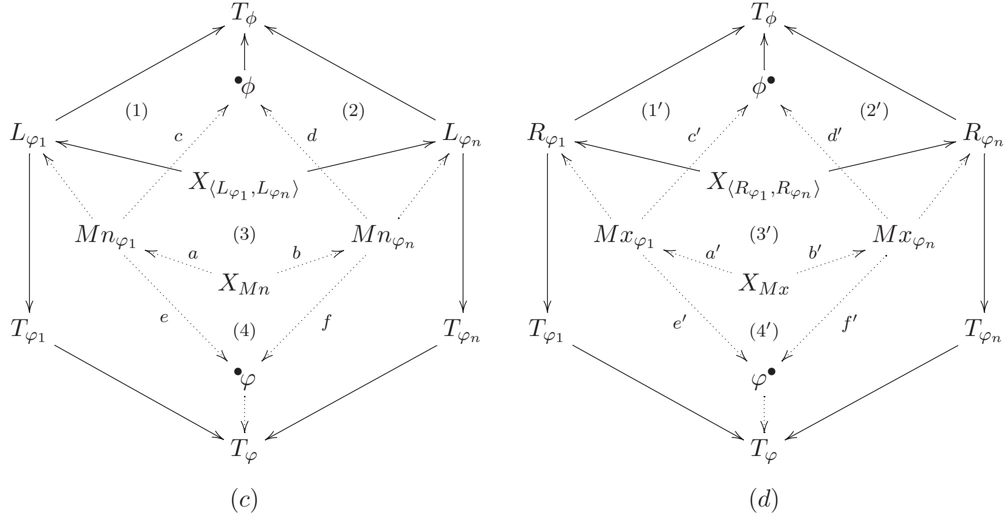
   - for $1 \leq i < j \leq n$ and $n = \#P_\phi$, $\langle T_\varphi, \varphi_T\rangle$ is the colimit (in T_2-**Graph**) of the diagram (a) depicted (only for the processes φ_1 and φ_n – the other ones are omitted) below:



(a) (b)

   where $p_i \in P_\phi$, $[\varphi i]_w = f_P(\phi_P(p_i))$, $\Pi_{\mathcal{Z}_2}([\varphi i]_w) = L_{\varphi_i} \leftarrow K\varphi_i \rightarrow R_{\varphi_i}$ and $\langle X_{\langle A_{\varphi_i}, B_{\varphi_j}\rangle}, p_{A_{\varphi_i}}, p_{B_{\varphi_j}}\rangle$ is the pullback (in **Graph**) of the diagram (b) above, with $A, B \in \{L, R\}$ and $f_\iota^A(p_i)$ and $f_\iota^B(p_j)$ are any choice for the isomorphisms required by definition of GTS morphism.

   - $P_\varphi = \{q | q \in P_{\varphi_k}, k = 1..\#P_\phi\}$
   - for all $q \in P_\varphi$, if $q \in P_{\varphi_k}$ then $\varphi_P(q) = {\varphi_k}_P(q)$.
   - $T_{\varphi_s} \hookrightarrow T_\varphi = \varphi_T^*(T_{2s} \hookrightarrow T_2)$
   - ${}^\bullet\varphi$ and $\varphi^\bullet$, minimal and maximal graphs, respectively, are defined by colimits (4) and $(4')$ depicted in diagrams below, where (1), (2), (3), $(1')$, $(2')$ and $(3')$ are pullbacks.



2. $\widehat{f}$ is well-defined. Since f is a T-GTS morphism, f_T is a graph morphism and preserves stable and unstable items. By definition of $\widehat{f}_P$, $\forall [\phi]_w \in \mathbf{wutProc}(\mathcal{Z}_1)\,\text{.}\, \widehat{f}_P([\phi]_w) = [\varphi]_w$. It remains to prove that any choice of the isomorphisms $f_\iota^A(p_i)$ and $f_\iota^B(p_j)$, in diagram (b) above, determines the same weak ut-process. Therefore, we must prove that all concrete process obtained as above from the same process ϕ, are weak equivalent, i.e., there exists a bijection between their sets of productions and their type, minimal and maximal graphs are isomorphic. Since the set of productions are always the same, independent of the choice for $f_\iota^A(p_i)$ and $f_\iota^B(p_j)$, we only must prove that type, minimal and maximal graphs are isomorphic.
   By definition of T-GTS morphism, there exists $f_\iota^A(p_i)$ such that (1) below is a pullback, with $A, B \in \{L, R\}$.

$$\begin{array}{ccccc} A_{p_i} & \xleftarrow{f_\iota^A(p_i)} & A_{\varphi_i} & & \\ \downarrow t_{A_{p_i}} & (1) & \downarrow t_{A_{\varphi_i}} & \searrow & \\ T_1 & \longleftarrow & dom(f_T) & \longrightarrow & T_2 \end{array}$$

   If there exists a different arrow $f_\iota^{A'}(p_i) : A_{\varphi_i} \to A_{p_i}$ such that (1) is a pullback, then (i) $\exists x \in A_{\varphi_i}\,\text{.}\, f_\iota^A(p_i)(x) \neq f_\iota^{A'}(p_i)(x)$ only if $t_{A_{p_i}}(f_\iota^A(p_i)(x)) = t_{A_{p_i}}(f_\iota^{A'}(p_i)(x))$.

   (a) Type graphs are isomorphic: If we use, in diagram (b) above, $f_\iota^{A'}(p_i)$ instead of $f_\iota^A(p_i)$, we will obtain a graph $X'_{\langle A_{\varphi_i}, B_{\varphi_j}\rangle}$, as depicted in the following diagram,

$$\begin{array}{ccccc} & X_{\langle A_{\varphi_i}, B_{\varphi_j}\rangle} & \underset{h'}{\overset{h}{\rightleftarrows}} & X'_{\langle A_{\varphi_i}, B_{\varphi_j}\rangle} & \\ p_{A_{\varphi_i}} \swarrow \; p'_{A_{\varphi_i}} & & p_{B_{\varphi_j}} & & \searrow p'_{B_{\varphi_j}} \\ A_{\varphi_i} & & t_{A_{p_i}} \circ id_{A_{p_i}} \circ f_\iota^{A'}(p_i) & & B_{\varphi_j} \\ & t_{A_{p_i}} \circ id_{A_{p_i}} \circ f_\iota^{A}(p_i) \searrow & & \swarrow t_{B_{p_j}} \circ id_{B_{p_j}} \circ f_\iota^{B}(p_j) & \\ & & T_\phi & & \\ & & \downarrow \phi_T & & \\ & & T_1 & & \end{array}$$

which is equivalent to $X_{\langle A_{\varphi_i}, B_{\varphi_j}\rangle}$: in the following we will use $f'_A = t_{A_{p_i}} \circ id_{A_{p_i}} \circ f_\iota^{A'}(p_i)$, $f_A = t_{A_{p_i}} \circ id_{A_{p_i}} \circ f_\iota^{A}(p_i)$ and $f_B = t_{B_{p_j}} \circ id_{B_{p_j}} \circ f_\iota^{B}(p_j)$.

- By (i), we have:
  (ii) $\forall e \in A_{\varphi_i} \centerdot f_\iota^A(p_i)(e) \neq f_\iota^{A'}(p_i)(e)$ only if $\phi_T(f_A(e)) = \phi_T(f'_A(e))$;
- By definition of pullbacks:
  (iii) $\forall e_1 \in A_{\varphi_i} \wedge \forall e_2 \in B_{\varphi_j} \centerdot f_B(e_2) = f_A(e_1) \centerdot$
  $\exists x \in X_{\langle A_{\varphi_i}, B_{\varphi_j}\rangle} \centerdot f_A(p_{A_{\varphi_i}}(x)) = f_B(p_{B_{\varphi_j}}(x))$
  (iv) $\forall e_1 \in A_{\varphi_i} \wedge \forall e_2 \in B_{\varphi_j} \centerdot f_B(e_2) = f'_A(e_1) \centerdot$
  $\exists x' \in X'_{\langle A_{\varphi_i}, B_{\varphi_j}\rangle} \centerdot f'_A(p'_{A_{\varphi_i}}(x)) = f_B(p'_{B_{\varphi_j}}(x))$
- By (ii), (iii) and (iv), we can define:
  (v) $h : X_{\langle A_{\varphi_i}, B_{\varphi_j}\rangle} \to X'_{\langle A_{\varphi_i}, B_{\varphi_j}\rangle}$, as follows: $\forall x \in X_{\langle A_{\varphi_i}, B_{\varphi_j}\rangle}$

$$h(x) = \begin{cases} x'_1, & \text{if } p_A(x) = e \wedge f_A(e) = f'_A(e), \\ & \text{where } p'_A(x'_1) = e; \\ x'_2, & \text{if } p_A(x) = e \wedge f_A(e) \neq f'_A(e) = e', \\ & \text{where } f'_A(p'_A(x'_2)) = e'. \end{cases}$$

  where, by definition, $p_A = p'_A \circ h$ and $p_B = p'_B \circ h$.
  (vi) $h' : X'_{\langle A_{\varphi_i}, B_{\varphi_j}\rangle} \to X_{\langle A_{\varphi_i}, B_{\varphi_j}\rangle}$, as follows: $\forall x' \in X'_{\langle A_{\varphi_i}, B_{\varphi_j}\rangle}$

$$h'(x') = \begin{cases} x_1, & \text{if } p'_A(x') = e \wedge f_A(e) = f'_A(e), \\ & \text{where } p_A(x_1) = e; \\ x_2, & \text{if } p'_A(x') = e \wedge f'_A(e) \neq f_A(e) = e', \\ & \text{where } f_A(p_A(x_2)) = e'. \end{cases}$$

  where, by definition, $p'_B = p_B \circ h'$ and $p'_A = p_A \circ h'$.
- By commutativity of diagram above, we have:

$$\begin{array}{lll} & f'_A \circ p'_A = f_B \circ p'_B & \\ \text{(vii)} & f'_A \circ p_A \circ h' = f_B \circ p'_B & \text{by (vi)} \\ & f'_A \circ p'_A \circ h \circ h' = f_B \circ p'_B & \text{by (v)} \\ & f_B \circ p'_B \circ h \circ h' = f_B \circ p'_B & \text{by definition of pullback} \\ & f_B \circ p'_B \circ h \circ h' = f_B \circ p'_B \circ id_{X'_{\langle A_{\varphi_i}, B_{\varphi_j}\rangle}} & \text{by definition of identity} \\ \text{(viii)} & h \circ h' = id_{X'_{\langle A_{\varphi_i}, B_{\varphi_j}\rangle}} & \text{by injectivity of } f_B \circ p'_B \end{array}$$

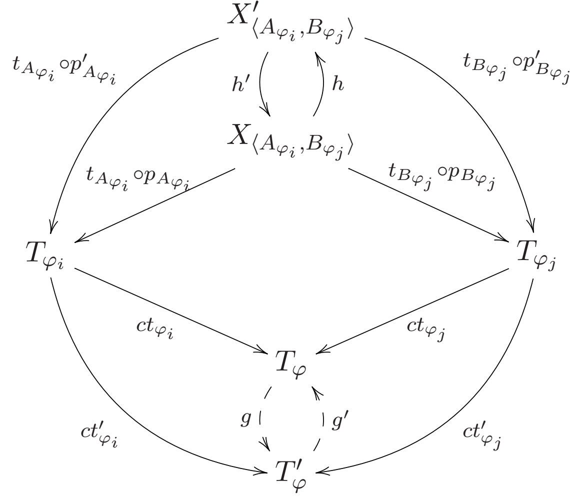
- By commutativity of diagram above, we have:

$$\begin{array}{lll} & f_A \circ p_A = f_B \circ p_B & \\ & f_A \circ p'_A \circ h = f_B \circ p_B & \text{by (v)} \\ & f_A \circ p_A \circ h' \circ h = f_B \circ p_B & \text{by (vi)} \\ & f_B \circ p_B \circ h' \circ h = f_B \circ p_B & \text{by definition of pullback} \\ & f_B \circ p_B \circ h' \circ h = f_B \circ p_B \circ id_{X_{\langle A_{\varphi_i}, B_{\varphi_j}\rangle}} & \text{by definition of identity} \\ \text{(ix)} & h' \circ h = id_{X_{\langle A_{\varphi_i}, B_{\varphi_j}\rangle}} & \text{by injectivity of } f_B \circ p_B \end{array}$$

- By (viii), (ix) and (ii) we have $X_{\langle A_{\varphi_i}, B_{\varphi_j}\rangle} \approx X'_{\langle A_{\varphi_i}, B_{\varphi_j}\rangle}$ in T_1-**Graph**.

Using $X_{\langle A_{\varphi_i}, B_{\varphi_j}\rangle}$ and $X'_{\langle A_{\varphi_i}, B_{\varphi_j}\rangle}$ to construct the type graph of φ we obtain T_φ and T'_φ, respectively, as the colimits depicted in diagram below. Now, we must prove that

$T_\varphi \approx T'_\varphi$.



- By (v) and (vi) we have:

(x) $t_{A_{\varphi_i}} \circ p'_{A_{\varphi_i}} = t_{A_{\varphi_i}} \circ p_{A_{\varphi_i}} \circ h'$ and $t_{A_{\varphi_i}} \circ p_{A_{\varphi_i}} = t_{A_{\varphi_i}} \circ p'_{A_{\varphi_i}} \circ h$

(xi) $t_{B_{\varphi_j}} \circ p_{B_{\varphi_j}} = t_{B_{\varphi_j}} \circ p'_{B_{\varphi_j}} \circ h$ and $t_{B_{\varphi_j}} \circ p'_{B_{\varphi_j}} = t_{B_{\varphi_j}} \circ p_{B_{\varphi_j}} \circ h'$

- By (x), (xi) and commutativity of diagram above, we have:

$$\begin{array}{llcl}
\text{(xii)} & ct'_{\varphi_i} \circ t_{A_{\varphi_i}} \circ p_{A_{\varphi_i}} & = & ct'_{\varphi_i} \circ t_{A_{\varphi_i}} \circ p'_{A_{\varphi_i}} \circ h \\
 & & = & ct'_{\varphi_j} \circ t_{B_{\varphi_j}} \circ p'_{B_{\varphi_j}} \circ h \\
 & & = & ct'_{\varphi_j} \circ t_{B_{\varphi_j}} \circ p_{B_{\varphi_j}} \\
 & & and & \\
\text{(xiii)} & ct_{\varphi_i} \circ t_{A_{\varphi_i}} \circ p'_{A_{\varphi_i}} & = & ct_{\varphi_i} \circ t_{A_{\varphi_i}} \circ p_{A_{\varphi_i}} \circ h' \\
 & & = & ct_{\varphi_j} \circ t_{B_{\varphi_j}} \circ p_{B_{\varphi_j}} \circ h' \\
 & & = & ct_{\varphi_j} \circ t_{B_{\varphi_j}} \circ p'_{B_{\varphi_j}}
\end{array}$$

- By (xii) and (xiii), we have: $g : T_\varphi \to T'_\varphi$ and $g' : T'_\varphi \to T_\varphi$ as the morphisms uniquely determined by universal property of colimits, such that:

(xiv) $g \circ ct_{\varphi_j} = ct'_{\varphi_j}$ and $g \circ ct_{\varphi_i} = ct'_{\varphi_i}$

(xv) $g' \circ ct'_{\varphi_j} = ct_{\varphi_j}$ and $g' \circ ct'_{\varphi_i} = ct_{\varphi_i}$

- By commutativity of diagram above and by definition of identities, we have:

$$\begin{array}{lcll}
id_{T_\varphi} \circ ct_{\varphi_i} \circ t_{A_{\varphi_i}} \circ p_{A_{\varphi_i}} & = & ct_{\varphi_j} \circ t_{B_{\varphi_j}} \circ p_{B_{\varphi_j}} & \\
 & = & g' \circ ct'_{\varphi_j} \circ t_{B_{\varphi_j}} \circ p_{B_{\varphi_j}} & \text{by (xv)} \\
 & = & g' \circ ct'_{\varphi_i} \circ t_{A_{\varphi_i}} \circ p_{A_{\varphi_i}} & \text{by (xii)} \\
 & = & g' \circ g \circ ct_{\varphi_i} \circ t_{A_{\varphi_i}} \circ p_{A_{\varphi_i}} & \text{by (xiv)} \\
 & and & & \\
id_{T_\varphi} \circ ct_{\varphi_j} \circ t_{B_{\varphi_j}} \circ p_{B_{\varphi_j}} & = & ct_{\varphi_i} \circ t_{A_{\varphi_i}} \circ p_{A_{\varphi_i}} & \\
 & = & g' \circ ct'_{\varphi_i} \circ t_{A_{\varphi_i}} \circ p_{A_{\varphi_i}} & \text{by (xv)} \\
 & = & g' \circ ct'_{\varphi_j} \circ t_{B_{\varphi_j}} \circ p_{B_{\varphi_j}} & \text{by (xii)} \\
 & = & g' \circ g \circ ct_{\varphi_j} \circ t_{B_{\varphi_j}} \circ p_{B_{\varphi_j}} & \text{by (xiv)}
\end{array}$$

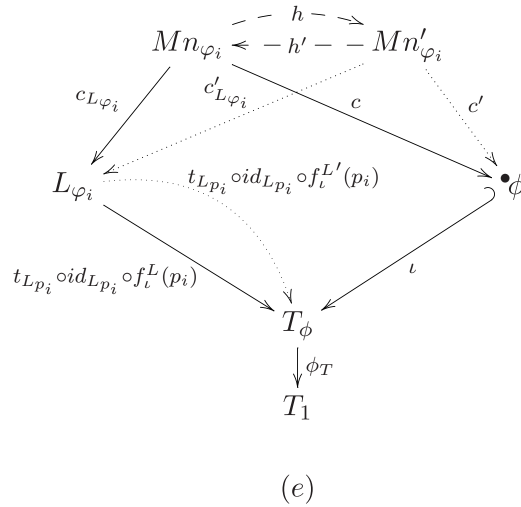
- Since $ct_{\varphi_i} \circ t_{A_{\varphi_i}} \circ p_{A_{\varphi_i}}$ and $ct_{\varphi_j} \circ t_{B_{\varphi_j}} \circ p_{B_{\varphi_j}}$ are jointly surjective, we have:

(xvi) $id_{T_\varphi} = g' \circ g$

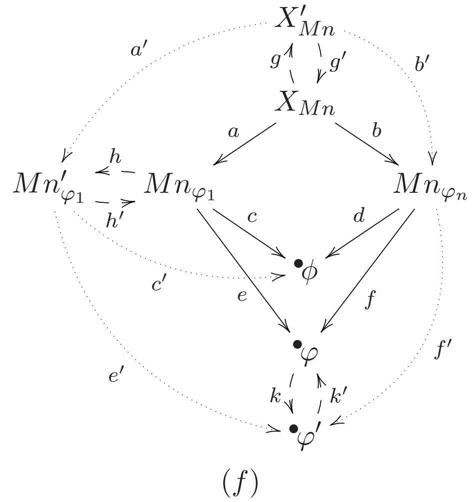
$$\begin{array}{rcll} id_{T'_\varphi} \circ ct'_{\varphi_i} \circ t_{A_{\varphi_i}} \circ p'_{A_{\varphi_i}} & = & ct'_{\varphi_j} \circ t_{B_{\varphi_j}} \circ p'_{B_{\varphi_j}} & \\ & = & g \circ ct_{\varphi_j} \circ t_{B_{\varphi_j}} \circ p'_{B_{\varphi_j}} & \text{by (xiv)} \\ & = & g \circ ct_{\varphi_i} \circ t_{A_{\varphi_i}} \circ p'_{A_{\varphi_i}} & \text{by (xiii)} \\ & = & g \circ g' \circ ct'_{\varphi_i} \circ t_{A_{\varphi_i}} \circ p'_{A_{\varphi_i}} & \text{by (xv)} \\ & and & & \\ id_{T'_\varphi} \circ ct'_{\varphi_j} \circ t_{B_{\varphi_j}} \circ p'_{B_{\varphi_j}} & = & ct'_{\varphi_i} \circ t_{A_{\varphi_i}} \circ p'_{A_{\varphi_i}} & \\ & = & g \circ ct_{\varphi_i} \circ t_{A_{\varphi_i}} \circ p'_{A_{\varphi_i}} & \text{by (xiv)} \\ & = & g \circ ct_{\varphi_j} \circ t_{B_{\varphi_j}} \circ p'_{B_{\varphi_j}} & \text{by (xiii)} \\ & = & g \circ g' \circ ct'_{\varphi_j} \circ t_{B_{\varphi_j}} \circ p'_{B_{\varphi_j}} & \text{by (xv)} \end{array}$$

- Since $ct'_{\varphi_i} \circ t_{A_{\varphi_i}} \circ p'_{A_{\varphi_i}}$ and $ct'_{\varphi_j} \circ t_{B_{\varphi_j}} \circ p'_{B_{\varphi_j}}$ are jointly surjective, we have:
  (xviii) $id_{T'_\varphi} = g \circ g'$
- Therefore, by (xvii) and (xviii), we have that $T_\varphi \approx T'_\varphi$.

(b) Minimal graphs are isomorphic: If we use, in diagram (c) above, $f_\iota^{A'}(p_i)$ instead of $f_\iota^{A}(p_i)$, we will obtain a graph Mn'_{φ_i}, as depicted in the diagram (e) below,



(e)



(f)

which is equivalent to Mn_{φ_i}: it holds by symmetry with $X'_{\langle A_{\varphi_i}, B_{\varphi_j}\rangle} \approx X_{\langle A_{\varphi_i}, B_{\varphi_j}\rangle}$ proved above. Using Mn_{φ_i} and Mn'_{φ_i} to construct the pullback objects X_{Mn} and X'_{Mn}, we have g and g' uniquely defined by universal property of pullbacks, where (i) $a \circ g' = h' \circ a'$, (ii) $b \circ g' = b'$, (iii) $h \circ a = a' \circ g$ and (iv) $b' \circ g = b$.

$$\begin{array}{rll} & h \circ h' \circ a' = h \circ a \circ g' & \text{by (i)} \\ & id_{Mn'_{\varphi_1}} \circ a' = h \circ a \circ g' & \text{by definition of } h \text{ and } h'. \\ & a' = h \circ a \circ g' & \text{by definition of identity.} \\ & a' = a' \circ g \circ g' & \text{by (iii)} \\ & a' \circ id_{X'_{Mn}} = a' \circ g \circ g' & \text{by definition of identity.} \\ \text{(v)} & id_{X'_{Mn}} = g \circ g' & \text{by injectivity of } a'. \end{array}$$

$$\begin{array}{rll} & h' \circ h \circ a = h' \circ a' \circ g & \text{by (iii)} \\ & id_{Mn_{\varphi_1}} \circ a = h' \circ a' \circ g & \text{by definition of } h \text{ and } h'. \\ & a = h' \circ a' \circ g & \text{by definition of identity.} \\ & a = a \circ g' \circ g & \text{by (i)} \\ & a \circ id_{X_{Mn}} = a \circ g' \circ g & \text{by definition of identity.} \\ \text{(vi)} & id_{X_{Mn}} = g' \circ g & \text{by injectivity of } a. \end{array}$$

By (v) and (vi) we have $X'_{Mn} \approx X_{Mn}$. If we construct ${}^{\bullet}\varphi$ and ${}^{\bullet}\varphi'$ as in the diagram (f) above, we have k and k' uniquely determined by property of colimits, where (vii) $e' \circ h = k \circ e$, (viii) $f' = k \circ f$, (ix) $e \circ h' = k' \circ e'$ and (x) $f = k' \circ f'$.

$$
\begin{array}{lll}
 & k' \circ e' \circ h = e \circ h' \circ h & \text{by (ix)} \\
 & k' \circ e' \circ h = e \circ id_{Mn_{\varphi_1}} & \text{by definition of } h \text{ and } h'. \\
 & k' \circ e' \circ h = e & \text{by definition of identity.} \\
 & k' \circ k \circ e = e & \text{by (vii)} \\
\text{(xi)} & k' \circ k \circ e = id_{{}^{\bullet}\varphi} \circ e & \text{by definition of identity.}
\end{array}
$$

$$
\begin{array}{lll}
 & k' \circ k \circ f = f & \text{by (x) and (viii)} \\
\text{(xii)} & k' \circ k \circ f = id_{{}^{\bullet}\varphi} \circ f & \text{by definition of identity.}
\end{array}
$$

$$
\begin{array}{lll}
 & k \circ e \circ h' = e' \circ h \circ h' & \text{by (vii)} \\
 & k \circ e \circ h' = e' \circ id_{Mn'_{\varphi_1}} & \text{by definition of } h \text{ and } h'. \\
 & k \circ e \circ h' = e' & \text{by definition of identity.} \\
 & k \circ k' \circ e' = e' & \text{by (ix)} \\
\text{(xiii)} & k \circ k' \circ e' = id_{{}^{\bullet}\varphi'} \circ e' & \text{by definition of identity.}
\end{array}
$$

$$
\begin{array}{lll}
 & k \circ k' \circ f' = f' & \text{by (viii) and (x)} \\
\text{(xiv)} & k \circ k' \circ f' = id_{{}^{\bullet}\varphi'} \circ f' & \text{by definition of identity.}
\end{array}
$$

Since e and f are jointly surjective and by (xi) and (xii), then $k' \circ k = id_{{}^{\bullet}\varphi}$. By symmetry, $k \circ k' = id_{{}^{\bullet}\varphi'}$. Therefore, ${}^{\bullet}\varphi \approx {}^{\bullet}\varphi'$.

(c) Maximal graphs are isomorphic: by symmetry with minimal graphs, we have $\varphi^{\bullet} \approx \varphi^{\bullet\prime}$.

□

Now we can explain how to compose implementation morphisms. In the next proposition, we define this composition and prove that it is well-defined and associative. Moreover, we define the identities of this morphisms and prove its neutrality with respect to composition. Figure 3.16 shows a schema of the composition of three implementation morphisms, on the productions component: the black diagram describes the mapping of productions of one T-GTS into wut-processes of another, where $f_P : P_{\mathcal{Z}_1} \rightarrow \mathbf{wutProc}(\mathcal{Z}_2)$, $g_P : P_{\mathcal{Z}_2} \rightarrow \mathbf{wutProc}(\mathcal{Z}_3)$ and $h_P : P_{\mathcal{Z}_3} \rightarrow \mathbf{wutProc}(\mathcal{Z}_4)$, all depicted by dashed arrows. Moreover, we have that $q_i \in P_\phi$ and $r_j \in P_{\varphi_i}$, for $i \in \{1, \ldots, n\}$ and $j = \{1, \ldots, m, \ldots, k, \ldots, z\}$. The composition of two implementation morphisms must map each production of the first T-GTS to a wut-process of the third one. Then, if we want to define the composition $g \circ f$ we have to maps each production of $\mathcal{Z}_1$ to a wut-process of $\mathcal{Z}_3$. Since we have that f_P maps each production of $\mathcal{Z}_1$ to a wut-process of $\mathcal{Z}_2$ and $\widehat{g}_P$ maps each wut-process of $\mathcal{Z}_2$ to a wut-process of $\mathcal{Z}_3$, we define the composition $g \circ f$ on productions as $\widehat{g}_P \circ f_P$. The blue diagram describes the composition of $\widehat{h}_P \circ (\widehat{g}_P \circ f_P)$ (dashed blue arrows) mapping production p (in $\mathcal{Z}_1$) to wut-process ψ_{1mkz} (in $\mathcal{Z}_4$) and the red one describes the composition $(\widehat{\widehat{h}_P \circ g_P}) \circ f_P)$ (dashed red arrows) mapping the same production to ψ'_{1mkz} (in $\mathcal{Z}_4$). In order to prove that composition of implementation morphisms is associative we need to show that both compositions (blue and red) are equivalent, i.e., $\psi_{1mkz} \approx^w \psi'_{1mkz}$. The blue and red processes in this figure are obtained gluing other ones, for example, the process φ_{1n} is obtained gluing the processes φ_i $(i \in \{1, n\})$ with respect to their items in common in T_ϕ (X_0). In Proposition 3.6 we define this composition and prove its associativity. The following lemmas are used to prove this proposition.

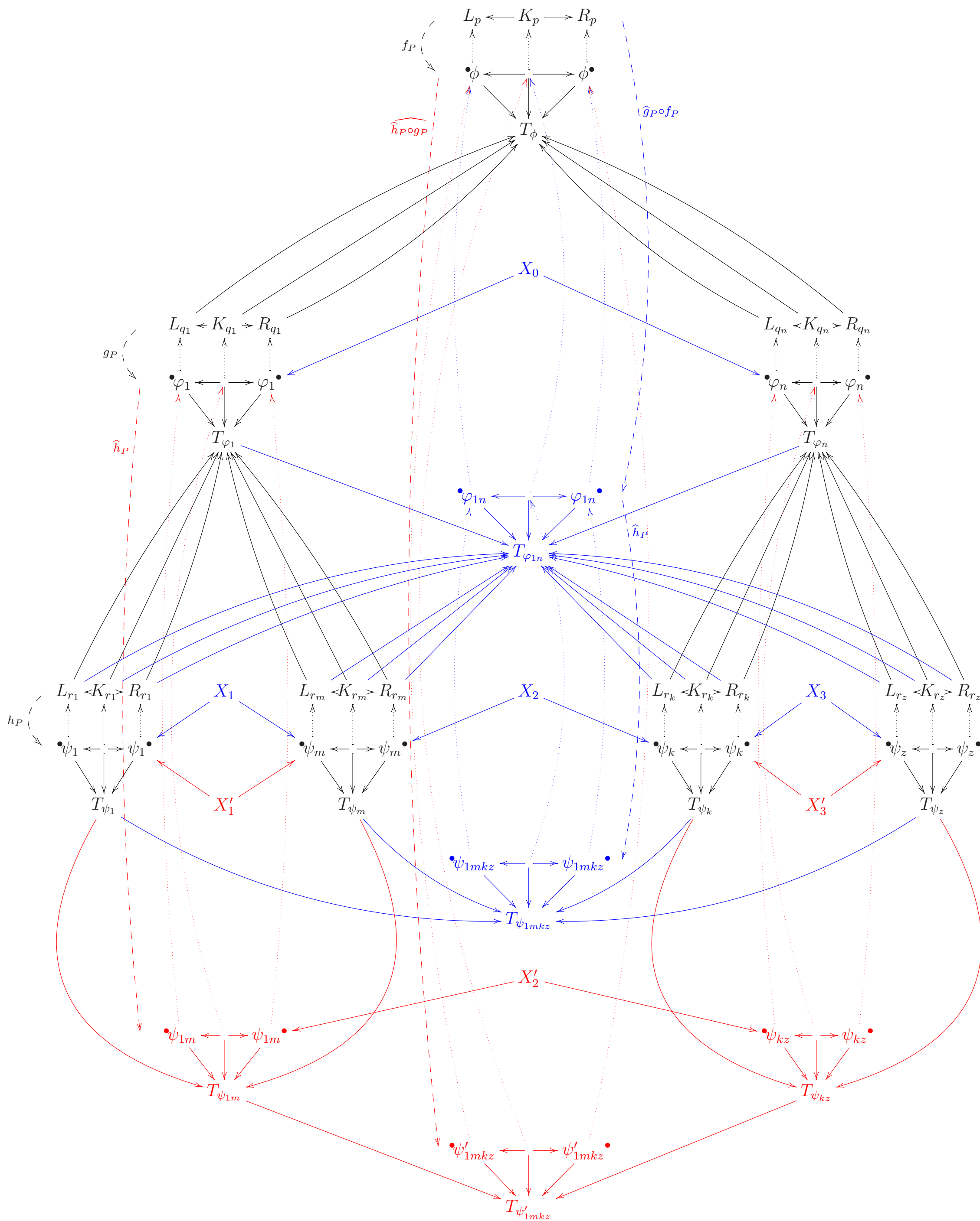
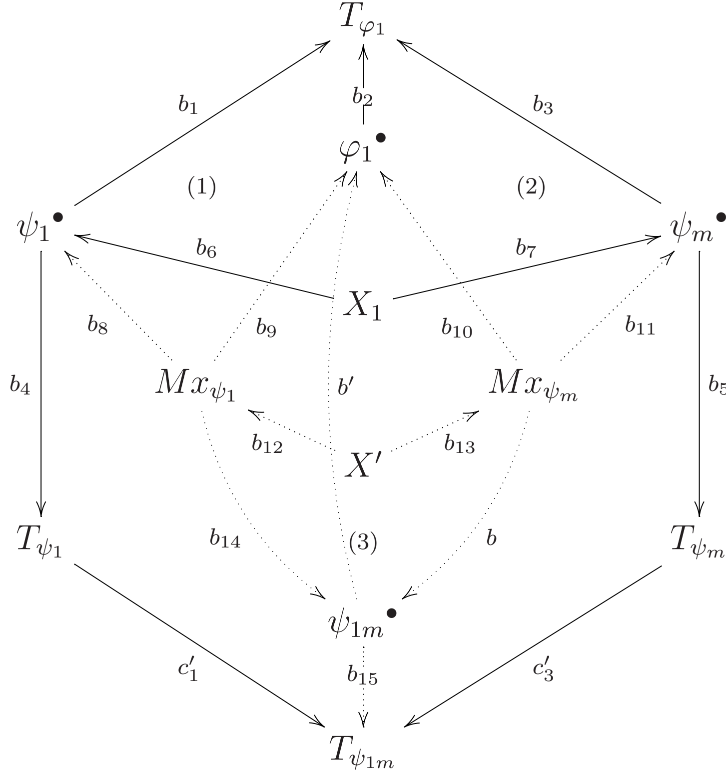


Figure 3.16: Composition of implementation T-GTS morphisms is associative.

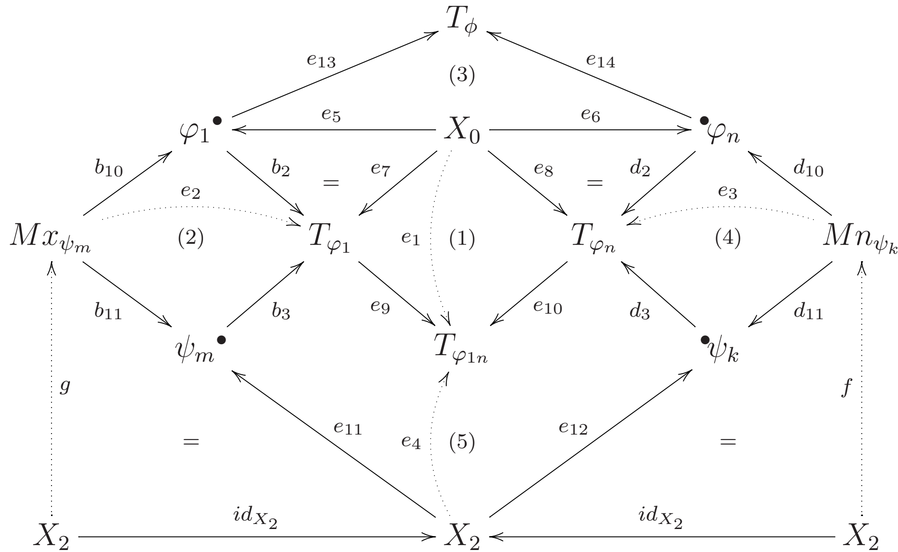
Lemma 3.1 *Given the following diagram in* **Graph**, *obtained by the extension of implementation morphisms defined as in Proposition 3.5, where* (1), (2) *and* $\langle X_1, b_6, b_7\rangle$ *are pullbacks and* $c_1' \circ b_4 \circ b_6 = c_3' \circ b_5 \circ b_7$*. Then there exist* $b : Mx_{\psi_m} \rightarrow \psi_{1m}{}^{\bullet}$,

$b_{11} : Mx_{\psi_m} \to {\psi_m}^\bullet$ and $b_{15} : {\psi_{1m}}^\bullet \to T_{\psi_{1m}}$, such that $c_3' \circ b_5 \circ b_{11} = b_{15} \circ b$.



Proof: b_{11} is defined by pullback (2). X' is obtained as pullback object of b_9 and b_{10}, and b is defined by the colimit (3). $b_{15} : {\psi_{1m}}^\bullet \to T_{\psi_{1m}}$ is uniquely determined by the universal property of the colimit (3), such that $c_3' \circ b_5 \circ b_{11} = b_{15} \circ b$. Moreover, $b' : {\psi_{1m}}^\bullet \to {\varphi_1}^\bullet$ is uniquely determined by universal property of colimit (3), such that $b' \circ b = b_{10}$. □

Lemma 3.2 *Let us consider the following commutative diagram, where (1) is a colimit and (2) − (5) are pullbacks. Then there exists $g : X_2 \to Mx_{\psi_m}$, such that $b_{11} \circ g = e_{11}$; and there exists $f : X_2 \to Mn_{\psi_k}$, such that $d_{11} \circ f = e_{12}$.*



Proof:

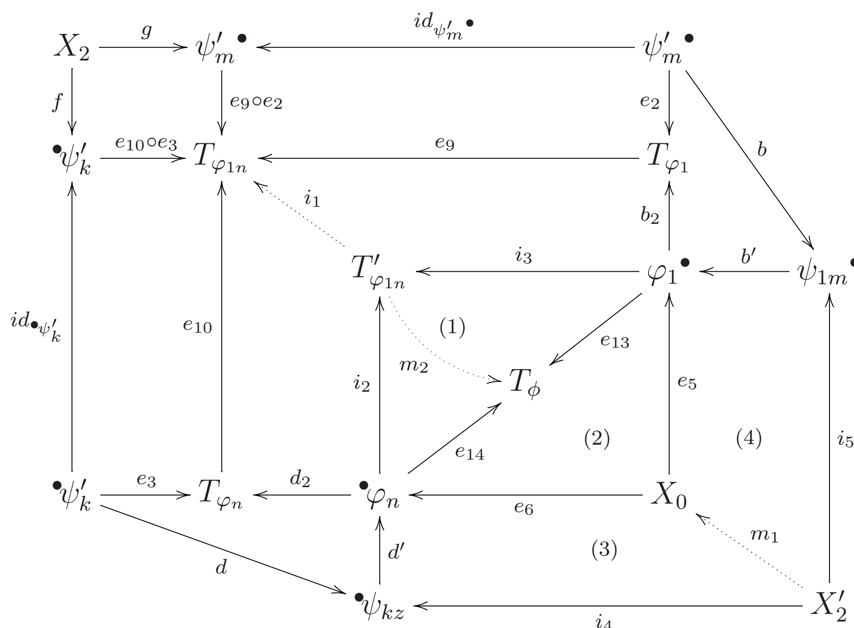
(i)	$rng(e_9) \cap rng(e_{10}) = rng(e_1)$	by (1) and (3)
(ii)	$rng(e_9) \cap rng(e_{10}) \supseteq rng(e_4)$	by (5)
	$rng(e_1) \supseteq rng(e_4)$	by (i)
(iii)	$rng(b_2) \supseteq rng(e_7)$	by (3) and $b_2 \circ e_5 = e_7$
(iv)	$(rng(b_3) - rng(b_2)) \cap rng(e_7) = \varnothing$	by (iii)
(v)	$rng(e_2) \cap rng(e_7) =$ $rng(b_3) \cap rng(e_7)$	by (2) and (iii)
(vi)	$rng(e_7) \supseteq rng(b_3 \circ e_{11})$	by (ii) and commuta–
(vii)	$rng(b_3) \supseteq rng(b_3 \circ e_{11})$	tivity of diagram above
(viii)	$rng(e_7) \cap rng(b_3) \supseteq rng(b_3 \circ e_{11})$	by (vi) and (vii)
(ix)	$rng(e_2) \cap rng(e_7) \supseteq rng(b_3 \circ e_{11})$	by (v) and $(viii)$
	$rng(e_2) \supseteq rng(b_3 \circ e_{11})$	
	$rng(b_3 \circ b_{11}) \supseteq rng(b_3 \circ e_{11})$	by (ix) and $e_2 = b_3 \circ b_{11}$
(x)	$rng(b_{11}) \supseteq rng(e_{11})$	by injectivity of b_3

By symmetry, (xi) $rng(d_{11}) \supseteq rng(e_{12})$. By (x) and (xi), we have that $\exists g : X_2 \to Mx_{\psi_m}$ and $\exists f : X_2 \to Mn_{\psi_k}$, defined as follows:

$$\forall x \in X_2 \bullet g(x) = y, \text{ where } b_{11}(y) = e_{11}(x)$$
$$\forall x \in X_2 \bullet f(x) = y, \text{ where } d_{11}(y) = e_{12}(x)$$

□

Lemma 3.3 *Let us consider diagram below, where* $(1{+}2)$ *is a pushout and* (2) *and* $(3{+}4)$ *are pullbacks. There exists* $y_2 : X_2 \to X_2'$*, such that* $i_5 \circ y_2 = b \circ g$ *and* $i_4 \circ y_2 = d \circ f$.



Proof:

$i_1 : T'_{\varphi_{1n}} \to T_{\varphi_{1n}}$ and $m_2 : T'_{\varphi_{1n}} \to T_\phi$ are uniquely determined by universal property of pushout $(1+2)$. Moreover, $m_1 : X'_2 \to X_0$ is uniquely determined by universal property of pullback (2).

By commutativity of diagram above, we have:

$$\begin{array}{rrcll} & e_{10} \circ e_3 \circ f & = & e_9 \circ e_2 \circ g & \\ & e_{10} \circ d_2 \circ d' \circ d \circ f & = & e_9 \circ b_2 \circ b' \circ b \circ g & \\ & i_1 \circ i_2 \circ d' \circ d \circ f & = & i_1 \circ i_3 \circ b' \circ b \circ g & \\ \text{(i)} & i_2 \circ d' \circ d \circ f & = & i_3 \circ b' \circ b \circ g & \text{by injectivity of } i_1 \end{array}$$

By (i) and commutativity of diagram above, we have:

$$\begin{array}{rrcl} & m_2 \circ i_2 \circ d' \circ d \circ f & = & m_2 \circ i_3 \circ b' \circ b \circ g \\ \text{(ii)} & e_{14} \circ d' \circ d \circ f & = & e_{13} \circ b' \circ b \circ g \end{array}$$

Since $(3+4)$ is a pullback and by (ii), we have that $y_2 : X_2 \to X'_2$ is uniquely determined by universal property of $(3+4)$, such that $i_5 \circ y_2 = b \circ g$ and $i_4 \circ y_2 = d \circ f$. □

Proposition 3.6 (composition and identity for T-GTS implementation morphisms) *Given a* T-GTS $\mathcal{Z}$*, let* $\widehat{\mathcal{Z}}$ *be as in Definition 3.19. Then, the properties below hold.*

1. *Given implementation morphisms* $f\colon \mathcal{Z}_1 \to \mathcal{Z}_2$ *and* $g\colon \mathcal{Z}_2 \to \mathcal{Z}_3$*, let their composition* $g \circ f\colon\ \mathcal{Z}_1 \to \mathcal{Z}_3$ *be defined by the* T-GTS *morphism* $\langle h_T, h_P\rangle : \widehat{g} \circ f : \mathcal{Z}_1 \to \widehat{\mathcal{Z}_3}$*. Then the composition is well defined and it is associative.*

2. *For each* T-GTS $\mathcal{Z}$*, let* $id_{\mathcal{Z}} = \langle id_{\mathcal{Z}T}, id_{\mathcal{Z}P}\rangle : \mathcal{Z} \to \widehat{\mathcal{Z}}$ *be defined as*

   - *the type graph component* $id_{\mathcal{Z}T}$ *is the identity;*
   - *each production* p *is mapped by* $id_{\mathcal{Z}P}$ *to the abstract process* $[\phi_{id_p}]_w$*;*

   Then $id_{\mathcal{Z}}$ *is well-defined and it is the identity on* $\mathcal{Z}$*.*

Proof:

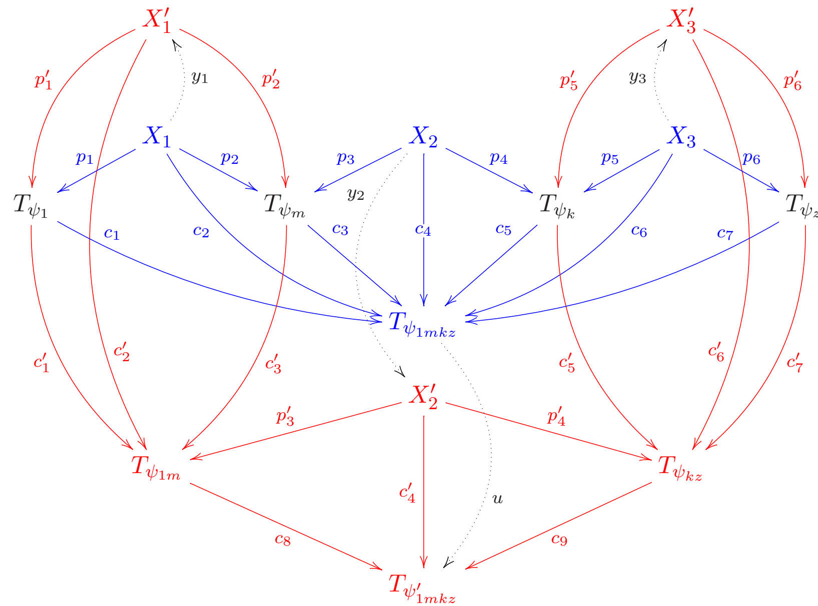
1. Since we proved, by Proposition 3.5, that $\widehat{g}$ maps each wut-process of $\mathcal{Z}_2$ into a wut-process of $\mathcal{Z}_3$ and it is well-defined, then the composition $\widehat{g} \circ f$ is well defined. It remains to prove that the composition is associative. Let $f : \mathcal{Z}_1 \to \mathcal{Z}_2$, $g : \mathcal{Z}_2 \to \mathcal{Z}_3$ and $h : \mathcal{Z}_3 \to \mathcal{Z}_4$ be implementation morphism. Then we must prove:

$$(h \circ g) \circ f = h \circ (g \circ f)$$

(a) $(h_T \circ g_T) \circ f_T = h_T \circ (g_T \circ f_T)$: by associativity of partial graph morphisms.

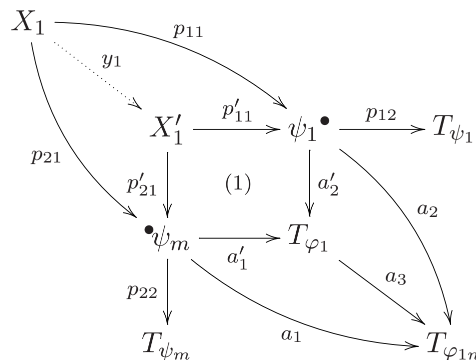
(b) $(\widehat{\widehat{h}_P \circ g_P}) \circ f_P = \widehat{h}_P \circ (\widehat{g}_P \circ f_P)$: as depicted in Figure 3.16, for each production $p \in P_1$. $f_P(p) = [\phi]_w$, $\widehat{g}_P([\phi]_w) = [\varphi_{1n}]_w$ and $\widehat{h}_P([\varphi_{1n}]_w) = [\psi_{1mkz}]_w$. Moreover, for all $q_i \in P_2$. $g_P(q_i) = [\varphi_i]_w$, $\widehat{h}_P([\varphi_1]_w) = [\psi_{1m}]_w$, $\widehat{h}_P([\varphi_n]_w) = [\psi_{kz}]_w$ and $(\widehat{\widehat{h}_P \circ g_P})([\phi]_w) = [\psi'_{1mkz}]_w$. By definition of composition, the sets of productions of $[\psi'_{1mkz}]_w$ and $[\psi_{1mkz}]_w$ contain the same productions (in Figure 3.16: the union of productions in $[\psi_1]_w$, ..., $[\psi_m]_w$, ..., $[\psi_k]_w$, ..., $[\psi_z]_w$). Then it remains to prove that $T_{\psi_{1mkz}} \approx T_{\psi'_{1mkz}}$:

i. $\exists! u : T_{\psi_{1mkz}} \to T_{\psi'_{1mkz}}$, such that the following diagram commutes, i.e., $c_8 \circ c'_1 = u \circ c_1$, $c_8 \circ c'_2 = u \circ c_2$, $c_8 \circ c'_3 = u \circ c_3$, $c_9 \circ c'_5 = u \circ c_5$, $c_9 \circ c'_6 = u \circ c_6$ and $c_9 \circ c'_7 = u \circ c_7$.



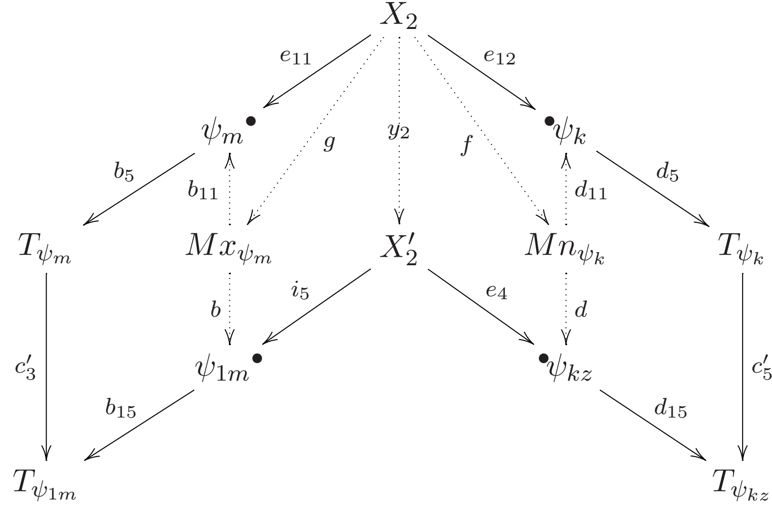
Since $T_{\psi_{1mkz}}$ is the colimit of diagram depicted in blue in diagram above, u is determined by universal property of this colimit. Then we must prove that exists morphisms $y_1 : X_1 \to X'_1$, $y_2 : X_2 \to X'_2$ and $y_3 : X_3 \to X'_3$ such that the diagram commutes.

- $\exists! y_1 : X_1 \to X'_1 \centerdot p'_1 \circ y_1 = p_1 \wedge p'_2 \circ y_1 = p_2$ and $\exists! y_3 : X_3 \to X'_3 \centerdot p'_5 \circ y_3 = p_5 \wedge p'_6 \circ y_3 = p_6$. Consider the commuting diagram below, where (1) is a pullback and (a) $p_1 = p_{12} \circ p_{11}$, (b) $p'_1 = p_{12} \circ p'_{11}$, (c) $p_2 = p_{22} \circ p_{21}$, (d) $p'_2 = p_{22} \circ p'_{21}$.



Since $a'_1 \circ p_{21} = a'_2 \circ p_{11}$ (by commutativity of diagram above), then the morphism $y_1 : X_1 \to X'_1$ is determined by the universal property of pullback (1), such that $p'_{11} \circ y_1 = p_{11}$ and $p'_{21} \circ y_1 = p_{21}$. Then $p'_2 \circ y_1 = p_{22} \circ p'_{21} \circ y_1 = p_{22} \circ p_{21} = p_2$ and $p'_1 \circ y_1 = p_{12} \circ p'_{11} \circ y_1 = p_{12} \circ p_{11} = p_1$. $y_3 : X_3 \to X'_3$ is determined by symmetry.

- $\exists! y_2 : X_2 \to X'_2 \centerdot p'_3 \circ y_2 = p_3 \wedge p'_4 \circ y_2 = p_4$.



By the commuting diagram above, where $p_3 = b_5 \circ e_{11}$, $p_4 = d_5 \circ e_{12}$, $p'_3 = b_{15} \circ i_5$ and $p'_4 = d_{15} \circ i_4$, we have $i_5 \circ y_2 = b \circ b$ and $i_4 \circ y_2 = d \circ f$. Moreover, by commutativity of diagram:

- $c'_3 \circ b_5 \circ e_{11} = c'_3 \circ b_5 \circ b_{11} \circ g = b_{15} \circ b \circ g = b_{15} \circ i_5 \circ y_2$. By definition of p_3 and p'_3, $c'_3 \circ p_3 = p'_3 \circ y_2$
- $c'_5 \circ d_5 \circ e_{12} = c'_5 \circ d_5 \circ d_{11} \circ f = d_{15} \circ d \circ f = d_{15} \circ i_4 \circ y_2$. By definition of p_4 and p'_4, $c'_5 \circ p_4 = p'_4 \circ y_2$

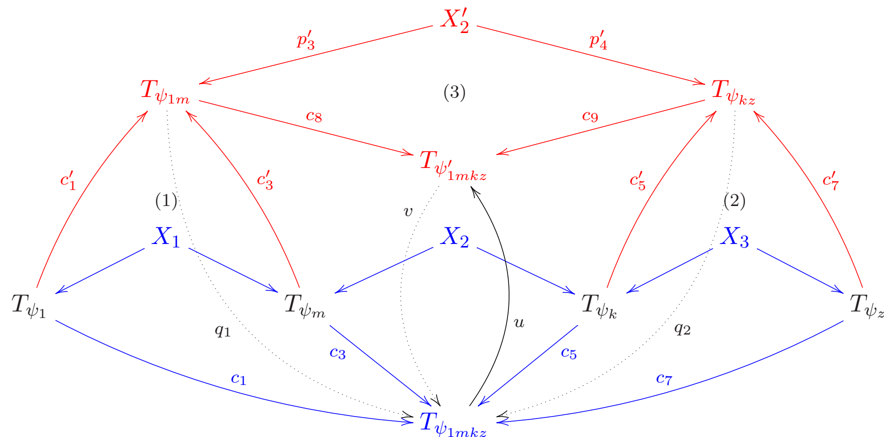
But, in order to construct the diagram above, we must prove:

A. There exist $b : Mx_{\psi_m} \to \psi_{1m}{}^{\bullet}$, $b_{11} : Mx_{\psi_m} \to \psi_m{}^{\bullet}$ and $b_{15} : \psi_{1m}{}^{\bullet} \to T_{\psi_{1m}}$, such that $c'_3 \circ b_5 \circ b_{11} = b_{15} \circ b$. It holds by Lemma 3.1. By symmetry, we have that there exist $d : Mn_{\psi_k} \to {}^{\bullet}\psi_{kz}$, $d_{11} : Mn_{\psi_k} \to {}^{\bullet}\psi_k$ and $d_{15} : {}^{\bullet}\psi_{kz} \to T_{\psi_{kz}}$, such that $c'_5 \circ d_5 \circ d_{11} = d_{15} \circ d$; and there exists $d' : {}^{\bullet}\psi_{kz} \to {}^{\bullet}\varphi_n$, such that $d' \circ d = d_{10}$.

B. There exists $g : X_2 \to Mx_{\psi_m}$, such that $b_{11} \circ g = e_{11}$; and there exists $f : X_2 \to Mn_{\psi_k}$, such that $d_{11} \circ f = e_{12}$. It holds by Lemma 3.2.

C. There exists $y_2 : X_2 \to X'_2$, such that $i_5 \circ y_2 = b \circ g$ and $i_4 \circ y_2 = d \circ f$. It holds by Lemma 3.3.

ii. $\exists! v : T_{\psi'_{1mkz}} \to T_{\psi_{1mkz}}$, such that $v \circ c_8 = q_1$ and $v \circ c_9 = q_2$, as depicted in diagram below.



Since (1) and (2) are pushouts, then there exist $q_1 : T_{\psi_{1m}} \to T_{\psi_{1mkz}}$ and $q_2 : T_{\psi_{kz}} \to T_{\psi_{1mkz}}$ such that (a) $c_1 = q_1 \circ c'_1$, (b) $c_3 = q_1 \circ c'_3$, (a') $c_5 = q_2 \circ c'_5$ and (b') $c_7 = q_2 \circ c'_7$. By definition, $u \circ c_1 = c_8 \circ c'_1$ and $u \circ c_3 = c_8 \circ c'_3$. Then, by (a), $u \circ q_1 \circ c'_1 = c_8 \circ c'_1$ and, by (b), $u \circ q_1 \circ c'_3 = c_8 \circ c'_3$. Since c'_1 and c'_3 are jointly surjective (c) $u \circ q_1 = c_8$. By symmetry, we have (d) $u \circ q_2 = c_9$. Moreover, by definition, $c_8 \circ p'_3 = c_9 \circ p'_4$. By (c) and (d), $u \circ q_1 \circ p'_3 = u \circ q_2 \circ p'_4$. Since u is injective, then (e) $q_1 \circ p'_3 = q_2 \circ p'_4$. By (e), $v : T_{\psi'_{1mkz}} \to T_{\psi_{1mkz}}$ is uniquely determined by universal property of colimit (3), such that $v \circ c_8 = q_1$ and $v \circ c_9 = q_2$.

iii. $v \circ u = id_{T_{\psi_{1mkz}}}$ and $u \circ v = id_{T_{\psi'_{1mkz}}}$.

By i. and ii., we have that:

$$\begin{array}{lrcll} & v \circ c_8 \circ c'_1 & = & q_1 \circ c'_1 & \\ & v \circ u \circ c_1 & = & q_1 \circ c'_1 & \\ & v \circ u \circ c_1 & = & c_1 & \text{by ii.(a)} \\ \text{(a)} & v \circ u \circ c_1 & = & id_{T_{\psi_{1mkz}}} \circ c_1 & \text{by definition of identity} \end{array}$$

$$\begin{array}{lrcll} & v \circ c_8 \circ c'_3 & = & q_1 \circ c'_3 & \\ & v \circ u \circ c_3 & = & q_1 \circ c'_3 & \\ & v \circ u \circ c_3 & = & c_3 & \text{by ii.(b)} \\ \text{(b)} & v \circ u \circ c_3 & = & id_{T_{\psi_{1mkz}}} \circ c_3 & \text{by definition of identity} \end{array}$$

$$\begin{array}{lrcll} & v \circ c_9 \circ c'_5 & = & q_2 \circ c'_5 & \\ & v \circ u \circ c_5 & = & q_2 \circ c'_5 & \\ & v \circ u \circ c_5 & = & c_5 & \text{by ii.(a}'\text{)} \\ \text{(c)} & v \circ u \circ c_5 & = & id_{T_{\psi_{1mkz}}} \circ c_5 & \text{by definition of identity} \end{array}$$

$$\begin{array}{lrcll} & v \circ c_9 \circ c'_7 & = & q_2 \circ c'_7 & \\ & v \circ u \circ c_7 & = & q_2 \circ c'_7 & \\ & v \circ u \circ c_7 & = & c_7 & \text{by ii.(b}'\text{)} \\ \text{(d)} & v \circ u \circ c_7 & = & id_{T_{\psi_{1mkz}}} \circ c_7 & \text{by definition of identity} \end{array}$$

By (a)-(d) and since c_1, c_3, c_5 and c_7 are jointly surjective, then

$$\text{(e) } v \circ u = id_{T_{\psi_{1mkz}}}$$

Moreover, by i., we have:

$$\begin{array}{lrcll} & c_8 \circ c'_1 & = & u \circ c_1 & \\ & & = & u \circ q_1 \circ c'_1 & \text{by ii.(a)} \\ & & = & u \circ v \circ c_8 \circ c'_1 & \text{by ii.} \\ \text{(f)} & id_{T'_{\psi_{1mkz}}} \circ c_8 \circ c'_1 & = & u \circ v \circ c_8 \circ c'_1 & \text{by definition of identity} \end{array}$$

$$\begin{array}{lrcll} & c_8 \circ c'_3 & = & u \circ c_3 & \\ & & = & u \circ q_1 \circ c'_3 & \text{by ii.(b)} \\ & & = & u \circ v \circ c_8 \circ c'_3 & \text{by ii.} \\ \text{(g)} & id_{T'_{\psi_{1mkz}}} \circ c_8 \circ c'_3 & = & u \circ v \circ c_8 \circ c'_3 & \text{by definition of identity} \end{array}$$

$$\begin{array}{lrcll} & c_9 \circ c'_5 & = & u \circ c_5 & \\ & & = & u \circ q_2 \circ c'_5 & \text{by ii.(a}'\text{)} \\ & & = & u \circ v \circ c_9 \circ c'_5 & \text{by ii.} \\ \text{(h)} & id_{T'_{\psi_{1mkz}}} \circ c_9 \circ c'_5 & = & u \circ v \circ c_9 \circ c'_5 & \text{by definition of identity} \end{array}$$

$$
\begin{array}{rrcll}
 & c_9 \circ c'_7 & = & u \circ c_7 & \\
 & & = & u \circ q_2 \circ c'_7 & \text{by ii.(b}') \\
 & & = & u \circ v \circ c_9 \circ c'_7 & \text{by ii.} \\
\text{(i)} & id_{T'_{\psi_{1mkz}}} \circ c_9 \circ c'_7 & = & u \circ v \circ c_9 \circ c'_7 & \text{by definition of identity}
\end{array}
$$

Since c'_1 and c'_3; c'_5 and c'_7; and c_8 and c_8 are jointly surjective, then $c_8 \circ c'_1$, $c_8 \circ c'_3$, $c_9 \circ c'_5$ and $c_9 \circ c'_7$ are jointly surjective, as well. Therefore,

$$\text{(j) } id_{T'_{\psi_{1mkz}}} = u \circ v$$

Thus, by (e) and (j), we have that $T_{\psi_{1mkz}} \approx T'_{\psi_{1mkz}}$.

2. (a) *$id_{\mathcal{Z}}$ is well-defined.* Since $id_{\mathcal{Z}T}$ is an identity, then it is total and preserves stable and unstable items. Moreover, $id_{\mathcal{Z}P}$ maps each production of $\mathcal{Z}$ into a wut-process of $\mathcal{Z}$, it remains to prove that, for all production $p \in P \centerdot id_{\mathcal{Z}P}(p) = [\phi_{id_p}]_w$, there are three morphisms from $\Pi_{\mathcal{Z}}([\phi_{id_p}]_w)$ to $L_p \hookleftarrow K_p \rightarrowtail R_p$. Since $[\phi_{id_p}]_w$ is the wut-process containing only p as production, its underlying span is isomorphic to $L_p \hookleftarrow K_p \rightarrowtail R_p$, then the required morphisms are given by any triple of isomorphisms mapping the span $\Pi_{\mathcal{Z}}([\phi_{id_p}]_w)$ to $L_p \hookleftarrow K_p \rightarrowtail R_p$ and making the two resulting squares commute.

   (b) *$id_{\mathcal{Z}}$ is the identity.* For each implementation morphism $f : \mathcal{Z}_1 \rightarrow \mathcal{Z}$ and $g : \mathcal{Z} \rightarrow \mathcal{Z}_2$, then:

   i. $id_{\mathcal{Z}} \circ f = f$
   - $id_{\mathcal{Z}T} \circ f_T = f_T$ by definition of identity of graphs.
   - $\widehat{id_{\mathcal{Z}P}} \circ f_P = f_P$. Since each production p of $\mathcal{Z}$ is mapped into a wut-processes which contains p as the unique production, for all $[\phi]_w \in \mathbf{wutProc}(\mathcal{Z})$, $\widehat{id_{\mathcal{Z}P}}([\phi]_w) = [\phi]_w$. This holds by definition of $\widehat{id_{\mathcal{Z}P}}$:
     – Let us consider P_ϕ as the set of productions of $\widehat{id_{\mathcal{Z}P}}([\phi]_w)$. Then P_ϕ is composed of all productions of the wut-processes associated with productions of $[\phi]_w$. Since $id_{\mathcal{Z}P}$ associates each production to a process containing it as unique production, then P_ϕ is the same set of productions of $[\phi]_w$.
     – By definitions of graph process (type graph is a colimit of associated derivation) and of wut- process (all items in minimal graph are used), we have that the type graph of $[\phi]_w$ is the colimit of components of all its productions. Let us consider T_ϕ the type graph of $\widehat{id_{\mathcal{Z}P}}([\phi]_w)$, then T_ϕ is obtained as colimit of type graphs of all wut-processes associated with the productions of $[\phi]_w$. As each one of these wut-processes contains only one production, its the type graph is the colimit of the production components. Then, T_ϕ is the colimit of components of all productions in P_ϕ, and therefore T_ϕ is the same type graph of $[\phi]_w$.

   ii. $g \circ id_{\mathcal{Z}} = g$
   - $g_T \circ id_{\mathcal{Z}T} = g_T$ by definition of identity of graphs.
   - $\widehat{g}_P \circ id_{\mathcal{Z}P} = g_P$. By definition of composition, for all $[\phi_{id_p}]_w$ in $\mathbf{wutProc}(\mathcal{Z})$ we have $\widehat{g}_P([\phi_{id_p}]_w) = g_P(p)$. Then,

   $$\forall p \in P \centerdot \widehat{g}_P(id_{\mathcal{Z}}(p)) = \widehat{g}_P([\phi_{id_p}]_w) = g_P(p).$$

   Then, by i. and ii. we can conclude that $id_{\mathcal{Z}}$ is the identity on $\mathcal{Z}$.

□

The Proposition 3.6 allows to introduce a category with implementation morphisms.

Definition 3.20 (category $\mathbf{TGTS}^{imp}$) *We denote by* $\mathbf{TGTS}^{imp}$ *the category having* T-GTS*s as objects and* T-GTS *implementation morphisms as arrows.*

3.6 Adjunction between $\mathbf{GTS}$ and $\mathbf{TGTS}^{imp}$

A T-GTS can be described in a higher level of abstraction, where the unobservable items are hidden and each transaction is performed in a single atomic step. This abstract system is given by a GTS obtained as described in Definition 3.18.

The main result of this chapter is that the abstract GTS associated to a T-GTS has the same behaviour of the original T-GTS, from the point of view of an external observer. This is proven by using a universal construction in the categorical setting: an adjunction between the categories $\mathbf{GTS}$ and $\mathbf{TGTS}^{imp}$. This construction is given by two functors: the abstraction functor $\mathcal{A}\langle_\rangle : \mathbf{TGTS}^{imp} \to \mathbf{GTS}$, that maps each T-GTS into its abstract counterpart; and the functor $\mathcal{I}\langle_\rangle : \mathbf{GTS} \to \mathbf{TGTS}^{imp}$, that allows to see a GTS as a T-GTS whose items are all stable.

Using these functors, we can relate an abstract system to its concrete counterpart through an implementation morphism, mapping each production (of the source) into the transaction (of the target) that originated it – remember that the productions of the abstract systems are all transactions of the concrete one. Since all productions of the abstract system are completely stable, they are also the transactions of this system. Thus, we have a one-to-one relation between the transactions of the abstract and the concrete systems.

Theorem 3.1 (Universality of abstraction) *The functor* $\mathcal{A} : \mathbf{TGTS}^{imp} \to \mathbf{GTS}$, *that maps every* T-GTS $\mathcal{Z}$ *into its abstract* GTS *(see Definition 3.9), has a left adjoint* $\mathcal{I} : \mathbf{GTS} \to \mathbf{TGTS}^{imp}$, *which can be seen as the inclusion of* $\mathbf{GTS}$ *into* $\mathbf{TGTS}^{imp}$.

Proof: This proof can be divided in three parts:

1. Definition of the functor $\mathcal{I}\langle_\rangle : \mathbf{GTS} \to \mathbf{TGTS}^{imp}$:
   - on objects: let $\mathcal{G} = \langle T, P, \pi\rangle$ be a GTS, then $\mathcal{I}\langle G\rangle = \langle\langle T, P, \pi\rangle, T\rangle$;
   - on morphisms: for any $f : \mathcal{G}_1 \to \mathcal{G}_2$, $\mathcal{I}\langle f\rangle : \mathcal{I}\langle \mathcal{G}_1\rangle \to \widehat{\mathcal{I}\langle \mathcal{G}_2\rangle} = \langle f_T, f'_P\rangle$, where $\forall p \in P \,.\, f'_P(p) = [\phi_{id_q}]_w$, such that $f_P(p) = q$.
2. Definition of the functor $\mathcal{A}\langle_\rangle : \mathbf{TGTS}^{imp} \to \mathbf{GTS}$:
   - on objects: let $\mathcal{Z}$ be a T-GTS, then $\mathcal{A}\langle \mathcal{Z}\rangle = A_{\mathcal{Z}} = \langle T_s, \mathbf{wtProc}(\mathcal{Z}), \Pi_{\mathcal{Z}}\rangle$;
   - on morphisms: for any $f : \mathcal{Z}_1 \to \mathcal{Z}_2$, $\mathcal{A}\langle f\rangle : \mathcal{A}\langle \mathcal{Z}_1\rangle \to \mathcal{A}\langle \mathcal{Z}_2\rangle = \langle h_T, h_P\rangle$, where:
     - $h_T = \langle f_{T_V|V_{T_{1_S}}}, f_{T_E|E_{T_{1_S}}}\rangle$;
     - $\forall [\phi]_w \in \mathbf{wtProc}(\mathcal{Z}_1) \,.\, h_P([\phi]_w) = \widehat{f}_P([\phi]_w)$.
3. To prove that $\mathcal{I}\langle_\rangle$ is left-adjoint to $\mathcal{A}\langle_\rangle$ (as depicted in Figure 3.17) we have to show

   $$\forall \mathcal{G} \in \mathbf{GTS}, \forall \mathcal{Z} \in \mathbf{TGTS}^{imp}, \forall f : \mathcal{I}\langle \mathcal{G}\rangle \to \mathcal{Z} \,.\, \exists! h : \mathcal{G} \to \mathcal{A}\langle \mathcal{Z}\rangle \,.\, \epsilon_{\mathcal{Z}} \circ \mathcal{I}\langle h\rangle = f$$

   For each T-GTS $\mathcal{Z}$, we define the component at $\mathcal{Z}$ of the counit $\epsilon_{\mathcal{Z}} : \mathcal{I}\langle \mathcal{A}\langle \mathcal{Z}\rangle\rangle \to \mathcal{Z}$. This is an implementation morphism, thus a T-GTS morphism $\epsilon_{\mathcal{Z}} : \mathcal{I}\langle \mathcal{A}\langle \mathcal{Z}\rangle\rangle \to \widehat{\mathcal{Z}}$ is defined as follows:

   - $\epsilon_{\mathcal{Z}T} = T_s \hookrightarrow T$;

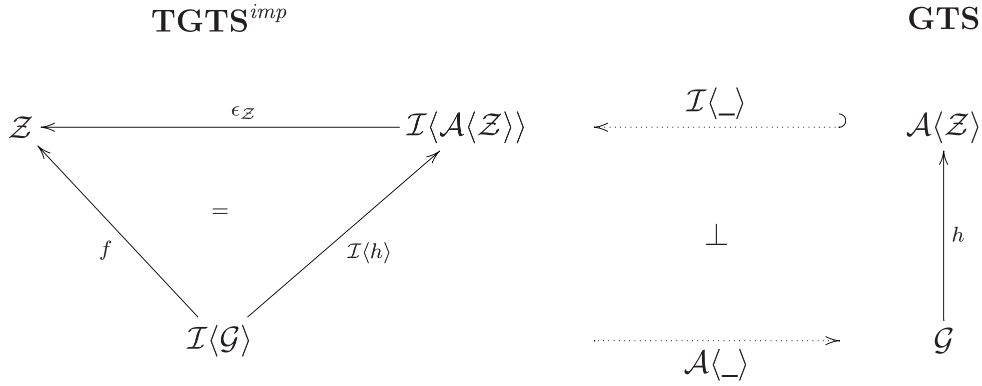


Figure 3.17: Universality of $\epsilon_{\mathcal{Z}}$ in $\mathbf{TGTS}^{imp}$.

- $\forall[\phi]_w \in \mathbf{wtProc}(\mathcal{Z})\centerdot \epsilon_{\mathcal{Z}P}([\phi]_w) = [\phi]_w$ (remember that productions in $\mathcal{A}\langle\mathcal{Z}\rangle$ are exactly the wt-processes of $\mathcal{Z}$).

It remains to show that given a GTS $\mathcal{G}$ and a T-GTS $\mathcal{Z}$, for each implementation morphism $f : \mathcal{I}\langle\mathcal{G}\rangle \to \mathcal{Z}$, there is a unique $h : \mathcal{G} \to \mathcal{A}\langle\mathcal{Z}\rangle$ such that $\epsilon_{\mathcal{Z}} \circ \mathcal{I}\langle h\rangle = f$.

(a) Definition of (GTS morphism) $h : \mathcal{G} \to \mathcal{A}\langle\mathcal{Z}\rangle$.

  - $h_T = f_T$ (since f_T maps the type graph of $\mathcal{G}$ into stable items of $\mathcal{Z}$);
  - $\forall p \in P\centerdot h_P(p) = f_P(p)$ (since productions in $\mathcal{A}\langle\mathcal{Z}\rangle$ are exactly the wt-processes of $\mathcal{Z}$ and f maps each production of $\mathcal{G}$ to a wt-process of $\mathcal{Z}$).

(b) $\epsilon_{\mathcal{Z}} \circ \mathcal{I}\langle h\rangle = f$.

  - $\epsilon_{\mathcal{Z}T} \circ \mathcal{I}\langle h_T\rangle = f_T$.
  By definition, $\epsilon_{\mathcal{Z}T} = \iota_{T_{\mathcal{Z}_s}}$ and $h_T = f_T$. Since $\mathcal{I}\langle h_T\rangle$ is the restriction of h_T to stable items and f_T relates only stable items (the type graph of $\mathcal{I}\langle\mathcal{G}\rangle$ is totally stable), then $\mathcal{I}\langle h_T\rangle = f_T$. Therefore, $\epsilon_{\mathcal{Z}T} \circ \mathcal{I}\langle h_T\rangle = \iota_{T_{\mathcal{Z}_s}} \circ f_T$. Since $\iota_{T_{\mathcal{Z}_s}}$ is the inclusion $T_{\mathcal{Z}s} \hookrightarrow T_{\mathcal{Z}}$ and f_T relates only stable items, then $\iota_{T_{\mathcal{Z}_s}} \circ f_T = f_T$. Therefore $\epsilon_{\mathcal{Z}T} \circ \mathcal{I}\langle h_T\rangle = f_T$.
  - $\widehat{\epsilon_{\mathcal{Z}P}} \circ \mathcal{I}\langle h_P\rangle = f_P$.
  Since all productions of $\mathcal{I}\langle\mathcal{A}\langle\mathcal{Z}\rangle\rangle$ are totally stable (they are all transactions of $\mathcal{Z}$), then $\mathcal{I}\langle\mathcal{A}\langle\mathcal{Z}\rangle\rangle$ have one wut-processes $[\phi_{id_{[\phi]_w}}]_w$ for each production $[\phi]_w$ (see the observation before Proposition 3.5). Moreover, by definition of $\widehat{\epsilon_{\mathcal{Z}P}}$, we have $\widehat{\epsilon_{\mathcal{Z}P}}([\phi_{id_{[\phi]_w}}]_w) = [\phi]_w$ By definition, $h_P = f_P$ and $\mathcal{I}\langle h_P\rangle$ maps each production $p \in P_{\mathcal{G}}$ to wut-process $[\phi_{id_{f_P(p)}}]_w$, then $\forall p \in P_{\mathcal{G}}\centerdot \widehat{\epsilon_{\mathcal{Z}P}}(\mathcal{I}\langle h_P\rangle(p)) = \widehat{\epsilon_{\mathcal{Z}P}}([\phi_{id_{f_P(p)}}]_w) = f_P(p)$.

(c) h is unique. Let us suppose that exists $u = \langle u_T, u_P\rangle \neq h$, such that $\epsilon_{\mathcal{Z}} \circ \mathcal{I}\langle u\rangle = f$:

  - Since u_T (it is a GTS morphism) relates only stable items and by definition $\mathcal{I}\langle u_T\rangle$ is the restriction of u_T to stable items, then $\mathcal{I}\langle u_T\rangle = u_T$. Since $\epsilon_{\mathcal{Z}T}$ is the inclusion $T_{\mathcal{Z}s} \hookrightarrow T_{\mathcal{Z}}$ and u_T relates only items in $T_{\mathcal{Z}s}$, then $\epsilon_{\mathcal{Z}T} \circ \mathcal{I}\langle u_T\rangle = u_T = f_T$. Since $h_T = f_T$, then $u_T = h_T$;
  - By definition of $\mathcal{I}\langle_\rangle$, $\forall p \in P_{\mathcal{G}}\centerdot \mathcal{I}\langle u_P\rangle(p) = [\phi_{id_{u_P(p)}}]_w$, and, by definition of $\widehat{\epsilon_{\mathcal{Z}P}}$, we have $\widehat{\epsilon_{\mathcal{Z}P}}([\phi_{id_{[\phi]_w}}]_w) = [\phi]_w$. Therefore, $\forall p \in P_{\mathcal{G}}\centerdot \widehat{\epsilon_{\mathcal{Z}P}}(\mathcal{I}\langle u_P\rangle(p)) = \widehat{\epsilon_{\mathcal{Z}P}}([\phi_{id_{u_P(p)}}]_w) = u_P(p) = f_P(p)$. Since $\forall p \in P_G\centerdot h_P(p) = f_P(p)$, then $\forall p \in P_G\centerdot u_P(p) = h_P(p)$.

$\square$

Proving that $\mathcal{A}\langle_\rangle$ is the right adjoint of $\mathcal{I}\langle_\rangle$ we have that the abstract system as defined in Definition 3.18 is the best approximation of a T-GTS when the unstable items are forgotten. Moreover, if we consider T-GTSs from an abstract point of view, a T-GTS $\mathcal{Z}$ has the same behaviour of $\mathcal{A}\langle\mathcal{Z}\rangle$, i.e., the transactions of both systems are isomorphic.

4 TRANSACTIONAL GRAPH TRANSFORMATION SYSTEMS WITH DEPENDENCY RELATION

In this section, we present an extension of T-GTSs to include a dependency relation in the productions. This dependency gives us an extra information about the relationship between the deleted and created elements in each production. The dependency information can be used to determine implementations (by transactions) of productions, restricting the set of possible valid implementations. A first approach was published in (FOSS; MACHADO; RIBEIRO, 2007).

When a production is applied, we can observe a total relation between the created elements and the consumed/preserved ones, i.e., all elements are created because of the existence of all the elements (consumed or preserved) in the left-hand side of the production. However, if we want to associate to a production an implementation composed by many productions applied in a specific way, we should be able to specify the relation between the elements of left- and right-hand sides in a more sophisticated way. For example, if we consider the production STOP depicted in Figure 3.1, we can have different transactions implementing it. In Figure 4.1, two implementations for production STOP are shown. In first transaction (at the top), when the pump finishes supplying gas, it is freed before operator to verify the charge of supplied gas. Thus, $Free$ flag of pump and `Charge` message depend on both $Busy$ flag of pump and `Finish` message. In the second transaction (at the bottom), when the pump finishes supplying gas, the charge of supplied gas can be verified even if the pump is not free. Thus, `Charge` message depends only on `Finish` message and $Free$ flag of pump depends on both `Finish` message and $Busy$ flag. Both transactions implement STOP production, since their minimal and maximal graphs are equal to left- and right-hand sides of this production, respectively, but the dependencies of $Free$ and `Charge` in their maximal graphs are different. However, if we want to specify that only one of them is a desirable implementation of the STOP production, we need a mechanism to describe these different dependencies. Therefore, we will add to each production a relation describing the desirable dependency between its created and consumed/preserved items that must be considered when the production is implemented.

Since we want to use productions with dependencies, called dep-productions, to model a transactions in an abstract way, the dependency relation associated to a production must satisfy some restrictions, such that a transaction implementing this production exists, i.e., the transaction must produce the same observable effect, but in more steps and possibly using unobservable items. Therefore, the dependency relation of a production must allow to decompose it in a set of productions, which together constitute a transaction. The first two restrictions of the next definition assure us that this set of productions exists: (1) the

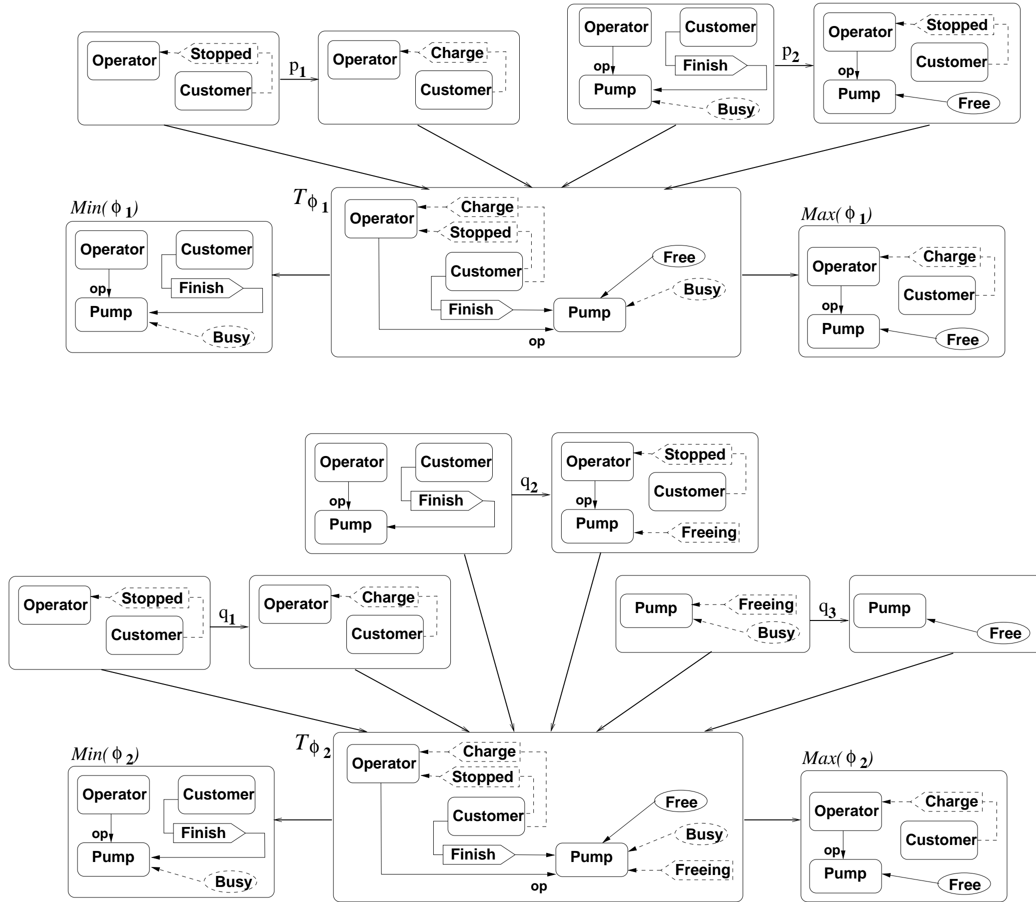


Figure 4.1: Transactions implementing STOP production depicted in Figure 3.1.

left- and right-hand sides and the interface of the productions are graphs, i.e., all edges in these graphs have the source and the target vertices; (2) all productions are consuming, i.e., all of them consume something. The next restriction guarantees that these productions compose a transaction: (3) all observable created items cannot be used (deleted or preserved) in the transaction. In order to define the latter restriction, we need to get the (weakly) connected component (DIESTEL, 2005) of the graph created by a production (i.e. the greatest subgraph in the right-hand side of the production containing only created elements). These components must be created in the same time, otherwise it would be necessary to use the vertices created within the transaction to create the edges.

We define $\mathcal{C}(G)$ as the set of connected components of G (disregarding the directions of the edges).

Definition 4.1 (dependency relation) *Given a T-typed production $p : L_p \overset{l}{\hookleftarrow} K_p \overset{r}{\rightarrow} R_p$. A* dependency relation *$\prec_p$ for p is a relation over $L_p \times (R_p - K_p)$, such that*

1. *each created item b depending on a preserved or consumed edge must depend on its target and source vertices, i.e.,*

$$\forall e \in E_{L_p} \centerdot \forall b \in R_p \centerdot e \prec_p b \Rightarrow t^{L_p}(e) \prec_p b \wedge s^{L_p}(e) \prec_p b$$

2. (a) *all created items that depends on a preserved one must depend on a deleted item as well, i.e.,*

$$\forall b \in R_p \centerdot \forall a \in K_p \centerdot a \prec_p b \Rightarrow \exists a' \in (L_p - K_p) \wedge a' \prec_p b$$

(b) *if there exists a preserved item that is not related to anything, then there exists a deleted item that is not related to anything as well, i.e.,*

$$\exists a \in K_p \,.\, \forall b \in R_p \,.\, a \not\prec_p b \Rightarrow \exists a' \in (L_p - K_p) \,.\, \forall b \in R_p \,.\, a' \not\prec_p b$$

3. *all the items of a connected component of the graph created by the production (all created items of the production, excluding the created edges having either source or target preserved), must have the same dependency, i.e.,*

$$\forall G \in \mathcal{C}(R'_p) \,.\, \forall b, b' \in G \wedge a \in L_p \,.\, a \prec_p b \Leftrightarrow a \prec_p b',$$

where $R'_p = \langle V_{R'_p}, E_{R'_p}, s^{R'_p}, t^{R'_p} \rangle$,
$V_{R'_p} = (V_{R_p} - V_{K_p}) \cup$
$\{v | v \in V_{K_p} \wedge \exists e \in E_{R'_p} \,.\, (s^{R_p}(e) = v \vee t^{R_p}(e) = v)\}$,
$E_{R'_p} = \{e | e \in (E_{R_p} - E_{K_p}) \wedge (s^{R_p}(e) \notin V_{K_p} \vee t^{R_p}(e) \notin V_{K_p})\}$ *and*
$\forall e \in E_{R'_p} \,.\, s^{R'_p}(e) = s^{R_p}(e) \wedge t^{R'_p}(e) = t^{R_p}(e)$

By the three conditions of definition above, we assure that it is possible to have a transaction implementing a production with an associated dependency relation, where this relation must be preserved and reflected, i.e., the relation is the same in both production and transaction. Considering a production p with a dependency relation, condition 1, requires that all elements which depend on an edge, depend on their source and target vertices, too. This assures that we can define productions that consume/preserve edges of p using their source and target vertices. Condition 2 (a and b), assures that the productions in the transaction are all consuming, by requiring that for all preserved items, there exists one deleted item with the same dependency (avoiding productions that preserve but do not consume something). Condition 3 assures that no created observable item is used in the transaction where it was created. This is done by requiring that all items in each connected component created by a production have the same dependency, which forces them to be all created at the same time (in the same production) (see Remark 3.1). This restriction avoids the need to read (preserve) vertices in order to create edges over them.

Definition 4.2 (*dep*-production) *A* dep-production *is a tuple* $\langle L_p \hookleftarrow K_p \rightarrow R_p, \prec_p \rangle$, *where* $L_p \hookleftarrow K_p \rightarrow R_p$ *is the span of production* p *and* $\prec_p$ *is a dependency relation for* p. *The class of all* T*-typed* dep*-production is denoted by* T-$\mathbf{DProd}$.

Example 4.1 (*dep*-production) *If we consider that the* STOP *production (Figure 3.1) can be partitioned into more steps, as depicted at the bottom of Figure 4.1, we will obtain the* dep*-production in Figure 4.2. The dependency is defined by the letters* x, y, z, b, d *and* h, *relating each created element (in the right-hand side) with the set of all consumed or read elements (in the left-hand side) on which it depends.*

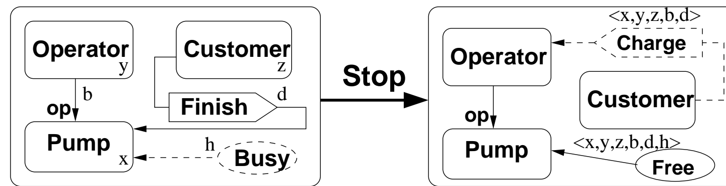


Figure 4.2: A dep-production.

Then the corresponding dependency relation associated to the STOP *production is:*

$$\begin{array}{lll}
\texttt{op} & \prec_{Stop} & \texttt{Free} \\
\texttt{Pump} & \prec_{Stop} & \texttt{Free} \\
\texttt{Operator} & \prec_{Stop} & \texttt{Free} \\
\texttt{Customer} & \prec_{Stop} & \texttt{Free} \\
\texttt{Finish} & \prec_{Stop} & \texttt{Free} \\
\texttt{Busy} & \prec_{Stop} & \texttt{Free}
\end{array}
\qquad
\begin{array}{lll}
\texttt{op} & \prec_{Stop} & \texttt{Charge} \\
\texttt{Pump} & \prec_{Stop} & \texttt{Charge} \\
\texttt{Operator} & \prec_{Stop} & \texttt{Charge} \\
\texttt{Custumer} & \prec_{Stop} & \texttt{Charge} \\
\texttt{Finish} & \prec_{Stop} & \texttt{Charge}
\end{array}$$

where the `Free` *attribute was created because both* `Busy` *and* `Finish` *are consumed, and the* `Charge` *message was created only because the* `Finish` *message was consumed; the preserved (or read) elements are $Pump$, $Operator$, $Customer$ and op (for both* `Free` *and* `Charge`*).*

┘

A T-GTS with dep-productions is defined as a T-GTS where the π function maps each production name into a dep-production.

Definition 4.3 (GTS and T-GTS with dependency relation) *A* graph transformation system with dependency relation *(d-*GTS*) is a* GTS $\langle T, P, \pi\rangle$*, where π is a total function mapping production names to dep-productions in T-**DProd**.*

A transactional graph transformation system with dependency relation *(d*T-GTS*) is a* T-GTS $\langle\langle T, P, \pi\rangle, T_s\rangle$*, where $\langle T, P, \pi\rangle$ is a d-*GTS*.*

Example 4.2 (T-GTS with dependency relation) *As an example of d*T-GTS*, we can consider the gas station system of Example 3.1 adding dependency relations to productions. The resulting d*T-GTS $\mathcal{D}epPumpOper$ *is shown in Figure 4.3, where all of them, but* STOP*, have total dependencies, i.e., all created elements depend on all deleted and preserved ones. The type graph is the same shown in Figure 3.1. The dependencies restrict implementations of these productions: for example,* STOP *can be implemented by transaction ϕ_2 in Figure 4.1 (at the bottom), but not by ϕ_1 in the same figure. The choice of a total dependency, for example, for the* SERVE *production, establishes that the $Busy$ flag must be created only if both* `Supply` *message and $Free$ flag are consumed, even if this consumption and creation are done in several small steps.*

┘

The dependency relation of productions does not interfere in the semantics of a T-GTS. The match is not restricted by dependency, it only gives an extra syntactic information about possible refinement or implementation. Therefore, the semantics of dT-GTSs are given by direct derivations and derivations like in Definition 2.4, substituting productions by dep-productions.

Definition 4.4 (direct derivations and derivations of dT-GTSs) *Given a T-typed graph G, a T-typed graph dep-production $q = \langle L_q \xleftarrow{l} K_q \xrightarrow{r} R_q, \prec_q\rangle$ and a match (i.e. an injective T-typed graph morphism) $m : L_q \to G$, a* direct derivation *from G to H using q (based on m) exists if and only if the diagram below can be constructed, where both squares are pushouts in T-**Graph**. In this case the direct derivation is denoted by $\delta : G \overset{q,m}{\Rightarrow} H$ or $\delta : G \overset{q}{\Rightarrow} H$ if we do not make explicit m.*

$$\begin{array}{ccccc}
L_q & \xleftarrow{l} & K_q & \xrightarrow{r} & R_q \\
m\downarrow & (1) & k\downarrow & (2) & \downarrow m^* \\
G & \xleftarrow[l^*]{} & D & \xrightarrow[r^*]{} & H
\end{array}$$

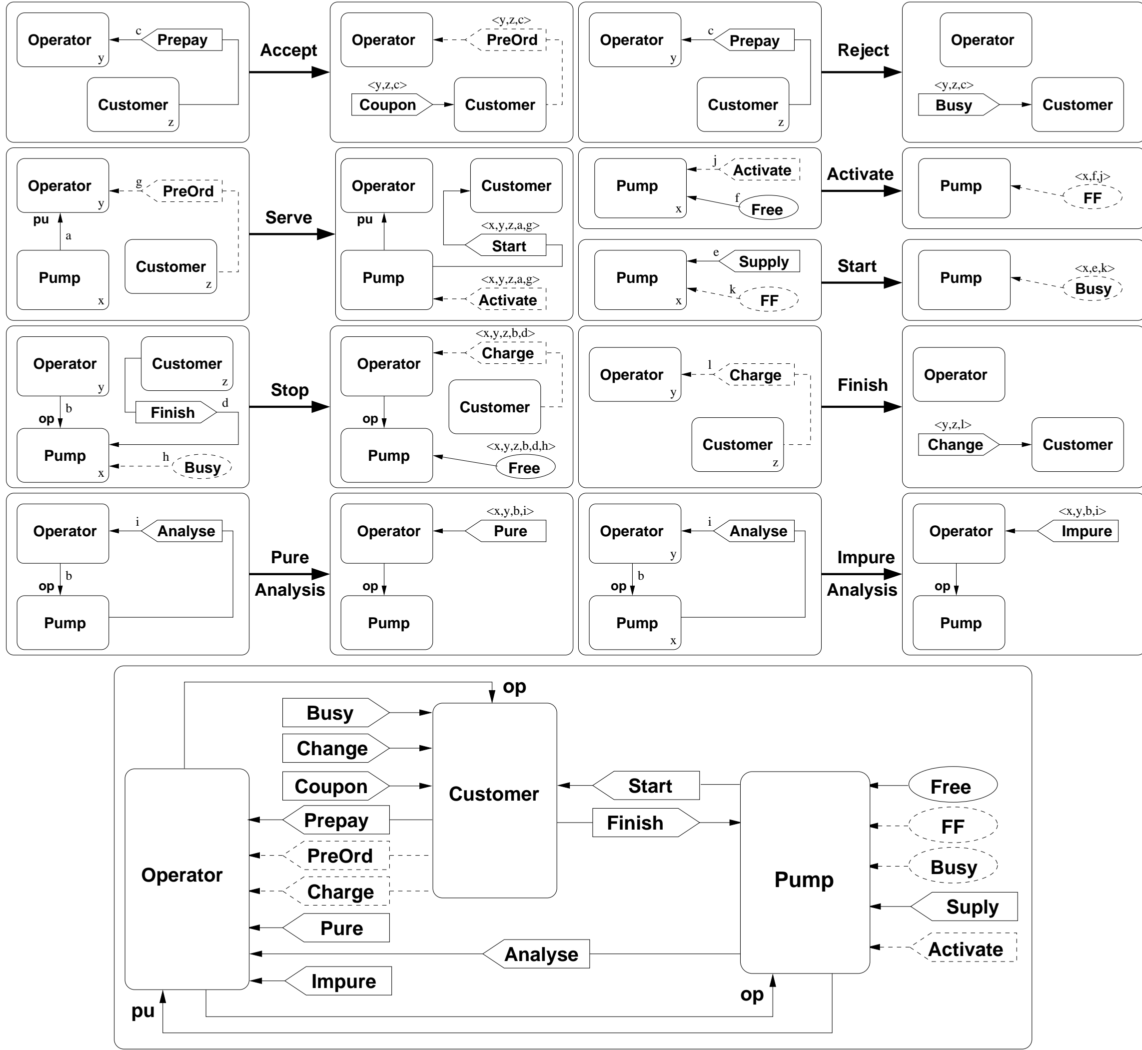


Figure 4.3: A T-GTS $\mathcal{D}epPumpOper$ with dependency relation for gas station system.

Given a dT-GTS $\mathcal{Z} = \langle\langle T, P, \pi\rangle, T_s\rangle$, *a* derivation $\rho : G_0^T \stackrel{p_1,m_1}{\Rightarrow} G_1^T \stackrel{p_2,m_2}{\Rightarrow} G_2^T \cdots$ *of* $\mathcal{Z}$ *is a finite or infinite of direct derivations* $\delta_i : G_i^T \stackrel{p_i,m_i}{\Rightarrow} H_i^T$, *where* $G_{i+1}^T = H_i^T$ *and* $i \geq 0$. *If a derivation* $\rho : G_0^T \stackrel{p_1,m_1}{\Rightarrow} \cdots \stackrel{p_n,m_n}{\Rightarrow} G_n^T$ *is finite we call* G_0^T *and* G_n^T *of initial and final graphs, respectively. The semantics of* $\mathcal{Z}$ *is the class of all derivations in* $\mathcal{Z}$, *denoted by* $\mathbf{dDer}(\mathcal{Z})$.

4.1 Categories dGTS and dTGTS

If we want to relate two d-GTSs we need to consider the dependency relation associated to their productions. Therefore, the production mapping must respect this relation, i.e., a production can be mapped only into another if the target one has the same dependency relation between the translated elements. This restriction reduces the possible relationships between d-GTSs, with respect to GTSs, because we impose an extra restriction, excluding same relationships valid for GTSs.

Definition 4.5 (d-GTS morphism) *Given two d-*GTS*s* $\mathcal{G}_1$ *and* $\mathcal{G}_2$. *A* d-GTS morphism $f : \mathcal{G}_1 \rightarrow \mathcal{G}_2$ *is a* GTS *morphism between* $\mathcal{G}_1$ *and* $\mathcal{G}_2$ *such that the production mapping respects dependency relations, i.e., for all* $p_1 \in P_1$ *and for each concrete choice of* $f_\iota(p_1)$, *with*

$f_P(p_1) = p_2$, *we have that:* $\forall a_1, b_1 \in P_1 \centerdot \forall a_2, b_2 \in P_2 \centerdot (f_\iota^L(p_1)(a_2) = a_1 \wedge f_\iota^R(p_1)(b_2) = b_1) \Rightarrow ((a_1 \prec_{p_1} b_1) \Leftrightarrow (a_2 \prec_{p_2} b_2))$.

Proposition 4.1 *d-*GTS*s and d-*GTS *morphisms form a category, denoted by* $\mathbf{dGTS}$*, in which composition and identities are defined as in* $\mathbf{GTS}$.

Proof:

1. *Composition is well-defined*: Let $f : \mathcal{G}_1 \to \mathcal{G}_2$ and $g : \mathcal{G}_2 \to \mathcal{G}_3$ be d-GTS morphisms and $g \circ f$ be their composition. By Proposition 2.2, we have that $g \circ f$ is a GTS morphism, then it remains to prove that $g_P \circ f_P$ respects dependency relations, i.e., for all $p_1 \in P_1$ and for each concrete choice of $f_\iota(p_1) \circ g_\iota(p_2)$, with $f_P(p_1) = p_2$ and $g_P(p_2) = p_3$, we have that: $\forall (a_1 \prec_{p_1} b_1) \centerdot \forall (a_3 \prec_{p_3} b_3) \centerdot f_\iota^L(p_1)(g_\iota^L(p_2)(a_3)) = a_1 \wedge f_\iota^R(p_1)(g_\iota^R(p_2)(b_3)) = b_1 \centerdot (a_1 \prec_{p_1} b_1) \Leftrightarrow (a_3 \prec_{p_3} b_3)$. By definition the following statements hold:

   (a) $\forall (a_1 \prec_{p_1} b_1) \centerdot \forall (a_2 \prec_{p_2} b_2) \centerdot f_\iota^L(p_1)(a_2) = a_1 \wedge f_\iota^R(p_1)(b_2) = b_1 \centerdot (a_1 \prec_{p_1} b_1) \Leftrightarrow (a_2 \prec_{p_2} b_2)$;

   (b) $\forall (a_2 \prec_{p_2} b_2) \centerdot \forall (a_3 \prec_{p_3} b_3) \centerdot g_\iota^L(p_2)(a_3) = a_2 \wedge g_\iota^R(p_2)(b_3) = b_2 \centerdot (a_2 \prec_{p_2} b_2) \Leftrightarrow (a_3 \prec_{p_3} b_3)$.

   By (a) and (b), we have that:

   $$\begin{aligned} &\forall (a_1 \prec_{p_1} b_1) \centerdot \forall (a_2 \prec_{p_2} b_2) \centerdot \forall (a_3 \prec_{p_3} b_3) \centerdot \\ &\quad f_\iota^L(p_1)(a_2) = a_1 \wedge f_\iota^R(p_1)(b_2) = b_1 \wedge g_\iota^R(p_2)(b_3) = b_2 \wedge g_\iota^R(p_2)(b_3) = b_2 \centerdot \\ &\quad (a_1 \prec_{p_1} b_1) \Leftrightarrow (a_2 \prec_{p_2} b_2) \wedge (a_2 \prec_{p_2} b_2) \Leftrightarrow (a_3 \prec_{p_3} b_3) \end{aligned}$$

   Therefore,

   $$\begin{aligned} &\forall (a_1 \prec_{p_1} b_1) \centerdot \forall (a_3 \prec_{p_3} b_3) \centerdot \\ &\quad f_\iota^L(p_1)(g_\iota^L(p_2)(a_3)) = a_1 \wedge f_\iota^R(p_1)(g_\iota^R(p_2)(b_3)) = b_1 \centerdot \\ &\quad (a_1 \prec_{p_1} b_1) \Leftrightarrow (a_3 \prec_{p_3} b_3) \end{aligned}$$

2. *Identities are well-defined morphisms*: Let $id_\mathcal{G} = \langle id_T, id_P \rangle$ be the identity on $\mathcal{G} = \langle T, P, \pi \rangle$. By definition of identities in $\mathbf{GTS}$, for all $p \in P \centerdot id_P(p) = p$ and $id_\iota(p) = \langle id_{L_p}, id_{K_p}, id_{R_p} \rangle$. By definition of identities in T-$\mathbf{Graph}$, we have

   $$\forall (a \prec_p b) \centerdot id_{L_p}(a) = a \wedge id_{R_p}(b) = b \centerdot (a \prec_p b) \Leftrightarrow (a \prec_p b)$$

3. Neutrality of identity and associativity of composition follow from these properties in $\mathbf{GTS}$.

□

As defined for T-GTSs, dT-GTS morphisms are d-GTS morphisms that, preserves stable and unstable items, and are total on unstable ones.

Definition 4.6 (dT-GTS morphism) *Let* $\mathcal{Z}_1 = \langle \mathcal{G}_1, T_{1s} \rangle$ *and* $\mathcal{Z}_2 = \langle \mathcal{G}_2, T_{2s} \rangle$ *be d*T-GTS*s. A* dT-GTS morphism $f : \mathcal{Z}_1 \to \mathcal{Z}_2$ *is a d-*GTS *morphism* $f : \mathcal{G}_1 \to \mathcal{G}_2$ *between the underlying d-*GTS*s, such that*

1. *for all* $z \in T_1 \setminus T_{1s}$*, we have that* $f_T(z)$ *is defined and* $f_T(z) \in T_2 \setminus T_{2s}$*;*
2. *for all* $z \in T_{1s}$*, if* $f_T(z)$ *is defined then* $f_T(z) \in T_{2s}$.

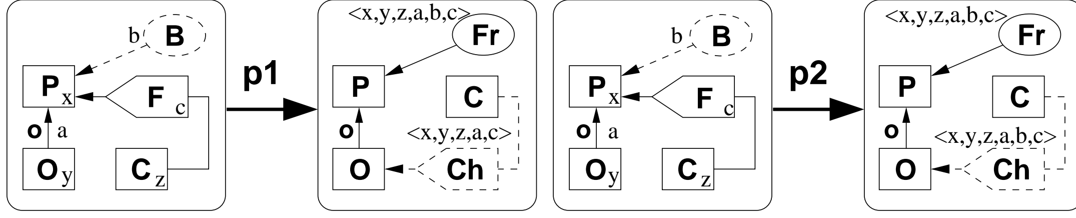


Figure 4.4: dep-productions p_1 and p_2.

Example 4.3 (dT-GTS morphism) *The dep-production* STOP*, defined in Example 4.1, could be mapped by a d*T-GTS *morphism f to the production p_1 in Figure 4.4, but not to p_2, because p_2 has a different dependency relation. We consider the mapping $f_\iota(\text{STOP})$ defined by $\{P \mapsto Pump, O \mapsto Operator, C \mapsto Customer, B \mapsto Busy, F \mapsto Finish,$ $Fr \mapsto Free, Ch \mapsto Charge, o \mapsto op\}$ for both p_1 and p_2.*

⌟

Proposition 4.2 *d*T-GTS*s and d*T-GTS *morphisms form a category, denoted by* **dTGTS***, in which composition and identities are defined as in* **dGTS**.

Proof: This proof follows from Propositions 4.1 and 3.2. □

Since the introduction of dependencies does not modify the semantics of a graph transformation system and graph processes are obtained from derivations, the graph processes of a dT-GTS are obtained as in Definition 3.10, substituting productions in graph processes by dep-productions, i.e. $\pi_\phi(\langle q_i, i\rangle) = \langle\langle L_i, c_{L_i}\rangle \overset{l_i}{\hookleftarrow} \langle K_i, c_{K_i}\rangle \overset{r_i}{\rightarrow} \langle R_i, c_{R_i}\rangle, \prec_{q_i}\rangle$, where $\pi_{\mathcal{Z}}(q_i) = \langle L_i \hookleftarrow K_i \rightarrow R_i, \prec_{q_i}\rangle$. Moreover, the transactional process of a dT-GTS is defined like in Definition 3.15 anddT-GTS morphisms preserve transactions as T-GTS morphisms.

4.2 Abstract d-GTS for a T-GTS with dependency relation

Since we are considering dep-productions, abstract transactions must consider the dependencies of their productions. Here, we will distinguish weak ut-processes with different relation dependencies. The relation dependency of a process can be obtained as the transitive closure of dependencies of their productions. Moreover, at a more abstract level, we only see the minimal and maximal graphs of a transaction, therefore, we will consider only the dependencies on elements in these graphs.

Definition 4.7 (Dependency relation of a process) *Given a process ϕ of a d*T-GTS *$\mathcal{Z}$, the* dependency relation of a process *ϕ is the relation $\prec_\phi$ over ${}^\bullet\phi \times \phi^\bullet - ({}^\bullet\phi \cap \phi^\bullet)$, defined by*

$$\prec_\phi = \preceq_{P_\phi} \cap\, R,$$

where $\preceq_{P_\phi}$ is the transitive closure of $\bigcup_{\forall p \in P_\phi} \prec_p$ and $R = \{(a, b)\ |a \in {}^\bullet\phi\ \wedge\ b \in \phi^\bullet\}$.

The transitive closure of dependency relations of productions of a graph process give us the dependency resulting of application of these productions in the associated derivation. For example, by two productions p and q of a process ϕ, if in p we have $a \prec_p b$ (a is consumed/preserved in order to create b) and in q we have $b \prec_q c$ (b is cosumed/preserved

in order to create c), we will have in the dependency relation of ϕ $a \prec_\phi c$ (a is consumed/preserved in order to create c). The intersection of $\preceq_{P_\phi}$ with R (set of pairs of elements in minimal and maximal graphs of ϕ) give us the restriction of the transitive closure of dependency relations of productions in ϕ to pairs of elements in minimal and maximal graphs for ϕ.

Example 4.4 (Dependency relation of a process) *Figure 4.5 shows the transactional process ϕ_1 of d*T-GTS *presented in Example 4.2.*

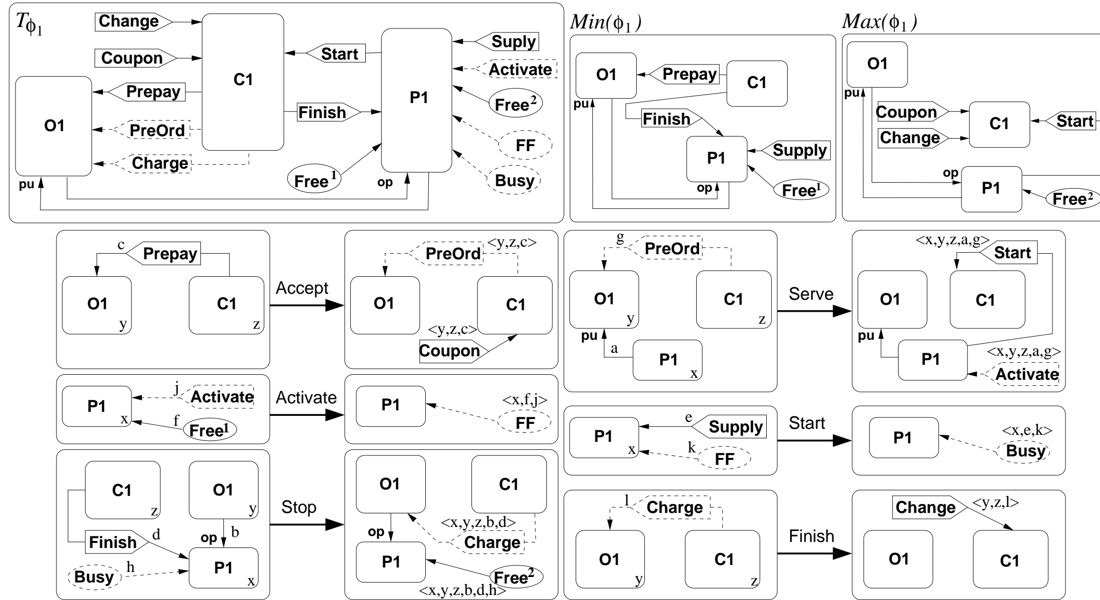


Figure 4.5: Transactional process of dT-GTS $\mathcal{D}epPumpOper$.

The dependency relations of all productions of ϕ_1 are illustrated in Table 4.1. The transitive closure of these relations is shown in Table 4.2. The transitive closure includes in a dependency relation a pair $a \preceq c$ if the pairs $a \preceq b$ and $b \preceq c$ are already in the relation. In the example, we have that operator accepts a prepayment only if there is some prepayment being offered it him (`Prepay` $\prec_{\text{ACCEPT}}$ `PreOrd`); moreover, we have that activation of pump is triggered only if the operator has accepted a prepayment (`PreOrd` $\prec_{\text{SERVE}}$ `Activate`). Consequently, we will have that the activation of the pump is triggered only if someone has offered a prepayment (`Prepay` $\preceq_{P_{\phi_1}}$ `Activate`). Besides, the pump is activated only if the activation has been triggered (`Activate` $\prec_{\text{ACTIVATE}}$ FF). Therefore, pump is activated only if a prepayment has been offered (`Prepay` $\preceq_{P_{\phi_1}}$ FF). Finally, Table 4.3 shows the restriction to minimal and maximal graphs of ϕ_1 of the relation shown in Table 4.2. This restriction eliminates from the dependency relation of ϕ all pairs containing elements that are not in minimal or maximal graphs. For example, the pair `Charge` $\preceq_{P_{\phi_1}}$ `Change` is eliminated since `Charge` is not in minimal neither in maximal graphs.

┘

Now, we can define the class of abstract transactions considering the dependency relation. A dependency weak ut-process is a class of weak equivalent processes that have the same dependency relation.

Table 4.1: Dependency relations of productions of transactional process ϕ_1.

O1 C1 Prepay	$\prec_{\text{ACCEPT}}$	Preord	O1 C1 Prepay	$\prec_{\text{ACCEPT}}$	Coupon
O1 P1 C1 pu Preord	$\prec_{\text{SERVE}}$	Start	O1 P1 C1 pu Preord	$\prec_{\text{SERVE}}$	Activate
P1 $Free^1$ Activate	$\prec_{\text{ACTIVATE}}$	FF	P1 FF Supply	$\prec_{\text{START}}$	$Busy$
O1 P1 C1 op $Busy$ Finish	$\prec_{\text{STOP}}$	$Free^2$	O1 P1 C1 op Finish	$\prec_{\text{STOP}}$	Charge
O1 C1 Charge	$\prec_{\text{FINISH}}$	Change			

Definition 4.8 (Dependency weak processes) *Let ϕ_1 and ϕ_2 be two weak equivalent ut-processes and $f_T : T_{\phi_1} \to T_{\phi_2}$ be the isomorphism between type graphs of the processes. Then, ϕ_1 and ϕ_2 are* dep-weak-equivalent*, written $\phi_1 \approx^d \phi_2$, if only if:*

$$\forall a, b \in {}^\bullet\phi_1 \cup \phi_1{}^\bullet \wedge \forall a', b' \in {}^\bullet\phi_2 \cup \phi_2{}^\bullet$$
$$\text{if} \quad f_T(a) = a' \wedge f_T(b) = b' \quad \text{then} \quad a \prec_{\phi_1} b \Leftrightarrow a' \prec_{\phi_2} b'$$

A dependency weak ut-process (dwut-process) *is defined as an equivalence class of ut-processes with respect to dep-weak-equivalence, denoted as $[\phi]_d$ for a representative ϕ. The set of dwut-processes of a* T-GTS *$\mathcal{Z}$ is denoted by $\mathbf{DwutProc}(\mathcal{Z})$. The set of* dep-weak t-processes (dwt-processes) $\mathbf{DwtProc}(\mathcal{Z})$ *is defined in an analogous way.*

We can obtain a *dep*-production associated to a transaction, considering their minimal and maximal graphs as left- and right-hand sides, and the intersection of them as interface of the production. The dependency relation of this production is given by the dependency relation associated to the transactional process. Thus, we obtain an abstract description (*dep*-production) of a transaction, and can see the transaction as an implementation of this abstract description.

Definition 4.9 (*dep*-production associated to a dwut-process) *Given a process ϕ for a d*T-GTS *$\mathcal{Z}$, we have*

$$\Pi(\phi) = \langle {}^\bullet\phi \hookleftarrow {}^\bullet\phi \cap \phi^\bullet \hookrightarrow \phi^\bullet, \prec_\phi \rangle,$$

where the intersection ${}^\bullet\phi \cap \phi^\bullet$ is taken componentwise and $\prec_\phi$ is the dependency relation associated to ϕ.
*Let us assume for each d*T-GTS *$\mathcal{Z}$ a choice function $\mathsf{ch}_\mathcal{Z}$, mapping each dwut-process $[\phi]_d$ to a concrete representative $\mathsf{ch}_\mathcal{Z}([\phi]_d) \in [\phi]_d$. The* *dep*-production associated to $[\phi]_d$ *is defined as $\Pi_\mathcal{Z}([\phi]_d) = \Pi(\mathsf{ch}_\mathcal{Z}([\phi]_d))$.*

Example 4.5 (*dep*-production associated to a process) *The dep-production associated to the process in Example 4.4 is shown in Figure 4.6. The associated dependency*

Table 4.2: Transitive closure of dependency relations of production of ϕ_1.

O1 C1 Prepay	$\preceq_{P_{\phi_1}}$	Preord	O1 C1 Prepay	$\preceq_{P_{\phi_1}}$	Coupon
O1 P1 C1 pu Prepay Preord	$\preceq_{P_{\phi_1}}$	Start	O1 P1 C1 pu Prepay Preord	$\preceq_{P_{\phi_1}}$	Activate
O1 P1 C1 pu $Free^1$ FF Prepay Preord Activate Supply	$\preceq_{P_{\phi_1}}$	$Busy$	O1 P1 C1 op pu $Free^1$ FF $Busy$ Prepay Preord Activate Supply Finish	$\preceq_{P_{\phi_1}}$	$Free^2$
O1 P1 C1 op Finish	$\preceq_{P_{\phi_1}}$	Charge	O1 P1 C1 op Finish Charge	$\preceq_{P_{\phi_1}}$	Change
O1 P1 C1 pu $Free^1$ Prepay Preord Activate	$\preceq_{P_{\phi_1}}$	FF			

Table 4.3: Dependency relation associated to transactional process ϕ_1.

O1 C1 Prepay	$\prec_{\phi_1}$	Coupon	O1 P1 C1 pu Prepay	$\prec_{\phi_1}$	Start
O1 P1 C1 op pu $Free^1$ Prepay Supply Finish	$\prec_{\phi_1}$	$Free^2$	O1 P1 C1 op Finish	$\prec_{\phi_1}$	Change

(Table 4.3) is described by the letters. This production describes an execution of the system in an abstract way and the dependency relation give us some information about the interaction of the system with its environment, for example, the system only sends a `Coupon` *or a* `Start` *message if it has received previously a* `Prepay` *message, and only sends a* `Change` *message after it has received a* `Prepay`, `Supplay` *and* `Finish` *messages.*

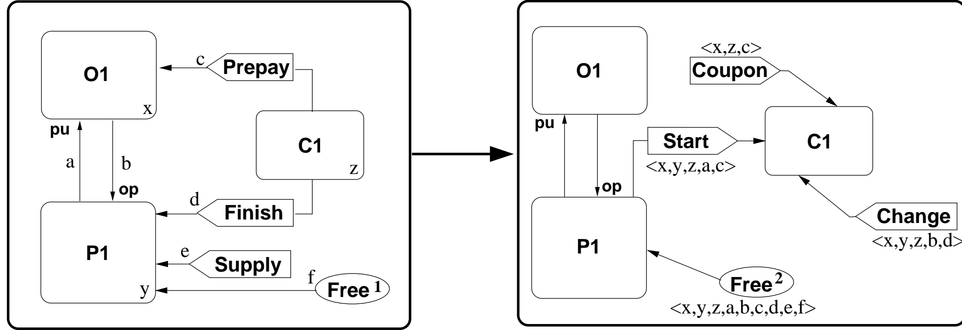


Figure 4.6: dep-production associated to the process in Example 4.4.

⌟

We are now able to define the abstract system associated with a dT-GTS. An *abstract system associated to a d*T-GTS $\mathcal{Z}$ is defined as for T-GTSs (Definition 3.18), where the set of productions names contains all dwt-processes of $\mathcal{Z}$.

Definition 4.10 (Abstract d-GTS) *Let* $\mathcal{Z} = \langle \mathcal{G}, T_s \rangle$ *be a d*T-GTS*. The* abstract d-GTS *associated to* $\mathcal{Z}$*, denoted by* $A_{\mathcal{Z}}$*, is the d-*GTS $\langle T_s, \mathbf{DwtProc}(\mathcal{Z}), \Pi_{\mathcal{Z}} \rangle$ *where* $\mathbf{DwtProc}(\mathcal{Z})$ *is the set of dwt-processes of* $\mathcal{Z}$ *and* $\Pi_{\mathcal{Z}}$ *is as in Definition 4.9.*

Example 4.6 (Abstract d-GTS) *In Figure 4.7, we can see the abstract system* $\mathcal{Z}_1$ *associated to d*T-GTS $\mathcal{D}ep\mathcal{P}ump\mathcal{O}per$ *(Example 4.2).*

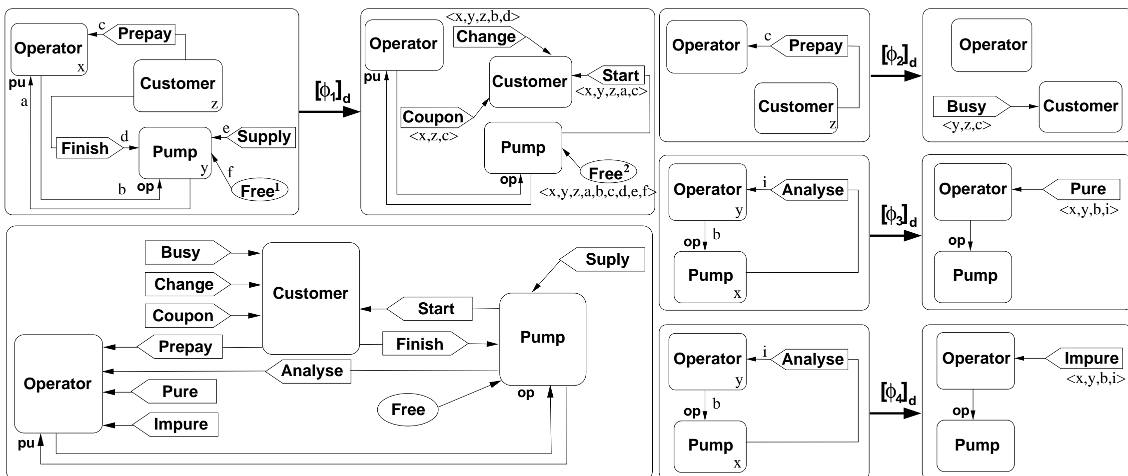


Figure 4.7: Abstract d-GTS $\mathcal{Z}_1$ associated to the dT-GTS $\mathcal{D}ep\mathcal{P}ump\mathcal{O}per$, described in Example 4.2.

As for the ordinary T-GTS*, the type graph of* $\mathcal{Z}_1$ *is the stable type graph of* $\mathcal{D}ep\mathcal{P}ump\mathcal{O}per$*. The productions of* $\mathcal{Z}_1$ *are all transactional process of the original d*T-GTS*:*

- *the process* $[\phi_1]_d$ *is shown in Figure 4.5;*
- *the processes* $[\phi_2]_d$, $[\phi_3]_d$ *and* $[\phi_4]_d$ *are those which have as the only productions* REJECT, PUREANALYSIS *and* IMPUREANALYSIS, *respectively.*

┘

4.3 Implementation morphism for dT-GTS

In this section we describe the relationship between a more abstract dT-GTS and its implementation. This definition is analogous to that for T-GTSs. Here, an implementation is given by a pair of mappings: (a) a morphism between the type graphs and (b) a function mapping each production of the source dT-GTS into a transaction of the target one. If we consider $\widehat{\mathcal{Z}_2}$ be the dT-GTS having the same type graph of $\mathcal{Z}_2$ and the dep-productions associated to all unstable transactional processes of $\mathcal{Z}_2$, we can define a implementation relationship between dT-GTSs $\mathcal{Z}_1$ and $\mathcal{Z}_2$ as a dT-GTS morphism, where the productions component associates each production of $\mathcal{Z}_1$ to a production of $\widehat{\mathcal{Z}_2}$, i.e., to a transaction of $\mathcal{Z}_2$.

Definition 4.11 (dT-GTS implementation morphisms) *Given* dT-GTS $\mathcal{Z}_i = \langle\langle T_i, P_i, \pi_i\rangle, T_{1s}\rangle$, *for* $i \in \{1, 2\}$. *Let* $\widehat{\mathcal{Z}_i} = \langle\langle T_i, \mathbf{DwutProc}(\mathcal{Z}_i), \Pi_{\mathcal{Z}_i}\rangle, T_{is}\rangle$ *be a* dT-GTS *having all dep-weak ut-processes of* $\mathcal{Z}_i$ *as productions. An* (dT-GTS) implementation morphism $f\colon \mathcal{Z}_1 \to \mathcal{Z}_2$ *is a* dT-GTS *morphism* $\langle f_T, f_P\rangle\colon \mathcal{Z}_1 \to \widehat{\mathcal{Z}_2}$.

Example 4.7 (implementation morphisms for dT-GTSs) *Consider the* dT-GTS $\mathcal{Z}_1$ *depicted in Figure 4.7. We can define an implementation morphism between* $\mathcal{Z}_1$ *and the* dT-GTS *of the gas station system (Example 4.2). The type mapping is given by the inclusion of* $T_{\mathcal{Z}_1}$ *into the type graph of the gas station system (Figure 3.1). The mapping between the productions is defined by:* $[\phi_1]_d$ *is mapped to the transaction in Example 4.4;* $[\phi_2]_d$, $[\phi_3]_d$ *and* $[\phi_3]_d$ *are mapped to transactions containing productions* REJECT, PUREANALYSIS *and* IMPUREANALYSIS, *respectively. Note that dependencies of all productions are respected, i.e., they have the same dependency of the transactions to which they are mapped.*

┘

In the following, $\mathbf{D}^{\prec}(a, a')$ is true if a and a' have the same dependencies in $\prec$, i.e., they are related with the same elements by $\prec$, that is

$$\mathbf{D}^{\prec}(a, a') = \begin{cases} \texttt{true} & \text{iff } \forall x\,\textbf{.}\,(a \prec x \Leftrightarrow a' \prec x) \text{ and } (x \prec a \Leftrightarrow x \prec a') \\ \texttt{false} & \text{otherwise.} \end{cases}$$

Proposition 4.3 *Given a* dT-GTS $\mathcal{Z}_i$, *let* $\widehat{\mathcal{Z}_i}$ *be as in Definition 4.11, for* $i \in \{1, 2\}$. *Then any* dT-GTS *morphism* $f\colon \mathcal{Z}_1 \to \widehat{\mathcal{Z}_2}$ *extends to a* dT-GTS *morphism* $\widehat{f}\colon \widehat{\mathcal{Z}_1} \to \widehat{\mathcal{Z}_2}$.

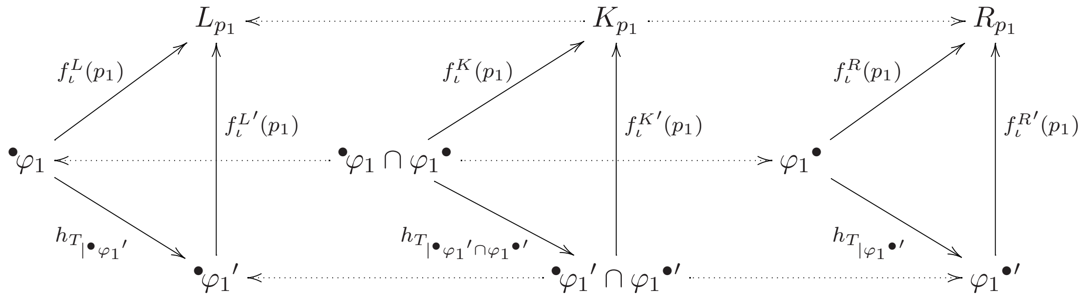
Proof: The definition of $\widehat{f}$ is given as in Proposition 3.5. Then it remains to prove that $\widehat{f}$ is well-defined. By Proposition 3.5, all concrete processes are weak equivalent, then it remains to prove that all concrete processes have the same dependency relation.

If we would chose a different concrete process φ_1' (dep-weak-equivalent to φ_1) in Proposition 3.5, we would obtain a different process φ', such that $\widehat{f}(\phi) = \varphi'$. By definition of $\widehat{f}$, we have that $P_\varphi = P_{\varphi'}$ and $T_\varphi \approx T_{\varphi'}$. Moreover, in order to obtain the dependency relation of φ, we can get the transitive closure of $\varphi_1 \cdots \varphi_n$, because the dependencies between elements of these processes is given only on those elements in maximal and minimal graphs.

By definition of dT-GTS implementation morphism, $\prec_{\varphi_1}$ and $\prec_{\varphi_1'}$ can differ only on elements having the same dependency, i.e.,

$$(1)\forall a \in {}^\bullet\varphi_1 \cup \varphi_1{}^\bullet \,.\, f_\iota^{A'}(p_1)(h_T(a)) \neq f_\iota^A(p_1)(a) \Rightarrow \mathbf{D}^{\prec_{\varphi_1'}}(h_T(a), a'),$$

where $f_\iota^A(p_1)$ and $f_\iota^{A'}(p_1)$ are defined as in diagram below, $h_T : T_{\varphi_1} \rightarrow T_{\varphi_1'}$ is an isomorphism, $A = \{L, R\}$, $f_\iota^A(p_1)(a) = f_\iota^{A'}(p_1)(a')$.



Let us consider $f_\iota^{A'}(p_1)(a') = f_\iota^A(p_1)(a)$. By definition of the dependency relation of a process, for all $(a, x) \in \prec_{\varphi_1}$ and all $(y, a) \in \prec_{\varphi_i}$, with $i = 1..n$, if $f_\iota^{A'}(p_1)(h_T(a)) = f_\iota^A(p_1)(a)$, then there exists $(h_T(a), x) \in \prec_{\varphi_1}$ and (y, x) is in $\prec_\varphi$ using φ_1 or φ_1'; otherwise, using φ_1, we have $(y, x) \in \prec_\varphi$ and using φ_1' we have (y, x) as well, because, by (1), there exists $(a', x) \in \prec_{\varphi_1'}$. This holds for all $(x, a) \in \prec_{\varphi_1}$ and all $(a, y) \in \prec_{\varphi_i}$ by symmetry. □

Now we can define the composition and identity for dT-GTS implementation morphisms.

Proposition 4.4 (composition and identity for dT-GTS implementation morphisms) *Given a dT-GTS $\mathcal{Z}$, let $\widehat{\mathcal{Z}}$ be as in Definition 4.11. Then, the properties below hold.*

1. *Given dT-GTS implementation morphisms $f \colon \mathcal{Z}_1 \rightarrow \mathcal{Z}_2$ and $g \colon \mathcal{Z}_2 \rightarrow \mathcal{Z}_3$, let their composition $\langle h_T, h_P \rangle = g \circ f \colon \mathcal{Z}_1 \rightarrow \mathcal{Z}_3$ be defined by dT-GTS morphism $\langle h_T, h_P \rangle \colon \widehat{g} \circ f \colon \mathcal{Z}_1 \rightarrow \widehat{\mathcal{Z}_3}$. Then the composition is well defined and it is associative.*

2. *For each dT-GTS $\mathcal{Z}$, let $id_\mathcal{Z} = \langle id_{\mathcal{Z}T}, id_{\mathcal{Z}P} \rangle \colon \mathcal{Z} \rightarrow \widehat{\mathcal{Z}}$ be defined as*
   - *the type graph component $id_{\mathcal{Z}T}$ is the identity;*
   - *each dep-production p is mapped by $id_{\mathcal{Z}P}$ to the dwut-process $[\phi_{id_p}]_d$.*

   Then $id_\mathcal{Z}$ is well-defined and it is the identity on $\mathcal{Z}$.

Proof:

1. Since we proved, by Proposition 4.3, that $\widehat{g}$ maps each dwut-process of $\mathcal{Z}_2$ into a dwut-process of $\mathcal{Z}_3$ and it is well-defined, then the composition $\widehat{g} \circ f$ is well defined. It remains to prove that the composition is associative. Let $f : \mathcal{Z}_1 \rightarrow \mathcal{Z}_2$, $g : \mathcal{Z}_2 \rightarrow \mathcal{Z}_3$ and $h : \mathcal{Z}_3 \rightarrow \mathcal{Z}_4$ be dT-GTS implementation morphisms. Then we must prove:

$$(h \circ g) \circ f = h \circ (g \circ f)$$

(a) $(h_T \circ g_T) \circ f_T = h_T \circ (g_T \circ f_T)$: by associativity of partial graph morphisms.

(b) $(\widehat{h_P \circ g_P}) \circ f_P = \widehat{h}_P \circ (\widehat{g}_P \circ f_P)$: since extension $\widehat{f}$ for any implementation morphism f is defined in the same way for T-GTSs and dT-GTSs and by Proposition 3.6, we have that $\forall p \in P_1 \,.\, ((\widehat{h_P \circ g_P}) \circ f_P)(p) \approx^w (\widehat{h}_P \circ (\widehat{g}_P \circ f_P))(p)$, i.e., both compositions result in the same wut-process. Then it remains to prove that the dependency relations of these processes are the same: it holds by associativity of union of relations.

2. (a) *$id_{\mathcal{Z}}$ is well-defined.* Since $id_{\mathcal{Z}T}$ is an identity, then it is total and preserves stable and unstable items. Moreover, $id_{\mathcal{Z}P}$ maps each *dep*-production of $\mathcal{Z}$ into a dwut-process of $\mathcal{Z}$. It remains to prove that, for all *dep*-production $p \in P \,.\, id_{\mathcal{Z}P}(p) = [\phi_{id_p}]_d$, there are three morphisms from $\Pi_{\mathcal{Z}}([\phi_{id_p}]_d)$ to $L_p \hookleftarrow K_p \rightarrowtail R_p$. Since $[\phi_{id_p}]_d$ is the dwut-process containing only p as production, its underlying span is isomorphic to $L_p \hookleftarrow K_p \rightarrowtail R_p$, then the required morphisms are given by any triple of isomorphisms mapping the span $\Pi_{\mathcal{Z}}([\phi_{id_p}]_d)$ to $L_p \hookleftarrow K_p \rightarrowtail R_p$ and making $\prec_{\phi_{id_p}} = \prec_p$.

   (b) *$id_{\mathcal{Z}}$ is the identity.* It holds by Proposition 3.6

□

Definition 4.12 (category $\mathbf{dTGTS}^{imp}$) *We denote by $\mathbf{dTGTS}^{imp}$ the category having dT-GTSs as objects and dT-GTS implementation morphisms as arrows.*

4.4 Adjunction between $\mathbf{dGTS}$ and $\mathbf{dTGTS}^{imp}$

As like for T-GTSs, we will show that the abstract d-GTS associated to a dT-GTS has the same behaviour of the original dT-GTS, from the point of view of an external observer. This is done by using an adjunction between the categories $\mathbf{dGTS}$ and $\mathbf{dTGTS}^{imp}$.

In order to prove that the adjunction exists, we define two functors: the abstraction functor and the inclusion functor. Using these functors, we can relate an abstract system to its concrete counterpart through an implementation morphism, showing the one-to-one relation between transactions of a dT-GTS and those of its abstract counterpart.

Theorem 4.1 (Universality of abstraction) *The functor $\mathcal{A}^D : \mathbf{dTGTS}^{imp} \rightarrow \mathbf{dGTS}$, that maps every dT-GTS $\mathcal{Z}$ into its abstract d-GTS (see Definition 4.10), has a left adjoint $\mathcal{I}^D : \mathbf{dGTS} \rightarrow \mathbf{dTGTS}^{imp}$, which is the inclusion of $\mathbf{dGTS}$ into $\mathbf{dTGTS}^{imp}$.*

Proof: This proof can be divided in three parts:

1. Definition of the functor $\mathcal{I}^{\mathcal{D}}\langle_\rangle : \mathbf{dGTS} \rightarrow \mathbf{dTGTS}^{imp}$:
   - on objects: let $\mathcal{G} = \langle T, P, \pi\rangle$ be a d-GTS, then $\mathcal{I}^{\mathcal{D}}\langle G\rangle = \langle\langle T, P, \pi\rangle, T\rangle$;
   - on morphisms: for any $f : \mathcal{G}_1 \rightarrow \mathcal{G}_2$, $\mathcal{I}^{\mathcal{D}}\langle f\rangle : \mathcal{I}^{\mathcal{D}}\langle \mathcal{G}_1\rangle \rightarrow \widehat{\mathcal{I}^{\mathcal{D}}\langle \mathcal{G}_2\rangle} = \langle f_T, f'_P\rangle$, where $\forall p \in P \,.\, f'_P(p) = [\phi_{id_q}]_d$, such that $f_P(p) = q$. Note that dependency of p is preserved by $\prec_{\phi_{id_q}}$ because $\prec_q = \prec_{\phi_{id_q}}$ and f_P preserves dependency of p (by definition).
2. Definition of the functor $\mathcal{A}^{\mathcal{D}}\langle_\rangle : \mathbf{dTGTS}^{imp} \rightarrow \mathbf{dGTS}$:
   - on objects: let $\mathcal{Z}$ be a dT-GTS, then $\mathcal{A}^{\mathcal{D}}\langle \mathcal{Z}\rangle = A_{\mathcal{Z}} = \langle T_s, \mathbf{DwtProc}(\mathcal{Z}), \Pi_{\mathcal{Z}}\rangle$;
   - on morphisms: for any $f : \mathcal{Z}_1 \rightarrow \mathcal{Z}_2$, $\mathcal{A}^{\mathcal{D}}\langle f\rangle : \mathcal{A}^{\mathcal{D}}\langle \mathcal{Z}_1\rangle \rightarrow \mathcal{A}^{\mathcal{D}}\langle \mathcal{Z}_2\rangle = \langle h_T, h_P\rangle$, where:
     - $h_T = \langle f_{T_V|V_{T_{1_S}}}, f_{T_E|E_{T_{1_S}}}\rangle$;

  – $\forall [\phi]_d \in \mathbf{DwtProc}(\mathcal{Z}_1) \,.\, h_P([\phi]_d) = \widehat{f}_P([\phi]_d)$. Note that, by definition of $\widehat{f}_P$, the dependency of $[\phi]_d$ is preserved by $\prec_{\widehat{f}_P([\phi]_d)}$.

3. To prove that $\mathcal{I}^{\mathcal{D}}\langle _ \rangle$ and $\mathcal{A}^{\mathcal{D}}\langle _ \rangle$ form an adjunction we have to prove that

$$\forall \mathcal{G} \in \mathbf{dGTS}, \forall \mathcal{Z} \in \mathbf{dTGTS}^{imp}, \forall f : \mathcal{I}^{\mathcal{D}}\langle \mathcal{G} \rangle \to \mathcal{Z} \,.\, \exists! h : \mathcal{G} \to \mathcal{A}^{\mathcal{D}}\langle \mathcal{Z} \rangle \,.\, \epsilon_{\mathcal{Z}} \circ \mathcal{I}^{\mathcal{D}}\langle h \rangle = f$$

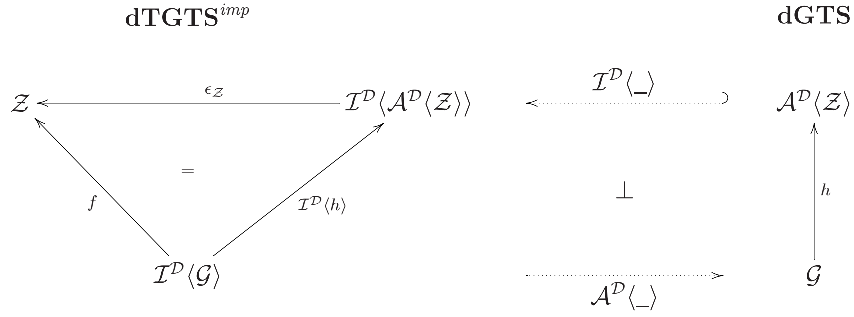


Figure 4.8: Universality of $\epsilon_{\mathcal{Z}}$ in $\mathbf{dTGTS}^{imp}$.

For each dT-GTS $\mathcal{Z}$, we define the component at $\mathcal{Z}$ of the counit $\epsilon_{\mathcal{Z}} \colon \mathcal{I}^{\mathcal{D}}\langle \mathcal{A}^{\mathcal{D}}\langle \mathcal{Z} \rangle \rangle \to \mathcal{Z}$. This is an implementation morphism, thus a dT-GTS morphism $\epsilon_{\mathcal{Z}} \colon \mathcal{I}^{\mathcal{D}}\langle \mathcal{A}^{\mathcal{D}}\langle \mathcal{Z} \rangle \rangle \to \widehat{\mathcal{Z}}$ is defined as follows:

- $\epsilon_{\mathcal{Z}T} = T_s \hookrightarrow T$;
- $\forall [\phi]_d \in \mathbf{DwtProc}(\mathcal{Z}) \,.\, \epsilon_{\mathcal{Z}P}([\phi]_d) = [\phi]_d$ (remember that productions in $\mathcal{A}^{\mathcal{D}}\langle \mathcal{Z} \rangle$ are exactly the dwt-processes of $\mathcal{Z}$).

It remains to show that given a d-GTS $\mathcal{G}$ and a dT-GTS $\mathcal{Z}$, for each dT-GTS implementation morphism $f \colon \mathcal{I}^{\mathcal{D}}\langle \mathcal{G} \rangle \to \mathcal{Z}$, there is a unique $h \colon \mathcal{G} \to \mathcal{A}^{\mathcal{D}}\langle \mathcal{Z} \rangle$ such that $\epsilon_{\mathcal{Z}} \circ \mathcal{I}^{\mathcal{D}}\langle h \rangle = f$.

(a) Definition of (d-GTS morphism) $h : \mathcal{G} \to \mathcal{A}^{\mathcal{D}}\langle \mathcal{Z} \rangle$.

  - $h_T = f_T$ (since f_T maps the type graph of $\mathcal{G}$ into stable items of $\mathcal{Z}$);
  - $\forall p \in P \,.\, h_P(p) = f_P(p)$ (since productions in $\mathcal{A}^{\mathcal{D}}\langle \mathcal{Z} \rangle$ are exactly the dwt-processes of $\mathcal{Z}$ and f maps each production of $\mathcal{G}$ to a dwt-process of $\mathcal{Z}$).

(b) $\epsilon_{\mathcal{Z}} \circ \mathcal{I}^{\mathcal{D}}\langle h \rangle = f$.

  - $\epsilon_{\mathcal{Z}T} \circ \mathcal{I}^{\mathcal{D}}\langle h_T \rangle = f_T$. This holds by item (3.b) of Theorem 3.1.
  - $\widehat{\epsilon_{\mathcal{Z}}}_P \circ \mathcal{I}^{\mathcal{D}}\langle h_P \rangle = f_P$.
    Since all productions of $\mathcal{I}^{\mathcal{D}}\langle \mathcal{A}^{\mathcal{D}}\langle \mathcal{Z} \rangle \rangle$ are totally stable (they are all transactions of $\mathcal{Z}$), then $\mathcal{I}^{\mathcal{D}}\langle \mathcal{A}^{\mathcal{D}}\langle \mathcal{Z} \rangle \rangle$ have one dwut-processes $[\phi_{id_{[\phi]_d}}]_d$ for each *dep*-production $[\phi]_d$ (see the observation before Proposition 3.5). Moreover, by definition of $\widehat{\epsilon_{\mathcal{Z}}}_P$, we have $\widehat{\epsilon_{\mathcal{Z}}}_P([\phi_{id_{[\phi]_d}}]_d) = [\phi]_d$, with $\prec_{\phi_{id_{[\phi]_d}}} = \prec_{\phi}$. By definition, $h_P = f_P$ and $\mathcal{I}^{\mathcal{D}}\langle h_P \rangle$ maps each *dep*-production $p \in P_{\mathcal{G}}$ to dwut-process $[\phi_{id_{f_P(p)}}]_d$, then $\forall p \in P_{\mathcal{G}} \,.\, \widehat{\epsilon_{\mathcal{Z}}}_P(\mathcal{I}^{\mathcal{D}}\langle h_P \rangle(p)) = \widehat{\epsilon_{\mathcal{Z}}}_P([\phi_{id_{f_P(p)}}]_d) = f_P(p)$, with $\prec_{\phi_{id_{f_P(p)}}} = \prec_{f_P(p)}$.

(c) h is unique. It holds by item (3.c) of Theorem 3.1.

□

4.5 Comparing T-GTS with dT-GTS

In this section, we compare T-GTSs and dT-GTSs in terms of information that each one can express. The inclusion of dependencies in a production does not modify its semantics, but in adds information about its implementation. This extra information permits to identify resources needed to produces partial results. For example, the dependency relation associated to the dep-production in Figure 4.6 determines that message `Coupon` maybe sent after a message `Prepay` has been received, even if the others (`Supplay`, `Finish` and $Free$) are not received. Using this information, we can restrict possible implementations for a production, because only transactions which respect its dependency relation can be valid implementations.

We can say that a T-GTS (or dT-GTS) $\mathcal{Z}$ implements another one $\mathcal{Z}'$, if there exists an implementation morphism $f : \mathcal{Z} \to \mathcal{Z}'$, where f_T is total. Moreover, when a dep-production can be related to a transaction (by means of an implementation morphism), we say that the transaction implements the production. Considering this relationship between dep-productions and transactions, we will show that all dep-production can be implemented by some transaction. This is due to the restrictions imposed on the dependency relation, that guarantee the existence of at least one transaction implementing the dep-production.

Proposition 4.5 *There exists at least one transaction that implements a dep-production.*

Proof: The proposition holds because, for any dep-production p, we can obtain a transaction ϕ, which implements p. A transaction can be constructed as follows: consider a, b and c ranging over the deleted, created and preserved items of p, respectively; R'_p be the graph created by p (as defined in Definition 4.1); and $G = \mathcal{C}(R'_p)$ ranging over the connected components of R'_p. The set of productions of ϕ contains:

deleting productions one production for each a, deleting a; preserving the source and target of a, if it is an edge; and creating an unstable item $\langle a, b\rangle$ for each b that depends on a, where if a is a vertex then $\langle a, b\rangle$ will be a vertex, else $\langle a, b\rangle$ will be an edge with same source and target of a;

creating productions one production for

- each G, deleting all unstable items $\langle a, b\rangle$, where b is in G; preserving each c such that some b in G depend on it; and creating an unstable item $\langle G\rangle$ (used for synchronisation) and creating all b in G;
- each $x \in R_p - K_p \wedge x \notin R'_p$ (edges created over preserved vertices), deleting all unstable items $\langle a, x\rangle$; preserving all c such that x depends on it; and creating x and unstable vertex $\langle x, \text{“}s\text{”}\rangle$

synchronisation production one production deleting all vertex $\langle G\rangle$ and $\langle x, \text{“}s\text{”}\rangle$.

The dependency relations associated to productions of ϕ are total, i.e., the created items depend of all items in the left-hand side. Besides, the type graph of ϕ is the graph obtained joining all left- and right-hand sides and interfaces of all deleting, creating and synchronisation productions.

One can note that ϕ is a transaction because: (1) all productions create unstable items, but the synchronisation one. As this production can be applied only after all stable items of p are created, it is the last production to applied, resulting in a stable state. Therefore, the intermediate states have always at least one unstable item; (2) all created stable items are not used in the transaction because the deleting productions (which are the only ones that delete stable items) delete only

items in the initial state; and (3) all items consumed and preserved are used, because there is one production for each deleted item and the preserved ones are used in the creating productions. Besides, we can see that ϕ implements p because the minimal graph (all deleted and preserved items) coincides with L_p, the maximal graph (all created and preserved items) coincides with R_p and, as the productions are constructed considering the dependency relation, ϕ preserves the dependency of p. □

The ut-process ϕ depicted in Figure 4.9 is a transaction that implements the production STOP in Figure 4.2, obtained as defined in Proposition 4.5. The productions named as d$DeletedElement$ and c$CreatedElement$ (c$CreatedGraph$) are productions obtained as described in deleting and creating productions, respectively. The synchronisation production has the obvious name. The type graph, depicted at the top-right, is constructed as the gluing of all graphs of productions and the minimal (maximal) graph is composed by items of type graph that are not created (consumed) by any production. Table 4.4 describes transitive closure of all dependency relations of productions of ϕ, and the corresponding restriction to minimal and maximal graphs is:

$$\begin{array}{r} \mathit{op} \\ \mathit{Pump} \\ \mathit{Operator} \\ \mathit{Customer} \\ \mathit{Finish} \\ \mathit{Busy} \end{array} \prec_\phi \mathit{Free} \qquad \begin{array}{r} \mathit{op} \\ \mathit{Pump} \\ \mathit{Operator} \\ \mathit{Custumer} \\ \mathit{Finish} \end{array} \prec_\phi \mathit{Charge}$$

Observing the minimal and maximal graphs, one can see that the minimal and maximal graphs of ϕ are isomorphic to left- and right-hand sides of STOP production, respectively. Moreover, the dependency relation associated to ϕ (described above) is equivalent to that associated to STOP production.

Table 4.4: Transitive closure of dependencies of productions of ϕ depicted in Figure 4.9.

Pump Customer Finish	$\preceq_{P_\phi}$	Finish/Free Finish/Charge	$Busy$ Pump	$\preceq_{P_\phi}$	$Busy/Free$
op Pump Customer Operator $Busy$ Finish Finish/Free $Busy/Free$	$\preceq_{P_\phi}$	$Free$ $Free$/"s"	op Pump Customer Operator Finish Finish/Charge	$\preceq_{P_\phi}$	Charge Charge/"s"

As mentioned, the association of the dependency relation to productions of a T-GTS restricts the possible valid implementations. This restriction means that, when we are in the context of T-GTSs, we can have different transactions implementing a production p. Even if they have different dependencies between their elements, they are still implementations of p, since dependencies are not being taken into account. On the other hand, if we consider the same production p having two different dependency relations associated to it, we would have two dep-productions p_1 and p_2 differing only on the dependencies. Some implementations of p are, now, implementations of p_1, while other ones may be implementations of p_2, but there are no common implementation for both, because the

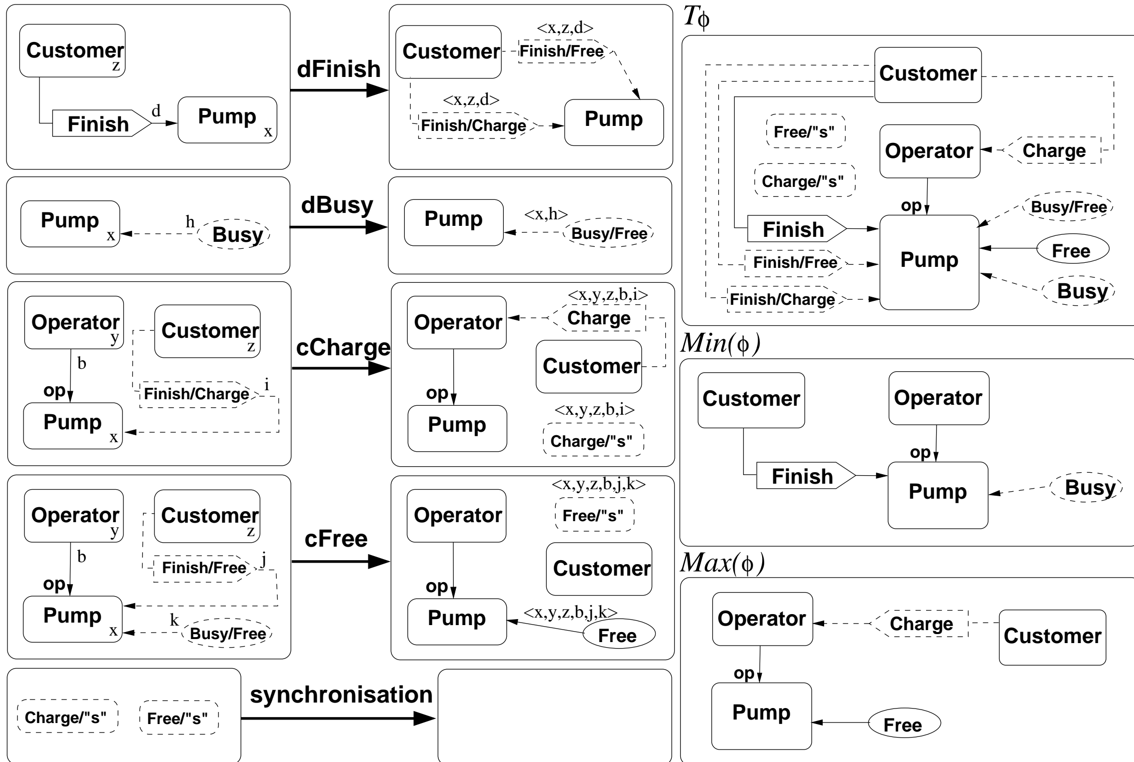


Figure 4.9: Transaction constructed from dep-production STOP in Figure 4.2.

implementation relationship must preserve the dependencies. Therefore, if we choose p_1 we will be discarding all implementations of p_2 (which are valid for p) and vice-versa. We demonstrate this fact in the following theorem.

In the following, the set $I(\mathcal{Z})$ contains all T-GTSs (or dT-GTSs) that implement $\mathcal{Z}$.

Theorem 4.2 (dependency relation restricts the implementations of a T-GTS) *Given a* dT-GTS $\mathcal{Z} = \langle\langle T, P, \pi\rangle, T_s\rangle$ *and a* T-GTS $\mathcal{U}(\mathcal{Z}) = \langle\langle T, P, \pi'\rangle, T_s\rangle$*, where* $\forall p \in P \wedge \pi(p) = \langle L_p \hookleftarrow K_p \rightarrow R_p, \prec_p\rangle \bullet \pi'(p) = L_p \hookleftarrow K_p \rightarrow R_p$.

$$\text{If } \exists p \in P \wedge \pi'(p) = \langle L_p \hookleftarrow K_p \rightarrow R_p, \prec_p\rangle \wedge \exists \prec'_p \neq \prec_p$$
$$\text{then } \exists \mathcal{Z}' \notin I(\mathcal{Z}) \bullet \mathcal{Z}'' \in I(\mathcal{U}(\mathcal{Z})) \wedge \mathcal{Z}' = \mathcal{U}(\mathcal{Z}'')$$

Proof: The theorem holds by definition of implementation relationship, that differs from T-GTS to dT-GTS by including a restriction for dT-GTS: it must preserve the dependency relations. Considering a T-GTS $\mathcal{Z}$ containing only the production p, if there are two different dependency relations $\prec_1$ and $\prec_2$ for p, then we can consider two different dT-GTSs $\mathcal{Z}_1$ and $\mathcal{Z}_2$ containing the *dep*-productions $\langle p, \prec_1\rangle$ and $\langle p, \prec_2\rangle$, respectively. For each dT-GTS $\mathcal{Z}_i$ there exists an implementation $\mathcal{Z}'_i$ associating the transaction $[\phi_i]_d$ with the production $\langle p, \prec_i\rangle$. By definition of implementation relationship, the dT-GTS $\mathcal{Z}'_1$ cannot be an implementation for $\mathcal{Z}_2$. When we consider T-GTSs, the restriction on dependencies does not exist and, therefore, $\mathcal{Z}$ can be implemented by both $\mathcal{Z}'_1$ and $\mathcal{Z}'_2$ (forgetting the dependency relations). □

Example 4.8 (restriction of implementations) *Let us consider the* dT-GTS $\mathcal{Z}_2$ *depicted in Figure 4.10. Note that* $\mathcal{Z}_1$*, in Figure 4.7, and* $\mathcal{Z}_2$ *differ only in the dependency for* `Coupon` *in the production* p_1*: here the* `Coupon` *message is sent only after the system have received the* `Prepay`*,* `Finish` *and* `Supply` *messages.*

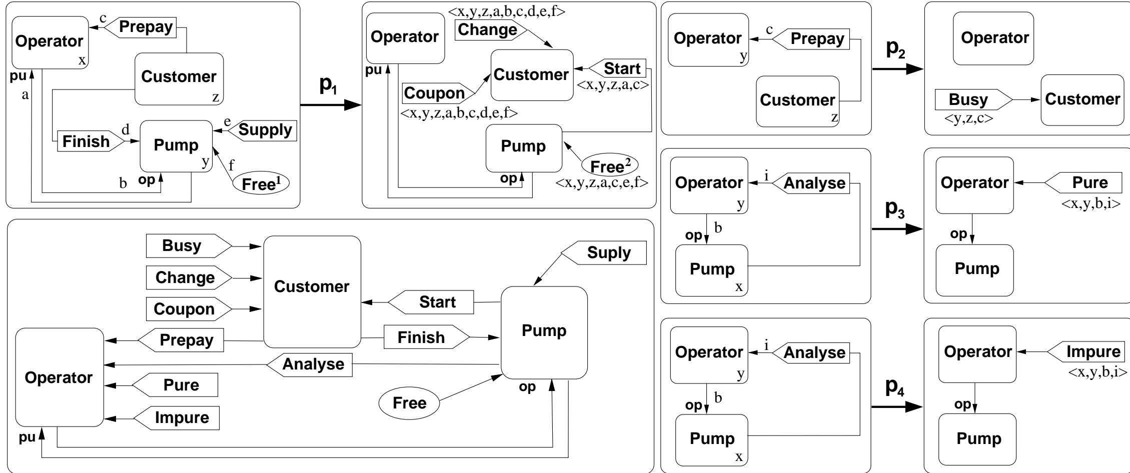


Figure 4.10: The dT-GTS $\mathcal{Z}_2$.

*As shown in Example 4.6, $\mathcal{D}epPumpOper$ (in Example 4.2) is a possible implementation of $\mathcal{Z}_1$, but we can see that $\mathcal{D}epPumpOper$ cannot be an implementation of $\mathcal{Z}_2$, because it does not have a transaction with the same dependency relation of p_1. A possible implementation for $\mathcal{Z}_2$, would be a d*T-GTS *$\mathcal{Z}_2'$ with the same type graph and productions of $\mathcal{D}epPumpOper$, substituting the* ACCEPT *and* FINISH *productions by those shown in Figure 4.11. If we forget the dependencies of $\mathcal{Z}_1$ and $\mathcal{Z}_2$ we obtain the same* T-GTS *that*

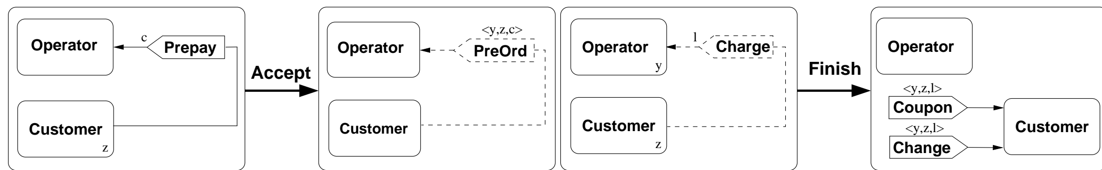


Figure 4.11: Productions ACCEPT and FINISH for the dT-GTS $\mathcal{Z}_2'$.

can be implemented by both $\mathcal{D}epPumpOper$ and $\mathcal{Z}_2'$ (forgetting their dependencies, as well).

┘

The proposed extension of T-GTSs including dependency relations on productions allows us to describe (abstractly) more information than its version without dependencies, for example, interactions between systems (or between a system and its environment). Comparing the abstract systems depicted in Figures 3.12 and 4.7, we can see that describing the same wut-process ϕ (described in Figures 3.9 and 4.5) using the production $[\phi_1]_w$ (version without dependencies) we are requiring that all messages `Prepay`, `Finish` and `Start` are present, as well as the pump is free, in order to produce its result (to send `Change`, `Coupon` and `Start` messages and to free the pump). But if we consider the implementation of this production, we can see, for example, that `Coupon` message is sent as soon as the `Prepay` message is received. This situation is perfectly described by the dependency relation of $[\phi_1]_d$ (version with dependencies).

4.6 Construction of the abstract system associated to a dT-GTS

It can be useful to have a method to construct an abstract system from a dT-GTS. This construction be used to obtain automatically the abstract view of a system and to verify the correction of an implementation, since each production of an abstract system must be associated to a transaction of the concrete one. In Definition 4.14, we propose an algorithm to construct the abstraction of a dT-GTS. However, this construction can not be performed for any dT-GTS, only for systems with finite number of transactions. In order to guarantee the finiteness of the system, we must restrict the kind of dT-GTSs used to specify our modules. Therefore, we will consider only systems whose productions do not have cycles on unstable items, i.e., if we consider the causal dependency of the unstable part of the type graph, we do not allow one element to depend on itself. In order to simplify the construction we will restrict to dT-GTSs whose productions create or delete only one element of each type and do not consume vertices.

In order to construct the abstract system associated to a dT-GTS we must obtain the set of transactions of this system. The construction of set of transactions of a dT-GTS is formally described as in Definition 4.13. Summarising this construction, we first initialise the set of transactions (line 1): we pick each stable production out of set of productions and include one transaction containing this production as the unique production; next we construct the set P' of all possible subsets of productions of the system; finally we eliminate from P' those subsets which do not contain at least one production with the left-hand side stable and other with the right-hand side stable. Then, we select (among the remaining subsets in P') the production subsets that constitute transactions and construct them (lines $2-43$).

A set of productions is balanced if all unstable items created by these productions are consumed by them. In line 3, for each subset of productions A (in P') the function $preTransactions$ returns a list of pairs of initial graph and balanced productions (based on productions in A). First, we test if the productions are balanced and, if this test fail, we test if it is possible to obtain other sets of balanced productions, including new instances of these productions. The inclusion of new instances is minimal and may result in several new subsets (if there are more than one way to balance the productions). Then, for each subset of balanced productions (based on A), we construct all possible initial graphs: gluing the stable part of left-hand side of all productions, in all possible ways.

For each pair of initial graph and balanced productions, we construct the graph process (lines $4-30$), test if the this process is transactional (lines $31-35$) and exclude processes that have a dep-equivalent process in the set of transactions (lines $36-40$). The description of functions used in the algorithm below are presented in Appendix C.

Definition 4.13 (construction of transactions of a dT-GTS) *Given a* dT-GTS $\mathcal{Z} = \langle\langle T_\mathcal{Z}, P_\mathcal{Z}, \pi_\mathcal{Z}\rangle, T_{\mathcal{Z}s}\rangle$*, the set of its transactions* $\mathbf{DwtProc}(\mathcal{Z})$ *can be obtained as follows:*

1: $\langle P', \mathbf{DwtProc}(\mathcal{Z})\rangle = init(P_\mathcal{Z})$ ▷ *initialise the set* $\mathbf{DwtProc}(\mathcal{Z})$ *of transactions of* $\mathcal{Z}$ *with one-step transactions and* P' *with subsets of* $P_\mathcal{Z}$ *containing no stable production and at least one production with left-hand side stable and other with right-hand side stable*

2: ***for all*** $A \in P'$ ***do*** ▷ *repeat for each set of productions in* P'

3: $\quad l = preTransactions(A)$ ▷ *generate initial graph and (balanced – possibly with different instances of a same production) productions of all possible processes with productions in* A

4: **while** $l \neq \lambda$ **do** ▷ *repeat for each pair in* l
5: $\langle G, P\rangle = head(l)$ ▷ *get the first initial graph* G *and set of productions* P
6: $l = tail(l)$ ▷ *eliminate the first element of* l
7: $U = \varnothing$ ▷ *initialise the set of graph processes containing productions in* P
8: $U_1 = \varnothing$ ▷ *initialise the first state of* U
9: $U_2 = \{\phi = \langle \phi_T, \phi_P\rangle : \langle\langle G, \varnothing, \varnothing\rangle, \mathcal{S}(G)\rangle \to \mathcal{Z}\}$, *where*
$\phi_T = \langle id_G, t_G\rangle : G \to T_{\mathcal{Z}}$ *is a partial morphism and*
$\phi_P = \varnothing : \varnothing \to P_{\mathcal{Z}}$ ▷ *initialise* U_2 *(second state of* U*) with a graph process composed of type graph* G *and none production*
10: $i = 2$
11: **while** $U_i \neq U_{i-1}$ **do** ▷ *repeat until there is no change in* U_i
12: **for all** $\phi_i : \langle\langle T_i, P_i, \pi_i\rangle, T_{is}\rangle \to \mathcal{Z} \in U_i$ **do** ▷ *repeat for each process in* U_i
13: **if** $P_i = P$ **then** ▷ *if all productions in* P *were used in the process, it is complete*
14: $\phi_i \in U_{i+1}$ ▷ *and it is included in* U_{i+1}
15: **else**
16: $x = 0$ ▷ *initialise* x *indicating that there is no change in* ϕ_i
17: **for all** $\langle p, k\rangle \in P$ *and* $\langle p, k\rangle \notin P_i$ **do** ▷ *repeat for each production in* P *that was not used in* ϕ_i
18: **for all** match $m : L_p^{T_{\mathcal{Z}}} \to \langle T_i, r_{\phi_{iT}}\rangle$ **do** ▷ *repeat for each match of selected production in the type graph of* ϕ_i
19: **if** $m(L_p)$ *is a concurrent subgraph of* T_i **then** ▷ *if the range of selected match contains only concurrent elements*
20: $make_{\mathcal{Z}}(\phi_i, \langle p, k\rangle, m) \in U_{i+1}$ ▷ *then include, in* U_{i+1}*, the process* ϕ_i *updated with used production and created elements in the type graph*
21: $x = 1$ ▷ *and assign* 1 *to* x *to indicate that* ϕ_i *was changed*
22: **end if**
23: **end for**
24: **end for**
25: **if** $x = 0$ **then** $\phi_i \in U_{i+1}$ ▷ *if* x *was not changed in line* 21*, the process* ϕ_i *cannot be changed and it is included in* U_{i+1}
26: **end if**
27: **end if**
28: **end for**
29: $i = i + 1$ ▷ *increment* i *in order to get next state of* U
30: **end while**
31: **for all** $\phi \in U_{i-1}$ **do** ▷ *get each process in the last state of* U
32: **if** $P_\phi = P$ *and* $transaction(\phi)$ **then** ▷ *if this process contains all productions in* P *and is a transaction*
33: $\phi \in U$ ▷ *then include it in the set of transactions* U
34: **end if**
35: **end for**
36: **for all** $\phi \in U$ **do** ▷ *for each transaction in* U
37: **if** *there exists* $\phi' \in U$ **.** $\phi' \neq \phi \wedge depEq(\phi', \phi)$ **then** ▷ *if there is other transaction in* U *that is dep-equivalent on selected transaction*
38: $U = U - \{\phi\}$ ▷ *exclude it of* U

39: **end if**
40: **end for**
41: $\mathbf{DwtProc}(\mathcal{Z}) = \mathbf{DwtProc}(\mathcal{Z}) \cup U$ ▷ *include the transactions obtained using G as initial graph and P as set of productions in the set of transactions of* $\mathcal{Z}$
42: **end while**
43: **end for**

Analysing the algorithm described above, we can see that the executions of (almost) all loops are controlled by finite structures, for example, the **forall** loop at lines $2 - 43$ is repeated for each element in P', that is a set containing a finite number of productions. However, the **while** loop at lines $11 - 30$ does not work in the same way: it stops if the set of graph processes does not change in a previous iteration. This loop always stops because this set is not changed if all productions in P are applied (lines $13 - 15$) or if there is no production in P that can be applied (lines $25 - 26$). Then, considering that all functions used in Definition 4.13 stop (see Appendix C), we can conclude that the proposed algorithm always stops.

Moreover, the algorithm generates exactly the set of transactions of the considered dT-GTS. It is easy to see that the generated processes are all dep-equivalent transactional processes (by tests in lines $32 - 34$ and $37 - 39$) obtained using productions of $\mathcal{Z}$. Then, it remains to see if all transactions are generated: the transactions with one production are constructed in the algorithm initialisation and the other ones are constructed based on all possible subsets of productions with possible minimal repetitions, i.e., the number of different instances of a same production is limited by the definition of transaction, in particular by the fact that a transaction can not be divided into small transactions. Moreover, for each subset of productions of $\mathcal{Z}$, the possible initial graphs are obtained considering all possible ways to compose the left-hand side of productions (remember that all items in the minimal graph must be used). Since the processes are constructed considering all possible ways to apply the productions in all initial graphs, there can not be transactions that are constructed using productions of $\mathcal{Z}$ applying in different ways (all ways have been considered).

Now, we can construct the abstract system of a dT-GTS $\mathcal{Z}$, using the stable type graph of $\mathcal{Z}$ as type graph and the set of transactions of $\mathcal{Z}$ (obtained as in Definition 4.13).

Definition 4.14 (abstraction construction) *Given a d*T-GTS $\mathcal{Z} = \langle\langle T, P, \pi\rangle, T_s\rangle$*, we can construct the abstract system associated to* $\mathcal{Z}$ *as the system* $\langle T_s, \mathbf{DwtProc}(\mathcal{Z}), \Pi\rangle$*, where* $\mathbf{DwtProc}(\mathcal{Z})$ *is obtained as in Definition 4.13 and for each* $\phi \in \mathbf{DwtProc}(\mathcal{Z})$*,* $\Pi(\phi)$ *is defined as in Definition 4.9.*

4.7 Refinement of transactional graph transformation systems

Since the aim of transformational systems is to produce a final result, given an input, they can be appropriately specified as a relation between initial and final states. However, the functionalities of reactive systems are given by an ongoing interaction with their environment, rather than by an output upon termination. In this context, notions of reactivity and concurrency are closely related. For example, a good way to explain the difference between transformational and reactive systems is that, in the transformational case, a system and its environment act sequentially, while in the reactive case they act concurrently (MANNA; PNUELI, 1992).

Graph transformation systems are a suitable formalism to specify complex systems, since graphs are used to describe naturally the structure of a system focusing attention on its components and their interconnections. Moreover, this formalism give us a simple manner to describe concurrency, where all productions of the system can be applied in parallel if they are independent. By means of the extension to the transactional version, introduced in Chapter 3, we can use graph transformation systems to describe atomic activities in a more detailed way and use a more abstract view when it is interesting.

Several methods for design and analysis of reactive systems propose synchronous languages as specification formalism (BERRY, 2000; HALBWACHS et al., 1991; LEGUERNIC et al., 1991), where the time of reaction to an event is null. This characteristic is important to simplify the model, but frequently, it does not correspond to the reality, as is the case of distributed systems, where the communication between components may take some time. Thus, new approaches were introduced to combine synchronous components and asynchronous communication (FILALI; MAURAN; PADIOU, 1993; BERRY; RAMESH; SHYAMASUNDAR, 1993; RIESCO; TUYA, 2004). GTSs have an asynchronous semantics and with the introduction of transaction notion, it becomes possible to synchronise internal activities of a component, modelling them as transactions. Thus, at a more abstract level, we can consider a transaction as an immediate reaction, where the intermediate steps are considered to take a null time. Some of cited approaches use similar notions to model the synchronous behaviour of system components.

Moreover, the extension of transactional GTS to express causal relation between input and output signals allows us to explicitly describe interaction patterns. The dependency associated with each production will be used to describe the dependency between exchanged messages/signals, as proposed in (FOSS; MACHADO; RIBEIRO, 2007): we describe the interaction of the system with its environment in order to realise their operations: which signals are sent to environment in reaction to received ones. Thus, the transactional GTS with dependency relation becomes a interesting formalism to specify reactive systems, that are characterised by an ongoing interaction and atomic reactions.

In a top-down approach to develop a reactive system, one can start abstracting the behaviour of the system defining only the interaction between system and its environment and then, to refine the specification adding new details of each reaction.

There are different notions of refinement, among them, behavioural refinement is more usual, where the properties of the abstract specification are implied by properties of the refined specification. Since we are describing the abstract behaviour of a system by its interaction with the environment, we can consider two kind of system views: black and glass-box views (BROY; STØLEN, 2001). While in the black-box view only the input and output signals are observable, and what happens inside of component between the consumption of an input signal and the generation of the corresponding output signal is hidden, in the glass-box view some constraints on the internal structure or behaviour can be defined. Using the defined implementation morphism we can define a glass-box refinement, where besides the external behaviour, also particular aspects of the internal structure are preserved. The internal structure to be preserved in our approach is the relation dependency between the input and output signals. Therefore, we can say that a dT-GTS $\mathcal{Z}_2$ is a refinement of a T-GTS $\mathcal{Z}_1$ if there is an implementation morphism from $\mathcal{Z}_1$ to $\mathcal{Z}_2$, where the type mapping is total and surjective on stable items and the production mapping is total and must associate a transactional process of $\mathcal{Z}_1$ to each transactional process of $\mathcal{Z}_2$. These requirements guarantee that all external behaviours are preserved and the definition of implementation morphism guarantees that dependency relation is

preserved. The verification of correction of refinement can be given by comparing the abstractions of both systems: the abstractions of original and refined specifications must be the same.

Definition 4.15 (refinement) *Let* $\mathcal{Z}_1 = \langle\langle T_1, P_1, \pi_1\rangle, T_{1s}\rangle$ *and* $\mathcal{Z}_2 = \langle\langle T_2, P_2, \pi_2\rangle, T_{2s}\rangle$ *be two d*T-GTS*s.* $\mathcal{Z}_2$ *is a* refinement *of* $\mathcal{Z}_1$ *if there exists an implementation morphism* $f\colon \mathcal{Z}_1 \to \mathcal{Z}_2$ *such that,*

- $f_T : T_1 \to T_2$ *is total and* $\mathcal{S}(f_T(T_1)) = T_{2s}$*;*
- $f_P : P_1 \to \mathbf{DwutProc}(\mathcal{Z})$ *is total;*
- $\forall [\phi_2]_w \in \mathbf{DwtProc}(\mathcal{Z}_2) \centerdot \exists [\phi_1]_w \in \mathbf{DwtProc}(\mathcal{Z}_1) \centerdot \widehat{f}_P([\phi_1]_w) = [\phi_2]_w.$

Based on this notion of refinement, a first refinement step of the abstract system in Figure 4.7 can result in the dT-GTS presented in Example 4.2. The implementation morphism defined between them is described in Example 4.7. It is easy to see that both have the same external behaviour, since the abstract specification is indeed the abstract system associated to the concrete one.

A further refinement step can result in the dT-GTS shown in Figure 4.12. The implementation morphism from $\mathcal{D}epPumpOper$ to $\mathcal{Z}$ is defined as follows: on the type graphs it is defined by the obvious inclusion of $T_{\mathcal{D}epPumpOper}$ in $T_{\mathcal{Z}}$ and on productions it is defined by mapping each production, except the STOP production, into a process containing itself as unique production. The STOP production is mapped into the unstable transactional process $[\psi]_d$ depicted in Figure 4.13. It is easy to see that the abstract d-GTS associated to $\mathcal{Z}$ is the same one associated to $\mathcal{D}epPumpOper$ depicted in Figure 4.7, this means that both systems have the same abstract behaviour and therefore $\mathcal{Z}$ is a correct refinement of $\mathcal{D}epPumpOper$.

Here, we propose to use dT-GTSs to specify, in an abstract way, reactive systems, using the dependency relation to specify reactions of the system to events/signals from the environment.

Figure 4.12: A refinement dT-GTS $\mathcal{Z}$ for $\mathcal{D}epPumpOper$ (Figure 4.3).

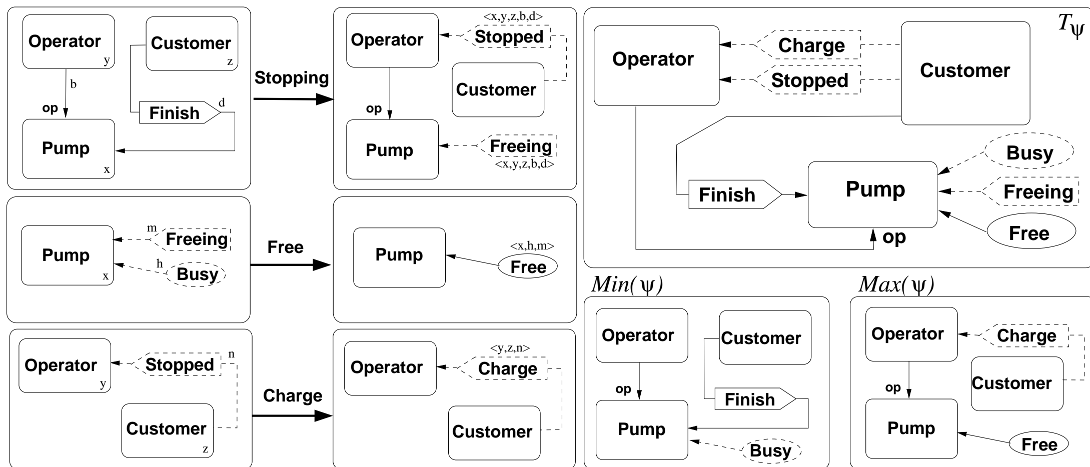


Figure 4.13: An unstable transactional process associated to STOP production.

5 CONCLUSIONS

The contributions of this work are related to two different areas of computer science: theoretical area, by providing a theoretical foundation of transaction concept for graph transformation; and software engineering area, by proposing a formalism to specify reactive systems, where interactions between system and its environment can be described by dependency relations associated to productions. Moreover, using implementation morphisms a method of incremental development can be defined, starting from an abstract view of the system and adding more details in each refinement step.

5.1 Theoretical contribution

In this work, we give the theoretical foundations of the notion of transactional activities in graph transformation. A transaction is defined as a class of derivations, where starting and ending states are stable and all intermediate states are unstable. Thus unstable items represent temporary resources, only visible within a transaction, and the distinction between stable and unstable items is enforced by a typing mechanism. This definition of transactions is inspired by the work on zero-safe nets (BRUNI; MONTANARI, 2000). It is motivated by the data-driven nature of graph transformation formalism, where any form of control on the application of productions has to be encoded in the graphs. The main theoretical result of the present work is the characterisation of the abstract system of a T-GTS, including all transactions as productions, in terms of a universal construction, presented as a right adjoint functor. In order to obtain this result, we first characterise transactions as graph processes and define the notion of implementation morphism, allowing to map productions to transactional processes. The notion of implementation defined for T-GTSs are more general than that defined in (BRUNI; MONTANARI, 2000), because our morphisms can relate unstable productions with unstable transactions and, consequently, we can also refine the implementations of stable productions.

Since stable items cannot be used within the transaction in which they are created, if we need to create stable vertices connected (by edges), this creation must be defined in a unique production. Otherwise. it would be necessary to use the created vertices in order to create the connecting edges. This restriction is not imposed in (BRUNI; MONTANARI, 2000), because Petri Nets can be seen as a GTSs where the states are represented by discrete graphs, i.e., graphs having only vertices.

Besides the above mentioned results, the extension of transactional graph transformations, introduced in Chapter 4 (dT-GTS), enriches the information given by T-GTS graph productions, making explicit the dependency between created and consumed/preserved items. This relation can be used to restrict the refinement of the transactions, since the notion of implementation morphism for dT-GTSs must respect the dependencies. This

extension does not change the semantics of the T-GTSs, only give an abstract information about desired dependencies in an implementation.

There are other notions of transactional activities in the area of graph transformations. Traditional notions of transaction have been considered, most importantly in the design of PROgramed Graph REwriting Systems (SCHÜRR, 1991; SCHÜRR; WINTER; ZÜNDORF, 1999; SCHÜRR; WINTER, 2000). PROGRES provides a development environment where basic operations, defined by graph transformation rules, can be combined using a rich set of control structures, including traditional programming language constructs, various kinds of non-deterministic choices, as well as transactions. It is a mixed textual and diagrammatic language. A basic specification in PROGRES is composed by a graph schema and a set of graph procedures. The graph schema specifies the static properties of a class of graphs, defining all used types of nodes, edges and their attributes. The graph procedures are defined by productions or transactions. The productions describe the modifications on a graph and have a graphical representation, while the transactions are described textually and provide control structures. The PROGRES approach is therefore similar to the way transactions are introduced in programming languages and other control-centered formalism.

The graph transformation units, used in the GRACE (KREOWSKI; KUSKE, 1996; ANDRIES et al., 1999) approach, give a "kind of transaction notion" based on control-flow. A transformation unit comprises a set of local rules, a set of used transformation units and a control condition which regulates how used units and rules must be applied, allowing infinite applications of them. We cannot consider a transformation unit as a real transaction because a basic characteristic of a transaction is to be a finite computation.

In (GROSSE-RHODE; PARISI-PRESICCE; SIMEONI, 1999), a notion of synchronous activities for typed graph transformation systems is defined. This notion is given by syntactical compositions: sequential and amalgamation compositions. Therefore we cannot extract any causal relation between the state elements.

5.2 Software Engineering contribution

The visual and data-driven approach of GTSs, makes then a natural formalism to specify reactive system, where the behaviour of components is defined by the flow of signals (data) that are generated by the environment and not by the control flow of the components. The transaction and dependency relation notions introduced in this work turn this formalism even more adequate to specify reactive systems. These extensions improve GTS formalism with a mechanism to specify atomic reactions and, thus, allow us to describe, at an abstract level, synchronous systems. We also give an insight on vertical structuring proposing a notion of refinement to relate abstract and concrete specifications, where the dependencies between input and output events/signals are given explicitly as an additional information about of abstract behaviour of the system. This information can be useful to specify the environment behaviour and for verification purposes, as will be discussed in section 5.3.

We can find, in the literature, several formalisms and frameworks for specification, verification and code generation of reactive systems, such that Statecharts (HAREL, 1987; SEKERINSKI, 1998), Esterel (BERRY; COURONNE; GONTHIER, 1988; BERRY, 2000; GIL; FERRO; BERNHARD, 1996; BHATTACHARJEE et al., 1999), CRP (Communicating Reactive Processes) (BERRY; RAMESH; SHYAMASUNDAR, 1993), UML diagrams (KERSTEN et al., 2002; ALAVIZAEDH; NEKOO; SIR-

JANI, 2007; ALAVIZADEH; SIRJANI, 2006), Graph Transformation Systems (HECKEL, 1998), Abstract State Machines (BÖRGER; GLÄSSER, 1995; MAIA; IORIO; BIGONHA, 1998; LAMCH; WYRZYKOWSKI, 2006), Synchronous Estelle (RIESCO; TUYA, 2004).

Statecharts are a visual approach to design reactive systems. They extends finite state diagrams (graphical representation of finite state machines) with tree concepts: hierarchy concurrency and communication. Statecharts are used in different frameworks as a language for graphical specification and are mapped into other formal languages for automatic verification and code generation (BHATTACHARJEE et al., 1999; KERSTEN et al., 2002; SEKERINSKI, 1998). In (SEKERINSKI, 1998), the authors propose a translation of Statecharts to Abstract Machine Notation (AMN) of the B method for analysis and refinement purposes. In this approach, a state diagram (simplest form of statecharts) is composed by a finite number of states and transitions, which are translated into AMN as enumerated set type and operations, respectively. The current state is stored in a variable and the operations reflect the state change. Refinement notions are given by AMN refinement rules, i.e., they are only for the translated code. The essence of a refinement relationship is that it preserves already proved system properties. It is based on observational substitutivity: any behaviour of the refined specification is one of the possible observable behaviours of the initial specification. More specifically, AMN refinement allows designers to reduce non-determinism of operations (strengthen the post-condition), to have more input values (weaken their preconditions) and to change the variable space. At the most abstract level it is mandatory to describe the static properties of a model by means of an invariant predicate. The refinement steps give rise to a number of proof obligations, which guarantee their correctness with respect to the invariant. Such proof obligations are discharged by the proof tool using automatic and interactive proof procedures supported by a proof engine. This approach uses a visual language to specify reactive systems, where the interaction pattern is described by labelled transitions of a state diagram. In a large specification, with several and complex interactions, the state diagram can become very large and difficult to understand. In our approach, each event involving simpler interactions can be described by an individual production, which is easier to be understood. Moreover, it is not necessary to know a different language (like Abstract Machine Notation) to refine the specified system. In (SCHOLZ, 1998), a refinement calculus for statecharts was proposed where the charts are obtained from non-deterministic sequential automata, hiding, and parallel composition. A reaction of a system is defined in terms of instants, where all input signals are received and all output signals are sent and the I/O behaviour of a system is described in terms of communication histories. The communication histories are given by streams carrying a set of signals, relation the input signals with output ones. They define a notion of refinement that requires that all input and output signals must be preserved (it is possible to extend them) and all behaviour relating these signals must be preserved, as well. Syntactic rules whose application guarantees a correct refinement step are defined. This approach avoids translations to other languages, but does not permit to describe a weaker relation between input/output signals of a more abstract level.

Esterel is a control-driven textual design language that can be used to generate complex state machines automatically. In this context, an assumption called perfect synchrony hypothesis (or atomic reaction) is made in order to simplify the behavioural specifications of reactive systems. This hypothesis states that the system reacts instantaneously to an input event, and the execution of reactions does not overlap with each other. Other assumption made in Esterel is that the systems are deterministic, thus their statements and con-

structs are guaranteed to be deterministic, as well. This language provides interface refinement. The interface of a system define signals which are exchanged to interact with other systems and can be refined to add new signals. Moreover, in order to reduce the number of states of the system, relations restricting input signals are provided. These restrictions are about incompatibility and master-slave combination of signals. In (BHATTACHARJEE et al., 1999), an graphical interface for Esterel was proposed. They integrate the graphical formalism of Statecharts with verification environments of Esterel, translating Statecharts specifications into Esterel program. CRP (BERRY; RAMESH; SHYAMASUNDAR, 1993) is a unification of Esterel and CSP (HOARE, 1978) languages, where their synchronous and asynchronous capabilities are combined. In this paradigm, a set of individually reactive synchronous processes is linked by asynchronous communication channels. The unification is given by a minor extension to the Esterel language. A primitive to permit asynchronous rendezvous is included, where the sending process emits a signal requesting for a rendezvous and receives rendezvous completion signals, by a given communication channel. The automatic verification of CRP programs is provided by a translation into a process calculus Meije (BOUDOL, 1985). These languages differ from dT-GTSs mainly by their control-driven nature that requires the designer to pay attention in control issues instead of concentrating on reactions of the system.

Synchronous Estelle is another language that merges both synchronous and asynchronous paradigms. The main idea is the same in CRP, but the asynchronous communication is given by means of message passing instead of rendezvous. Moreover, it is possible specify asynchronous systems, too. In Standard Estelle a system is viewed as a black-box and is specified by a state in a finite state machine which can be refined into a set of substates. Synchronous Estelle extends the standard one by including synchronous systems, which are specified by a state in a hierarchical state machine, i.e., a set of states (with a hierarchy) that is seen as a unique state. Using the synchronous version, it is possible specify reactive systems, while the communication between them is specified using the standard version. The execution of synchronous systems is divided into a set of computations steps, which are also divided into micro-steps, similar to the idea of transactions. At the beginning of the computation step, all messages arriving to the synchronous system are processed (possible generating internal events) and at the end all external messages generated by the computation are sent. The semantics of Synchronous Estelle is deterministic, using priorities on messages for choice purposes. The development environment of Synchronous Estelle includes a graphical (like Statecharts) editor, a compiler, a graphical animation. Moreover, they provide a translation from Synchronous Estelle to PROMELA (PROMELA LANGUAGE REFERENCE, 2008), for model checking purposes. The main difference from our approach is that this language is also control-driven, requiring to pay attention in control issues.

UML is a general-purpose visual modelling language that provides a complex family of diagrams to specify, construct and document the artifacts of a software system. Because of this complexity, usually, this family of diagrams is restricted by UML profiles. A UML profile is a subset of UML concepts which is adequate to define specific domains. Thus, most of the approaches to model reactive systems using this language propose a profile for this purpose. In general, class diagrams are used to define static aspects of the system and Statecharts or sequence diagrams are used to define dynamic ones. Moreover, despite of semi-formal semantics definition for UML, several approaches use translations to formal languages in order to define a formal semantics for this language, that is used as a graphical interface for visual specification. Most research on the formalisation of UML

refinements adhere to the approach of mapping the graphical notation into a formal domain, for example the works presented in (LEDANG; SOUQUIÈRES, 2002; USELTON; SMOLKA, 1994) among others. In (ALAVIZADEH; SIRJANI, 2006; ALAVIZAEDH; NEKOO; SIRJANI, 2007), a UML profile for reactive systems was proposed, using class and object diagrams to describe the structure of system and sequence diagrams to describe the behaviour of it. Besides, they propose a translation to Rebeca language (SIRJANI; MOVAGHAR, 2001), that, in its turn, has translations to PROMELA and Java languages, in order to provide automatic verification and code generation, respectively. A Rebeca model is an actor-based language, with independent reactive objects and asynchronous message passing. The communication is given by means of (buffered) messages and execution of atomic associated methods. They prefer to use sequence diagrams instead of Statecharts, since the latter is more appropriate to specify objects which have a interesting lifecycle and the reactive objects of Rebeca have a limited number of states (idle, waiting and running) and actions (sending a message). Thus, the behaviour of these objects can be better described by its message exchange pattern with its environment/other objects. In (KERSTEN et al., 2002), a translation of the UML profile for reactive systems to Ada (ADA 95 REFERENCE MANUAL, 1995) was proposed, to achieve automatic code generation. This profile uses class diagrams to specify static aspects and Statecharts to specify dynamic aspects. The notion of refinement of this approach is applied to translated code. Different of GTSs, UML is not a formal language and requires knowledge of other languages to have notions of refinement and verification. Moreover, different languages (diagrams) are necessary to describe static and dynamic aspects, while in GTSs both aspects are described in the same specification, avoiding to have to check the compatibility between these two points of view.

In (HECKEL, 1998), a different semantics for graph transformations was defined, which allows to represent effects of environment's events on the system states. This approach substitutes the pushout operation in definition of derivation by a pullback construction and permits to express more transformations than those specified by the systems' graph productions, simulating events from environment. The author proposes this semantics to provide a compositional verification of reactive systems, where a system is decomposed in views that anticipate the potential behaviour of the complete system. The composability of this approach ensures that properties of a complete system can be derived from those properties shown for its views. The interactivity with the environment is described only semantically and there is no construction in the formalism allowing to specify this interaction explicitly. Moreover, this approach produces an overhead on specification of properties, where, in addition of desired properties, some constraints must be described to ensure that the additional information generated in the behaviour of a view can be produced only by productions in the other views. A notion of refinement for typed GTSs was proposed in (HECKEL et al., 1996), where two systems are related by partial graph morphisms and each rule is mapped into another one. Moreover, the abstract production is required to be an instance of concrete production, i.e., the visible part of the refined production must not coincide with the original one. Therefore, the refinement relation guarantees only the existence of specialised transformations in the refining system. Another notion of refinement for GTSs was proposed in (GROSSE-RHODE; PARISI-PRESICCE; SIMEONI, 2000), where an abstract system is related to a concrete one by a total graph morphism and a mapping associating each abstract production to an expression over the name productions of the concrete system. Expressions are syntactical descriptions of sequential and parallel compositions of productions. The retyped produc-

tion of the abstract system must coincide with the refining production. This requirement guarantees that refinement relations preserve the full behaviour of abstract system. In this notion of refinement it is not possible to describe causality in the expressions describing composition of productions.

Abstract State Machines (GUREVICH, 1995) (ASM) are a mathematically defined, high-level environment for the system design, verification and analysis (LAMCH; WYRZYKOWSKI, 2006). An Abstract State Machine is a state machine which in each step computes a set of updates of variables from a specific vocabulary, in accordance with transaction rules. In one execution step, all updates are committed simultaneously. ASMs have been introduced in (GUREVICH, 1985) as "*a computation model that is more powerful and more universal than standard computation models*". Nowadays, there exist several extensions for ASMs, such as distributed, reactive and timed versions. In (BÖRGER; GLÄSSER, 1995), a predecessor of ASM, external functions are proposed to express environment effects in the system behaviour. A function is called external to a set of rules if it does not appear in any update in these rules. This kind of function is used to define the imported operations of a system and can be specified at any abstraction level. In this model is not possible to maintain a causal relation between elements of the system states, without adding explicit control to do this. In (MAIA; IORIO; BIGONHA, 1998), an extension of original model of ASMs is proposed, that explicitly enables the designer to define how the interactions occur between a system and its environment. There, a system is defined as a set of unit definitions and instances, where the units can be classified as system units (that are completely defined) and environment units (that are partially defined). A unit is composed by three parts: function, interaction and rules specifications. The interaction specification is defined by interaction operators that allow inputs and outputs within a unit, synchronisation and complex interaction patterns (using well-known composition operations: non-deterministic choice, parallel and sequential compositions). The environment units are restrictions of units, showing only the interaction specification, and play the role of interfaces of these units. There is also a notion of atomic reactions defined by interactions cycle. No notion of refinement is defined for this extension.

In the following section, we present some hints about how to improve our design environment.

5.3 Future work

The following open issues will be subject of future works:

- If we restrict our model to a special kind of GTSs called Object-Based Graph Grammars (OBGG), we can use a development environment for visual specification, simulation, automatic verification and code generation (DOTTI et al., 2005, 2006). In (DOTTI et al., 2006) an approach was proposed to verify partial systems using OBGGs based on the assume-guarantee approach. The basic idea is to see each part of a system as an open system - a system whose behaviour is not fully specified and that depends on interactions with its environment. In order to be possible we use this approach, it is necessary to describe the both output and input interactions. The former interaction is already specified by the interface and it is derived from the body behaviour. The input interaction (i.e., the reactions from the environment to the system events/signals) cannot be derived form the system specification, then it is necessary to include a mechanism to explicitly describe this kind of interaction.

It can be done by adding a new dependency relation for each production in a interface module. This extension can permit restrict the semantics of modules, since we have more dependencies to consider in order to complete the module behaviour.

- Besides, if we consider a dT-GTS and its abstraction related by an implementation morphism, as defined in Theorem 4.1, we can define a module notion, where the abstract system is the interface, describing the interaction between system and its environment; and the dT-GTS is the body that implements the interface. Moreover, in order to define notions of module composition, it is necessary to define first a notion of interaction compatibility, that requires that composed modules must have transactions of dual interaction patterns. Moreover, for two transactions (of different modules) holding to interaction compatibility, can exist synchronising elements, i.e., stable items that are created by one transaction and consumed by the other, before the end of former. In the composed module, these observable elements must become unobservable because the transactions must be merged. It is necessary to reasoning more about the resulting composed interface, as well as, the semantics of composed module.
- Other notions of refinement must be studied, for example, we can consider a T-GTS without dependency relation as an abstract specification and to include the dependency in a refinement of this specification. Other kind of refinement that can be considered is on the dependency relation: in the abstract level it can be less restrictive and in the concrete level it more restrictive.
- The OBGG model is extended, in (MICHELON; COSTA; RIBEIRO, 2006), with time notions. We must consider to apply this extension, in order to try to express synchronicity in the concrete level and preemption notions.

REFERENCES

ADA 95 Reference Manual. ISO/IEC 8652: 1995.ed. [S.l.]: US Government, 1995. Available at: http://www.adahome.com/Resources/References.html. Visited on: Apr. 2008.

ALAVIZADEH, S. F.; SIRJANI, M. Using UML to Develop Verifiable Reactive Systems. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING RESEARCH AND PRACTICE & CONFERENCE ON PROGRAMMING LANGUAGES AND COMPILERS, SERP, 2006, Las Vegas, USA. **Proceedings. . .** [S.l.]: CSREA Press, 2006. v.2, p.554–561.

ALAVIZAEDH, S. F.; NEKOO, A. H.; SIRJANI, M. ReUML: a UML profile for modeling and verification of reactive systems. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING ADVANCES, ICSEA, 2., 2007, Cap Esterel, France. **Proceedings. . .** Washington: IEEE Computer Society, 2007. p.50.

ANDRIES, M. et al. Graph Transformation for Specification and Programming. **Science of Computer Programming**, Amsterdam, The Netherlands, v.34, n.1, p.1–54, April 1999.

BALDAN, P. **Modelling Concurrent Computations**: from contextual petri nets to graph grammars. 2000. PhD Thesis — University of Pisa.

BALDAN, P.; CORRADINI, A.; DOTTI, F. L.; FOSS, L.; GADDUCCI, F.; RIBEIRO, L. Towards a Notion of Transaction in Graph Rewriting. In: INTERNATIONAL WORKSHOP ON GRAPH TRANSFORMATION AND VISUAL MODELING TECHNIQUES, GT-VMT, 5., 2006, Vienna, Austria. **Proceedings. . .** Amsterdam: Elsevier, 2008. p.39–50. (Electronic Notes in Theoretical Computer Science, v.211).

BALDAN, P.; CORRADINI, A.; FOSS, L.; GADDUCCI, F. Graph Transactions as Processes. In: INTERNATIONAL CONFERENCE ON GRAPH TRANSFORMATIONS, ICGT, 3., 2006, Natal, Brazil. **Proceedings. . .** Berlin: Springer, 2006. p.199–214. (Lecture Notes in Computer Science, v.4178).

BALDAN, P.; CORRADINI, A.; MONTANARI, U. Concatenable Graph Processes: relating processes and derivation traces. In: INTERNATIONAL COLLOQUIUM ON AUTOMATA, LANGUAGES AND PROGRAMMING, ICALP, 25., 1998, Aalborg, Denmark. **Proceedings. . .** Berlin: Springer, 1998. p.283–295. (Lecture Notes in Computer Science, v.1443).

BALDAN, P.; CORRADINI, A.; MONTANARI, U. Unfolding of Double-Pushout Graph Grammars is a Coreflection. In: INTERNATIONAL WORKSHOP ON THEORY AND

APPLICATION OF GRAPH TRANSFORMATIONS, TAGT, 6., 1998, Paderborn, Germany. **Proceedings...** Berlin: Springer, 1998. p.145–163. (Lecture Notes in Computer Science, v.1764).

BERRY, G. The foundations of Esterel. **Proof, language, and interaction: essays in honour of Robin Milner**, Cambridge, USA, p.425–454, 2000.

BERRY, G.; COURONNE, P.; GONTHIER, G. Synchronous programming of reactive systems: an introduction to esterel. In: FRANCO-JAPANESE SYMPOSIUM ON PROGRAMMING OF FUTURE GENERATION COMPUTERS, 1., 1986, Tokyo, Japan. **Proceedings...** Amsterdam: Elsevier, 1988. p.35–56.

BERRY, G.; RAMESH, S.; SHYAMASUNDAR, R. K. Communicating reactive processes. In: SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, POPL, 20., 1993, Charleston, USA. **Proceedings...** New York: ACM, 1993. p.85–98.

BHATTACHARJEE, A. K. et al. A Graphical Environment for the Specification and Verification of Reactive Systems. In: INTERNATIONAL CONFERENCE ON COMPUTER SAFETY, RELIABILITY AND SECURITY, SAFECOMP, 18., 1999, Toulouse, France. **Proceedings...** London: Springer, 1999. p.431–444.

BÖRGER, E.; GLÄSSER, U. **Modelling and Analysis of Distributed and Reactive Systems using Evolving Algebras**. Aarhus: University of Aarhus, 1995. Technical Report. (BRICS-NS-95-4).

BOUDOL, G. Notes on algebraic calculi of processes. **Logics and models of concurrent systems**, New York, p.261–303, 1985.

BROY, M.; STØLEN, K. **Specification and development of interactive systems**: focus on streams, interfaces, and refinement. New York: Springer, 2001.

BRUNI, R.; MONTANARI, U. Zero-Safe Nets: comparing the collective and individual token approaches. **Information and Computation**, Duluth, USA, v.156, n.1-2, p.46–89, 2000.

BRUNI, R.; MONTANARI, U. Transactions and zero-safe nets. In: EHRIG, H. et al. (Ed.). **Unifying Petri Nets, Advances in Petri Nets**. London: Springer, 2001. p.380–426. (Lecture Notes in Computer Science, v.2128).

CORRADINI, A.; DOTTI, F. L.; FOSS, L.; RIBEIRO, L. Translating Java Code to Graph Transformation Systems. In: INTERNATIONAL CONFERENCE ON GRAPH TRANSFORMATION, ICGT, 2., 2004, Roma, Italy. **Proceedings...** Berlin: Springer, 2004. p.383–398. (Lecture Notes in Computer Science, v.3256).

CORRADINI, A. et al. The Category of Typed Graph Grammars and its Adjunctions with Categories of Derivations. In: INTERNATIONAL WORKSHOP ON GRAPH GRAMARS AND THEIR APPLICATION TO COMPUTER SCIENCE, TAGT, 5., 1994, Williamsburg, USA. **Selected Papers.** Berlin: Springer, 1996. p.56–74. (Lecture Notes in Computer Science, v.1073).

CORRADINI, A. et al. Algebraic Approaches to Graph Transformation I: basic concepts and double pushout approach. In: ROZENBERG, G. (Ed.). **Handbook of Graph Grammars and Computing by Graph Transformation**. River Edge: World Scientific, 1997. v.1, p.163–245.

CORRADINI, A.; MONTANARI, U.; ROSSI, F. Graph Processes. **Fundamenta Informaticae**, Amsterdam, The Netherlands, v.26, n.3-4, p.241–265, 1996.

DIESTEL, R. **Graph Theory**. 2nd.ed. New York: Springer, 2005. (Graduate Texts in Mathematics).

DOTTI, F. L.; DUARTE, L. M.; FOSS, L.; RIBEIRO, L.; RUSSI, D.; SANTOS, O. M. dos. An environment for the development of concurrent object-based applications. In: INTERNATIONAL WORKSHOP ON GRAPH-BASED TOOLS, GRABATS, 2004, Rome, Italy. **Proceedings...** [S.l.]: Elsevier, 2005. n.1, p.3–13. (Electronic Notes In Theoretical Computer Science, v.127).

DOTTI, F. L.; RIBEIRO, L.; SANTOS, O. M. dos; PASINI, F. Verifying Object-based Graph Grammars: an assume-guarantee approach. **Software and Systems Modeling**, Berlin, v.5, n.3, p.289–311, September 2006. Special Section Paper.

DREWES, F. et al. Graph Transformation Modules and their Composition. In: INTERNATIONAL WORKSHOP ON APPLICATIONS OF GRAPH TRANSFORMATIONS WITH INDUSTRIAL RELEVANCE, AGTIVE, 1999, Kerkrade, The Netherlands. **Proceedings...** Berlin: Springer, 2000. p.15–30. (Lecture Notes in Computer Science, v.1779).

EHRIG, H.; ENGELS, G. **Towards a module concept for graph transformation systems**. Leiden, Netherlands: Leiden University, 1993. Technical Report. (TR93-34).

EHRIG, H.; ENGELS, G. Pragmatic and Semantic Aspects of a Module Concept for Graph Transformation Systems. In: INTERNATIONAL WORKSHOP ON GRAPH GRAMARS AND THEIR APPLICATION TO COMPUTER SCIENCE, TAGT, 5., 1994, Williamsburg, USA. **Selected Papers.** Berlin: Springer, 1996. p.137–154. (Lecture Notes in Computer Science, v.1073).

EHRIG, H. et al. (Ed.). **Handbook of Graph Grammars and Computing by Graph Transforpmation**. River Edge: World Scientific, 1999. v.2.

EHRIG, H. et al. (Ed.). **Handbook of Graph Grammars and Computing by Graph Transformation**. River Edge: World Scientific, 1999. v.3.

FILALI, M.; MAURAN, P.; PADIOU, G. Unity, as a Tool for Reactive Systems Specification and Derivation. In: EUROMICRO WORKSHOP ON REAL-TIME SYSTEMS, EWRTS, 5., 1993. **Proceedings...** [S.l.]: IEEE Press, 1993. p.274–279.

FOSS, L.; MACHADO, R.; RIBEIRO, L. Graph productions with dependencies. In: BRAZILIAN SYMPOSIUM ON FORMAL METHODS, SBMF, 10., 2007, Ouro Preto, Brazil. **Proceedings...** [S.l.: s.n.], 2007. p.128–143.

GAJSKI, D. D. et al. **Specc**: specification language and methodology. [S.l.]: Springer, 2000.

GIL, J. G.; FERRO, M. V.; BERNHARD, R. Communication protocols verification with Esterel. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING IN HIGHER EDUCATION, SEHE, 2., 1995, Alicante, Spain. **Proceedings...** Billerica: Computational Mechanics, 1996. p.255–265.

GOLTZ, U.; REISIG, W. The nonsequential behaviour of Petri nets. **Information and Control**, [S.l.], v.57, n.2-3, p.125–147, 1983.

GROSSE-RHODE, M.; PARISI-PRESICCE, F.; SIMEONI, M. Refinements and Modules for Typed Graph Transformation Systems. In: INTERNATIONAL WORKSHOP ON RECENT TRENDS IN ALGEBRAIC DEVELOPMENT TECHNIQUES, WADT, 13., 1998, Lisbon, Portugal. **Selected Papers.** Berlin: Springer, 1999. p.138–151. (Lecture Notes in Computer Science, v.1589).

GROSSE-RHODE, M.; PARISI-PRESICCE, F.; SIMEONI, M. Refinements of Graph Transformation Systems via Rule Expressions. In: INTERNATIONAL WORKSHOP ON THEORY AND APPLICATION OF GRAPH TRANSFORMATIONS, TAGT, 6., 1998, Paderborn, Germany. **Selected Papers.** London: Springer, 2000. p.368–382. (Lecture Notes in Computer Science, v.1764).

GUREVICH, Y. A New Thesis. **American Mathematical Society Abstracts**, [S.l.], p.317, August 1985.

GUREVICH, Y. Evolving Algebras 1993: Lipari Guide. In: BÖRGER, E. (Ed.). **Specification and Validation Methods**. [S.l.]: Oxford University Press, 1995. p.9–37.

HABEL, A.; MÜLLER, J.; PLUMP, D. Double-Pushout Graph Transformation Revisited. **Mathematical Structures in Computer Science**, [S.l.], v.11, n.5, p.637–688, October 2001.

HALBWACHS, N. et al. The synchronous data-flow programming language LUSTRE. **Proceedings of the IEEE**, [S.l.], v.79, n.9, p.1305–1320, September 1991.

HAREL, D. Statecharts: a visual formalism for complex systems. **Science of Computer Programming**, Amsterdam, The Netherlands, v.8, n.3, p.231–274, 1987.

HECKEL, R. Compositional Verification of Reactive Systems Specified by Graph Transformation. In: INTERNATIONAL CONFERENCE ON FUNDAMENTAL APPROACHES TO SOFTWARE ENGINEERING, FASE, 1., 1998, Lisbon, Portugal. **Proceedings...** Berlin: Springer, 1998. p.138–153. (Lecture Notes in Computer Science, v.1382).

HECKEL, R. et al. Horizontal and Vertical Structuring of Typed Graph Transformation Systems. **Mathematical Structures in Computer Science**, [S.l.], v.6, n.6, p.613–648, 1996.

HECKEL, R. et al. Simple Modules for GRACE. In: INTERNATIONAL WORKSHOP ON THEORY AND APPLICATION OF GRAPH TRANSFORMATIONS, TAGT, 6., 1998, Paderborn, Germany. **Proceedings...** Berlin: Springer, 1998. p.383–395. (Lecture Notes in Computer Science, v.1764).

HECKEL, R. et al. Classification and Comparison of Module Concepts for Graph Transformation Systems. In: EHRIG, H. et al. (Ed.). **Handbook of Graph Grammars and Computing by Graph Transformation**. River Edge: World Scientific, 1999. v.2, p.669–689.

HOARE, C. A. R. Communicating sequential processes. **Communications of the ACM**, New York, v.21, n.8, p.666–677, August 1978.

KERSTEN, M. et al. Customizing UML for the development of distributed reactive systems and code generation to Ada 95. **Ada User Journal**, [S.l.], v.23, n.3, September 2002.

KREOWSKI, H.-J.; KUSKE, S. Graph Transformation Units and Modules. In: EHRIG, H. et al. (Ed.). **Handbook of Graph Grammars and Computing by Graph Transformation**. River Edge, USA: World Scientific, 1999. v.2, p.607–638.

KREOWSKI, H.-J.; KUSKE, S. On the interleaving semantics of transformation units - A step into GRACE. In: INTERNATIONAL WORKSHOP ON GRAPH GRAMARS AND THEIR APPLICATION TO COMPUTER SCIENCE, TAG, 5., 1994, Williamsburg, USA. **Selected Papers.** Berlin: Springer, 1996. (Lecture Notes in Computer Science, v.1073).

LAMCH, D.; WYRZYKOWSKI, R. Specification, analysis and testing of grid environments using Abstract State Machines. In: INTERNATIONAL SYMPOSIUM ON PARALLEL COMPUTING IN ELECTRICAL ENGINEERING, PARELEC, 2006, Bialystok, Poland. **Proceedings...** Washington: IEEE Computer Society, 2006. p.116–120.

LEDANG, H.; SOUQUIÈRES, J. Integration of UML and B Specification Techniques: systematic transformation from ocl expressions into B. In: ASIA-PACIFIC SOFTWARE ENGINEERING CONFERENCE, APSEC, 9., 2002, Gold Coast, Australia. **Proceedings...** Washington: IEEE Computer Society, 2002. p.495–504.

LEGUERNIC, P. et al. Programming real-time applications with SIGNAL. **Proceedings of the IEEE**, [S.l.], v.79, n.9, p.1321–1336, September 1991.

MAIA, M. A.; IORIO, V. O.; BIGONHA, R. S. Interacting Abstract State Machines. In: INTERNATIONAL WORKSHOP ON ABSTRACT STATE MACHINES, ASM, 5., 1998, Magdenburg, Germany. **Proceedings...** Magdenburg: Magdeburg University, 1998. p.37–49.

MANNA, Z.; PNUELI, A. **The Temporal Logic of Reactive and Concurrent Systems**: specification. Berlin: Springer, 1992.

MICHELON, L.; COSTA, S. A. da; RIBEIRO, L. Specification of Real-Time Systems with Graph Grammars. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, SBES, 20., 2006, Florianópolis, Brasil. **Anais...** [S.l.: s.n.], 2006. p.97–112.

PNUELI, A. In transition from global to modular temporal reasoning about programs. In: KRZYSZTOF, R. A. (Ed.). **Logics and Models of Concurrent Systems**. New York: Springer, 1985. p.123–144. (Nato Asi Series F: Computer And Systems Sciences, v.13).

PROMELA Language Reference. [S.l.: s.n.], 2008. Available at: http://cm.bell-labs.com/cm/cs/what/spin/Man/promela.html. Visited on : Apr. 2008.

RIBEIRO, L. **Parallel composition and unfolding semantics of graph grammars**. 1996. PhD Thesis — Technical University of Berlin.

RIESCO, M.; TUYA, J. Synchronous Estelle: just another synchronous language? In: SYNCHRONOUS LANGUAGES, APPLICATIONS AND PROGRAMMING, SLAP, 2., 2003, Porto, Portugal. **Proceedings...** [S.l.]: Elsevier, 2004. p.71–86. (Electronic Notes in Theoretical Computer Science, v.88).

ROZENBERG, G. (Ed.). **Handbook of Graph Grammars and Computing by Graph Transformation**. River Edge: World Scientific, 1997. v.1.

SCHOLZ, P. A refinement calculus for statecharts. In: INTERNATIONAL CONFERENCE FUNDAMENTAL APPROACHES TO SOFTWARE ENGINEERING, FASE, 1., 1998, Lisbon, Portugal. **Proceedings...** Berlin: Springer, 1998. p.285–301. (Lecture Notes in Computer Science, v.1382).

SCHÜRR, A. PROGRESS: a VHL-language based on graph grammars. In: INTERNATIONAL WORKSHOP ON GRAPH-GRAMMARS AND THEIR APPLICATION TO COMPUTER SCIENCE, 4., 1990, Bremen, Germany. **Proceedings...** London: Springer, 1991. p.641–659. (Lecture Notes in Computer Science, v.532).

SCHÜRR, A.; WINTER, A. J. UML Packages for PROgrammed Graph REwriting Systems. In: INTERNATIONAL WORKSHOP ON THEORY AND APPLICATION OF GRAPH TRANSFORMATIONS, TAGT, 6., 1998, Paderborn, Germany. **Selected Papers.** Berlin: Springer, 2000. p.396–410. (Lecture Notes in Computer Science, v.1764).

SCHÜRR, A.; WINTER, A.; ZÜNDORF, A. The PROGRES Approach: language and environment. In: EHRIG, H. et al. (Ed.). **Handbook of Graph Grammars and Computing by Graph Transformation**. River Edge, NJ, USA: World Scientific, 1999. v.2, p.547–668.

SECELEANU, C. C.; SECELEANU, T. Synchronization Can Improve Reactive Systems Control and Modularity. **Universal Computer Science**, [S.l.], v.10, n.10, p.1429–1468, 2004.

SEKERINSKI, E. Graphical Design of Reactive Systems. In: INTERNATIONAL B CONFERENCE ON RECENT ADVANCES IN THE DEVELOPMENT AND USE OF THE B METHOD, B, 2., 1998, Montpellier, France. **Proceedings...** London: Springer, 1998. p.182–197. (Lecture Notes in Computer Science, v.1393).

TAYLOR, I. J. et al. (Ed.). **Control- Versus Data-Driven Workflows**. London: Springer, 2007. p.167–173.

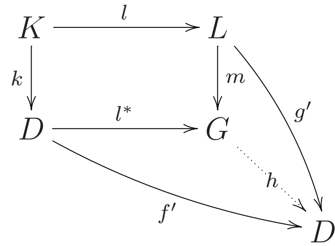
SIRJANI, M.; MOVAGHAR, A. **An Actor-Based Model for Formal Modelling of Reactive Systems**: Rebeca. Tehran, Iran: Computer Engineering Dept, Sharif University of Technology, 2001. Technical Report. (CS-TR-80-01).

TAENTZER, G.; SCHÜRR, A. DIEGO, another step towards a module concept for graph transformation systems. In: JOINT COMPUGRAPH/SEMAGRAPH WORKSHOP ON GRAPH REWRITING AND COMPUTATION, SEGRAGRA, 1995, Volterra, Italy. **Proceedings...** [S.l.]: Elsevier, 1995. p.277–285. (Electronic Notes Theoretical Computer Science, v.2).

USELTON, A. C.; SMOLKA, S. A. A Process Algebraic Semantics for Statecharts via State Refinement. In: IFIP WORKING CONFERENCE ON PROGRAMMING CONCEPTS, METHODS AND CALCULI, PROCOMET, 1994, San Miniato, Italy. **Proceedings...** Amsterdam: North-Holland Publishing, 1994. p.267–286. (IFIP Transactions, v.A-56).

APPENDIX A CATEGORICAL DEFINITIONS

Definition A.1 (Pushout and pushout complement) *(CORRADINI et al., 1997) Given a category* $\mathcal{C}$ *and two arrows* $l : K \to L$ *and* $k : K \to D$ *of* $\mathcal{C}$*, a triple* $\langle G, l^* : D \to G, m : L \to G \rangle$ *as in the diagram below is called a pushout of* $\langle l, k \rangle$ *if*



Commutativity $m \circ l = l^* \circ k$*, and*

Universal Property *for all objects* G' *and arrows* $g' : L \to G'$ *and* $f' : D \to G'$*, with* $g' \circ l = f' \circ k$*, there exists a unique arrow* $h : G \to G'$ *such that* $h \circ m = g'$ *and* $h \circ l^* = f'$*.*

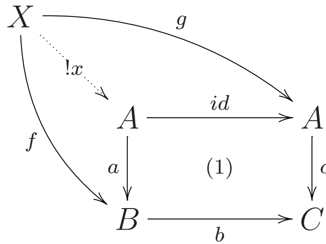
In this situation, G *is called a* pushout object *of* $\langle l, k \rangle$*. Moreover, given the arrows* $l : K \to L$ *and* $m : L \to G$*, a* pushout complement *of* $\langle l, m \rangle$ *is the triple* $\langle D, k : K \to D, l^* : D \to G \rangle$ *such that* $\langle G, m, l^* \rangle$ *is a pushout of* $\langle l, k \rangle$*. In this case* D *is called a* pushout complement *of* $\langle l, m \rangle$*.*

Proposition A.1 (Existence and uniqueness of pushout complement) *(CORRADINI et al., 1997) Let* $l : K \to L$ *and* $m : L \to G$ *be two morphisms in* T-**Graph***, where* m *is injective. Then there exists a pushout complement* $\langle D, k : K \to D, l^* : D \to G \rangle$ *of* $\langle l, m \rangle$ *iff the following condition is satisfied:*

Dangling condition *No edge* $e \in E_G - m_E(E_L)$ *is incident to any vertex in* $g_V(V_L - l_v(V_K))$*.*

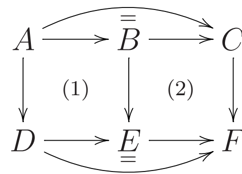
In this case, $\langle l, m \rangle$ *satisfies the* gluing condition *(or* m *satisfies the gluing condition with respect to* l*). If morphism* l *is injective, then the pushout complement is unique up to isomorphism.*

Lemma A.1 *Considering the following commuting diagram, with b mono, then (1) is a pullback in any category:*



Proof: For all object X and two morphisms g and f, such that $c \circ g = b \circ f$, then $\exists! x : X \to A$ such that $id \circ x = g$ and $a \circ x = f$. By definition of identity $x = g$, then it remains to prove $a \circ g = f$. By commutativity of (1), then $b \circ a \circ g = b \circ f$. Since b is mono, then $a \circ g = f$ as we want to prove. □

In the following, we consider the following commutative diagram:

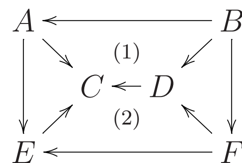


Lemma A.2 (composition of pushouts and pullbacks). *(HABEL; MÜLLER; PLUMP, 2001) If diagrams (1) and (2) are pushouts (pullbacks) then $(1{+}2)$ is a pushout (pullback) as well.*

Lemma A.3 (decomposition of pushouts and pullbacks). *If diagrams $(1 + 2)$ and (1) are pushouts then (2) is a pushout. If diagrams $(1 + 2)$ and (2) are pullbacks then (1) is a pullback (HABEL; MÜLLER; PLUMP, 2001).*

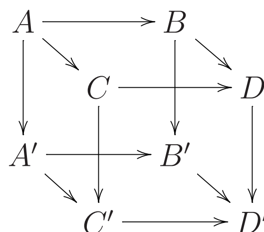
Lemma A.4 (Special decomposition). *(HABEL; MÜLLER; PLUMP, 2001) If diagrams $(1 + 2)$ and (2) are pushouts and $B \to C$ is injective, then (1) is a pushout. If diagrams $(1 + 2)$ is pullback, (1) is a pushout, and $C \to F$ is injective, then (2) is a pullback.*

Lemma A.5 *Consider the following commuting diagram in* $\mathbf{Set}$



If the two internal squares (1) and (2) are pullbacks, then the outer square with vertices A, B, E and F is a pullback as well ().

Lemma A.6 (3-cube lemma) *(CORRADINI et al., 1996) Consider the following commuting diagram in* $\mathbf{Set}$*, with bottom and top morphisms injective.*



We have the following 3-cube lemmata:

1. *If the bottom square is a pushout, the back and left squares are pullbacks, then the top square is a pushout if and only if the front and right squares are pullbacks.*

2. *If the top square is a pullback, the front and right squares are pushouts, then the bottom square is a pullback if and only if the back and left squares are pushouts.*

APPENDIX B PROPER QUOTIENT PRODUCTIONS

In the following, for a set P of productions, we will write $G \stackrel{P}{\Rightarrow} H$ when there is a direct derivation from G to H using a production in P.

Definition B.1 (Quotient production) *Given a production $q_1 = L_1 \stackrel{l_1}{\leftarrow} K_1 \stackrel{r_1}{\rightarrow} R_1$, a production $q_2 = L_2 \stackrel{l_2}{\leftarrow} K_2 \stackrel{r_2}{\rightarrow} R_2$ is a quotient production of q_1 if there are two pushouts (in $\mathbf{Graph}$) of the form*

$$\begin{array}{ccccc} L_1 & \longleftarrow & K_1 & \longrightarrow & R_1 \\ \downarrow & (1) & \downarrow & (2) & \downarrow \\ L_2 & \longleftarrow & K_2 & \longrightarrow & R_2 \end{array}$$

where the vertical morphisms are surjective. The set of quotient productions of q_1 is denoted by $Q(q_1)$.

By the next lemma we have that every application of a production corresponds to an application of one of its quotient productions obeying the injectivity restriction. So, if there is a derivation using a quotient production, then there is an equivalent derivation using the original production. Moreover, if there is a derivation using a production based on an arbitrary match, then there exists an equivalent derivation using some quotient production of it based on an injective match.

Lemma B.1 (Quotient Lemma) *For all graphs G and H, and all production q:*

1. $G \stackrel{Q(q)}{\Rightarrow} H$ *implies* $G \stackrel{q}{\Rightarrow} H$;
2. $G \stackrel{q,m}{\Rightarrow} H$ *implies* $G \stackrel{q',m'}{\Rightarrow} H$ *for* $q' \in Q(q)$ *and some injective* m'.

The construction of proper quotient production for a parallel production can be find in (HABEL; MÜLLER; PLUMP, 2001).

APPENDIX C TRANSACTIONS OF A DT-GTS

In this chapter we define the functions used to construct the set of transactions of a dT-GTS. We do an analysis of termination for each function, after its definition, when is necessary, i.e., when there are loops in its definition.

Definition C.1 (Initialisation) *Let $P_\mathcal{Z}$ be a set of production. The initialisation of set of transactions, denoted by $init(P_\mathcal{Z})$, is defined by the pair $\langle P', \mathbf{DwtProc}(\mathcal{Z})\rangle$ as in the following:*

1: ***for all*** $p \in P_\mathcal{Z}$ ***do*** ▷ *for each production p in $P_\mathcal{Z}$*
2: ***if*** $\mathcal{S}(p) = p$ ***then*** ▷ *if this production is stable*
3: $\phi_{id_p} \in \mathbf{DwtProc}(\mathcal{Z})$ ▷ *then include in the set of transactions of $\mathcal{Z}$ the process containing p as its unique production*
4: $P_\mathcal{Z} = P_\mathcal{Z} - \{p\}$ ▷ *exclude production p from set of productions $P_\mathcal{Z}$*
5: ***end if***
6: ***end for***
7: $P' = \mathcal{P}(P_\mathcal{Z})$ ▷ *assign to P' the power set of remaining productions in $P_\mathcal{Z}$*
8: ***for all*** $A \in P'$ ***do*** ▷ *for all subset A of $P_\mathcal{Z}$*
9: ***if*** $\nexists p \in A \,.\, (\mathcal{S}(L_p) = L_p \vee \mathcal{S}(R_p) = R_p)$ ***then*** ▷ *if there is no production with left-hand side stable or no production with right-hand side stable*
10: $P' = P' - \{A\}$ ▷ *then the subset of productions cannot compose a transaction and it is excluded from P'*
11: ***end if***
12: ***end for***

The function defined above always stops since we can see that the executions of all loops are controlled by finite structures (set of productions $P_\mathcal{Z}$ and P').

Definition C.2 (Making a process) *Let $\mathcal{Z} = \langle\langle T_\mathcal{Z}, P_\mathcal{Z}, \pi_\mathcal{Z}\rangle, T_{\mathcal{Z}s}\rangle$ be a dT-GTS, $\phi = \langle\langle T_\phi, P_\phi, \pi_\phi\rangle, T_{\phi_s}\rangle \to \mathcal{Z}$ be a graph process, $\langle p, k\rangle$ be a pair where $p \in P_\mathcal{Z}$ and $m : L_p^{T_\mathcal{Z}} \to \langle T_\phi, r_{\phi_T}\rangle$ be a match. For each $\mathcal{Z}$, the process constructed by applying $\langle p, k\rangle$ in T_ϕ based on m, denoted by $make_\mathcal{Z}(\phi, \langle p, k\rangle, m)$, is defined by ϕ_{aux} as follows:*

1: $T_{aux} = glue_{\langle p,k\rangle}(p, m, T_\phi)$ ▷ *construct T_{aux} applying $\langle p, k\rangle$ in T_ϕ based on m*
2: $P_{aux} = P_\phi \cup \{\langle p, k\rangle\}$ ▷ *assign to P_{aux} the set of productions of ϕ plus $\langle p, k\rangle$*
3: $\pi_{aux} = \pi_\phi \cup \{(\langle p, k\rangle, \langle L_p, m\rangle \hookleftarrow \langle K_p, m_{|K_p}\rangle \to \langle R_p, t\rangle)\}$, *where*

$\pi_\mathcal{Z}(p) = L_p^{T_\mathcal{Z}} \hookleftarrow K_p^{T_\mathcal{Z}} \to R_p^{T_\mathcal{Z}}$ *and* $t\colon R_p \to T_\phi$ *is defined by*

$$t(x) = \begin{cases} m(x), & \text{if } x \in K_p; \\ \langle x, \langle p, k\rangle\rangle, & \text{otherwise.} \end{cases}$$

▷ *associate production name* $\langle p,k\rangle$ *to the spam* $L_{\langle p,k\rangle} \hookleftarrow K_{\langle p,k\rangle} \rightarrow R_{\langle p,k\rangle}$, *where* $L_{\langle p,k\rangle}$ *and* $K_{\langle p,k\rangle}$ *are typed over* T_{aux} *by* m *and* $R_{\langle p,k\rangle}$ *is typed over* T_{aux} *by* m *on preserved items and the created items* x *are mapped to* $\langle x, \langle p,k\rangle\rangle$ *(item included in* T_{aux} *by application of* $\langle p,k\rangle$*)*

4: $\phi_{auxT} = \phi_T \cup \{(\langle x, \langle p,k\rangle\rangle, t_{R_p}(x)) \mid x \in R_p - rng(r_p)\}$ ▷ *include in the type graph mapping the elements created by* $\langle p,k\rangle$

5: $\phi_{auxP} = \phi_P \cup \{(\langle p,k\rangle, p)\}$ ▷ *include in the production mapping the production* $\langle p,k\rangle$

Definition C.3 (Concurrent graph) *Let* ϕ *be a graph process. A subgraph* G *of* T_ϕ *is called concurrent if* G *is a subgraph of a graph reachable from* $Min(\phi)$ *by means of a derivation which applies all productions in* $\bigcup_{x \in G} \lfloor x \rfloor$, *where* $\lfloor x \rfloor = \{p \mid p \in P_\phi \wedge p \preceq_\phi x\}$

Definition C.4 (Gluing) *Let* $p : L_p \hookleftarrow K_p \rightarrow R_p$ *be a production,* G *a graph,* $m\colon L_p \rightarrow G$ *a graph morphism and* $*$ *any symbol. The gluing of* G *and* R_p*, according to* m *and marked by* $*$*, denoted by* $glue_*(p, m, G)$ *is the graph* $\langle V, E, s, t\rangle$*, where:*

$$V = \{V_G \cup m_*(V_{R_p})\} \qquad E = \{E_G \cup m_*(E_{R_p})\}$$

with m_* *defined by:*

$$m_*(x) = \begin{cases} m(x), & \text{if } x \in K_p; \\ \langle x, *\rangle, & \text{otherwise.} \end{cases}$$

The source and target functions are inherited from G *and* R_p.

Definition C.5 (Pre-transactions) *Let* A *be a set of* T-**Graph** *productions. The list of initial graphs and productions, which can possible constitute a transaction, based on* A*, denoted by* $preTransactions(A)$*, is the list* l *of graphs and production sets, defined as follows:*

1: $X = \{\{\langle p, 1\rangle \mid p \in A\}\}$ ▷ *initialise* X *with one set of productions containing one instance of each production in* A
2: $l = \lambda$ ▷ *initialise the list of pre-transactions (initial graph and production instances that can constitute a transaction) with an empty list*
3: $ok =$ **false** ▷ *initialise ok indicating that the list of pre-transactions is not complete*
4: ***while*** $\neg ok$ ***do*** ▷ *repeat while there are new production instances to be considered to construct the pre-transactions*
5: $ok =$ **true** ▷ *assign* **true** *to ok, that is set to* **false** *if new instances are need*
6: $X_{aux} = \varnothing$ ▷ *assign empty set to* X_{aux} *(set containing sets of production instances generated in order to balance productions in all* X' *in* X*)*
7: ***for all*** $X' \in X$ ***do*** ▷ *for each set of production instances* X' *in* X
8: ***for all*** $\langle p,k\rangle \in X'$ ***do*** ▷ *for each production instance* $\langle p,k\rangle$ *in* X'
9: ***if*** $x \in L_p - \mathcal{S}(L_p) - K_p$ ***then***
10: $\langle x, \langle p,k\rangle\rangle \in L$ ▷ *include in* L *(set of unstable elements consumed by productions in* X'*) all unstable items consumed by* $\langle p,k\rangle$
11: ***end if***
12: ***if*** $x \in R_p - \mathcal{S}(R_p) - K_p$ ***then***
13: $\langle x, \langle p,k\rangle\rangle \in R$ ▷ *include in* R *(set of unstable elements created by productions in* X'*) all unstable items created by* $\langle p,k\rangle$*, and*
14: ***end if***

15: ***if*** $x \in K_p - \mathcal{S}(K_p)$ ***then***

16: $\langle x, \langle p, k \rangle \rangle \in K$ $\triangleright$ *include in* K *(set of unstable elements preserved by productions in* X'*) all unstable items preserved by* $\langle p, k \rangle$

17: ***end if***

18: ***end for***

19: ***if*** $\forall \langle x, \langle p, k \rangle \rangle \in K \centerdot \exists \langle y, \langle q, k' \rangle \rangle \in R \centerdot t_{K_p}(x) = t_{R_q}(y) \wedge$
$\forall \langle x, \langle p, k \rangle \rangle \in L \centerdot \exists \langle y, \langle q, k' \rangle \rangle \in R \centerdot t_{L_p}(x) = t_{R_q}(y) \wedge$
$\forall \langle y, \langle q, k' \rangle \rangle \in R \centerdot \exists \langle x, \langle p, k \rangle \rangle \in L \centerdot t_{R_q}(y) = t_{L_p}(x)$ ***then***
$\triangleright$ *if all unstable items preserved/consumed (created) by production instaces in* X' *are created (consumed) by some production instance in* X'

20: ***for all*** $\langle x, \langle p, k \rangle \rangle \in L$ ***do*** $\triangleright$ *then, for each consumed element* x *in* L

21: ***if*** $\exists \langle y, \langle q, k' \rangle \rangle \in R \centerdot t_{l_p}(x) = t_{R_q}(y)$ ***then*** $\triangleright$ *if there is one created element* y *with the same type of* x

22: $L = L - \{\langle x, \langle p, k \rangle \rangle\}$ $\triangleright$ *exclude* x *consumed by* $\langle p, k \rangle$ *from set of elements consumed by productions in* X'

23: $R = R - \{\langle y, \langle q, k' \rangle \rangle\}$ $\triangleright$ *exclude* y *created by* $\langle q, k' \rangle$ *from set of elements created by productions in* X'

24: ***end if***

25: ***end for***

26: ***if*** $L = \varnothing$ *and* $R = \varnothing$ ***then*** $\triangleright$ *if all unstable elements consumed (created) by production instances in* X' *are created (consumed) by some production instance in* X'

27: $l = l \centerdot InitGraph(X')$ $\triangleright$ *then, add to list of pre-transactions, the list of pairs of initial graphs and productions of* X'*, and*

28: $X_2 = \varnothing$ $\triangleright$ *assign empty set to* X_2 *(set containing sets of production instances generate in order to balance all production instances of each* X'*), indicating that is not necessary to add new production instances to* X'

29: ***else***

30: $X_1 = \{X'\}$ $\triangleright$ *otherwise, initialise* X_1 *(set containing sets of production instances genereted for each* X' *in order to balance the consumed elements with created ones, if it is necessary) with the set of production instances* X'

31: ***if*** $L \neq \varnothing$ ***then*** $\triangleright$ *and, if there are consumed elements that are not created by some production instance in* X'

32: ***for all*** $\langle x, \langle p, k \rangle \rangle \in L$ ***do*** $\triangleright$ *then, for each consumed element remaining in* L

33: ***for all*** $C \in X_1$ ***do*** $X_C = \varnothing$ $\triangleright$ *initialise an empty set* X_C *for each set of production instances in* X_1

34: ***end for***

35: ***for all*** $C \in X_1$ ***do*** $\triangleright$ *for each set of production instances in* X_1 *and*

36: ***for all*** $q \in A$ ***do*** $\triangleright$ *for each production* q *in* A

37: ***if*** $\exists y \in R_q - K_q \centerdot t_{L_p}(x) = t_{R_q}(y)$ ***then*** $\triangleright$ *such that* q *creates elements of same type that* x

38: $X_C = \{\{\langle q, \#D + 1 \rangle\} \cup C\} \cup X_C$,
where $D = \{\langle q, i \rangle \mid \langle q, i \rangle \in C\}$
$\triangleright$ *include a new instance of* q *in* X_C *(set containing sets of production instances generated for each* C *in* X_1 *in order to create items in* L*), and*

39: $ok = \text{false}$ $\triangleright$ *assign* false *to* ok*, indicating that there are new production instances to be considered*

40: **end if**
41: **end for**
42: **end for**
43: $X'_1 = \bigcup_{C \in X_1} X_C$
44: $X_1 = X'_1$ ▷ *after generate all sets of production instances in X_C, uptdate X_1 with sets of production instances genereted in order to create all items $\langle x, \langle p, k \rangle \rangle$ in L*
45: **end for**
46: **end if**
47: $X_2 = X_1$ ▷ *initialise X_2 with all new sets of production instances that creates all elements in L*
48: **if** $R \neq \varnothing$ **then** ▷ *if there are created elements that are not consumed by some production instance in X'*
49: **for all** $\langle y, \langle q, k' \rangle \rangle \in R$ **do** ▷ *then, for each created element remaining in R*
50: **for all** $C \in X_2$ **do** $X_C = \varnothing$ ▷ *initialise an empty set X_C for each set of production instances in X_2*
51: **end for**
52: **for all** $C \in X_2$ **do** ▷ *for each set of production instances in X_2 and*
53: **for all** $p \in A$ **do** ▷ *for each production p in A*
54: **if** $\exists x \in L_q - K_q \centerdot t_{R_q}(y) = t_{L_p}(x)$ **then** ▷ *such that p creates elements of same type that y*
55: $X_C = \{\{\langle p, \#D + 1 \rangle\} \cup C\} \cup X_C$, *where* $D = \{\langle p, i \rangle \mid \langle p, i \rangle \in C\}$ ▷ *include a new instance of p in X_C (set containing sets of production instances generated for each C in X_2 in order to consume items in R), and*
56: $ok = \textbf{false}$ ▷ *assign* **false** *to ok, indicating that there are new production instances to be considered*
57: **end if**
58: **end for**
59: **end for**
60: $X'_2 = \bigcup_{C \in X_2} X_C$
61: $X_2 = X'_2$ ▷ *after generate all sets of production instances in X_C, uptdate X_2 with sets of production instances genereted in order to consume all items $\langle y, \langle q, k' \rangle \rangle$ in R*
62: **end for**
63: **end if**
64: **end if**
65: **else**$X_2 = \varnothing$ ▷ *if there is an unstable element consumed/preserved (created) by production instances in X' that is not created (consumed) by some other instance production in X', assign empty set to X_2 (it is because the considered set of instance productions cannot constitute a transaction)*
66: **end if**
67: $X_{aux} = X_{aux} \cup X_2$ ▷ *update X_{aux} with the sets of production instances generate in order to balance the productions in X'*
68: **end for**
69: $X = X_{aux}$ ▷ *update X with new sets of production instances generate in the previous iteration in order to balance the productions in all X' in X*

70: ***end while***

In the function above, almost all loops are controlled by finite structures, but **while** loop (at lines $4 - 70$) is repeated up to $x = true$. x is set to $true$ at the begin of each iteration and becomes $false$ when an instance of a production is included to compensate a consumed/created item (lines $37 - 40$ and $54 - 57$). Since productions of considered T-GTSs do not have cycles on creation and consuming of unstable items, the number of needed instances of each production in a transaction is always finite. Consequently, x eventually is not set to $false$ and the loop stops.

Definition C.6 (Initial graphs) *Let X be a set of sets of T-*Graph *productions which have balanced unstable items. The possible initial graphs and for each $X' \in X$, denoted by $InitGraph(X)$, is given by list l of graphs and production sets, defined as follows:*

1: $l = \lambda$ ▷ *initialise the list of initial graphs and productions with an empty list*
2: ***for all*** $X' \in X$ ***do*** ▷ *for each set of production instances in X*
3: $ord = order(X')$ ▷ *order the production instances based on possibility of application (created and consumed unstable items)*
4: ***for all*** $G \in GI(ord)$ ***do*** ▷ *for each initial graph obtained from ordered productions in X'*
5: $l = \langle\langle G, X'\rangle\rangle \centerdot l$ ▷ *include the pair of initial graph and productions in list l*
6: ***end for***
7: ***end for***

where $order(X')$ orders the productions X' based on typing of unstable items, which is defined by l as follows:

1: $l = \lambda$ ▷ *initialise the ordered list of productions in X'*
2: $U = \varnothing$ ▷ *initialise the current unstable items (set of items created by considered productions that are not consumed by them)*
3: ***for all*** $\langle p, k\rangle \in X'$ ***do*** ▷ *for each production in X'*
4: ***if*** $L_p = \mathcal{S}(L_p)$ ***then*** ▷ *if its left-hand side is stable then*
5: $l = l \centerdot \langle\langle p, k\rangle\rangle$ ▷ *include it in the list l, and*
6: $U = U \cup \{\langle x, \langle p, k\rangle\rangle \mid x \in R_p - \mathcal{S}(R_p) - K_p\}$ ▷ *include in U the unstable items created by the production, and*
7: $X' = X' - \{\langle p, k\rangle\}$ ▷ *exclude the production from X'*
8: ***end if***
9: ***end for***
10: ***while*** $X' \neq \varnothing$ ***do*** ▷ *repeat while there is some production in X' that is not used*
11: $U_R = \varnothing$ ▷ *initialise the set of unstable items created by all enabled productions in each iteration*
12: ***for all*** $\langle p, k\rangle \in X'$ ***do*** ▷ *for each production $\langle p, k\rangle$ in X' that can be applied in the graph containing the current items in U*
13: $C_p = \{x \mid x \in L_p - \mathcal{S}(L_p)\}$ ▷ *assign the set of unstable items consumed/preserved by $\langle p, k\rangle$ to C_p*
14: $U_{aux} = U$ ▷ *assign the set of current items to U_{aux}, which will be used to update U excluding items consumed by $\langle p, k\rangle$*
15: $C = \varnothing$ ▷ *initialise the set of unstable items consumed/preserved by $\langle p, k\rangle$ for which there exists a corresponding item in U_{aux}*
16: ***for all*** $x \in C_p$ ***do*** ▷ *for each unstable item x consumed/preserved by $\langle p, k\rangle$*
17: ***if*** $\exists\langle y, \langle q, j\rangle\rangle \in U_{aux} \centerdot t_{L_p}(x) = t_{R_q}(y)$ ***then*** ▷ *if there is an item in U_{aux} with same type of x*

18: **if** $x \in L_p - K_p$ **then** ▷ *then, if x is consumed by $\langle p, k\rangle$*
19: $U_{aux} = U_{aux} - \{\langle y, \langle q, j\rangle\rangle\}$ ▷ *then, exclude the item with same type of x from U_{aux}, and*
20: **end if**
21: $C = C \cup \{x\}$ ▷ *include x in C*
22: **end if**
23: **end for**
24: **if** $C = C_p$ **then** ▷ *if all unstable items consumed/preserved by $\langle p, k\rangle$ are currently in U_{aux}*
25: $l = l \centerdot \langle\langle p, k\rangle\rangle$ ▷ *then, include $\langle p, k\rangle$ in the list l, and*
26: $X' = X' - \{\langle p, k\rangle\}$ ▷ *exclude $\langle p, k\rangle$ from X' indicating that it was already used, and*
27: $U = U_{aux}$ ▷ *update U, excluding the unstable items consumed by $\langle p, k\rangle$, and*
28: $U_R = U_R \cup \{\langle x, \langle p, k\rangle\rangle \mid x \in R_p - \mathcal{S}(R_p) - K_p\}$ ▷ *include in U_R all unstable items created by $\langle p, k\rangle$*
29: **end if**
30: **end for**
31: $U = U \cup U_R$ ▷ *include in set of current items all unstable items created by productions enabled in the current iteration*
32: **end while**

$GI(l)$ is the set of possible initial stable graphs for the productions in l, applied in the order in which they appear in l. It is defined as follows:

1: $\langle p, k\rangle = head(l)$ ▷ *get the first the first production in l*
2: $l = tail(l)$ ▷ *exclude the first production from l*
3: $G_0 = L_p \cup (R_p - \mathcal{S}(R_p))$ ▷ *initialise the partial initial graph with items consumed by p and unstable items created by p*
4: $C = \{G_0\}$ ▷ *initialise the set of initial graphs with items the first partial initial graph G_0*
5: **while** $l \neq \lambda$ **do** ▷ *repeat up to all productions in l to be considered*
6: **for all** $G \in C$ **do** $C_G = \varnothing$ ▷ *for each graph in C, initialise a set C_G, which will contain all graphs obtained applying in G the production on the top of l*
7: **end for**
8: **for all** $G \in C$ **do** ▷ *for each partial initial graph G in C*
9: $\langle p, k\rangle = head(l)$ ▷ *get the production on the top of l*
10: **for all** $q \in Pr(p)$ **do** ▷ *and for each production q, which include in the partial initial graph G all stable items consumed by $\langle p, k\rangle$ and parts of stable items preserved by $\langle p, k\rangle$ (the preserved items must be considered because they can be preserved only by $\langle p, k\rangle$ and must be included in G)*
11: **for all** match $m\colon L_q \to G$ **do** ▷ *and for each match of q in G*
12: $C_G = H \cup C_G$, where $G \overset{q,m}{\Rightarrow} H$ ▷ *include in C_G the graph obtained by applying q in G based on match m*
13: **end for**
14: **end for**
15: **end for**
16: $C_{aux} = \bigcup_{G \in C} C_G$ ▷ *get the union of partial initial graphs obtained from all G in C*
17: $C = C_{aux}$ ▷ *actualise C with all partial initial graphs obtained with base on production $\langle p, k\rangle$ on the top of l*
18: $l = tail(l)$ ▷ *exclude the production on the top of l*
19: **end while**

*and $Pr(p)$ contains, for each possible subgraph L of preserved stable part of production p, a production that consumes all unstable item consumed by p; preserves L plus all unstable items preserved by p; and creates all stable part of L_p, which is not contained in L, plus all unstable items created by p. It is defined by set X as follows, where p is a T-**Graph** production:*

1: $X = \varnothing$ ▷ *initialise the set of productions which create all stable items consumed by p and some stable items preserved by p*

2: ***for all*** *subgraph L of $\mathcal{S}(rng(l_p))$* ***do*** ▷ *for each subgraph L of L_p containing only stable preserved items*

3: $X = X \cup \{q\}$ ▷ *include in X a production that consumes the unstable items consumed by p and creates the stable items of L_p that is not in L and the unstable items created by p*

*where q is the T-**Graph** production $L_q \overset{l_q}{\leftarrow} K_q \overset{r_q}{\rightarrow} R_q$, with*

$L_q = \langle V_{L_q}, E_{L_q}, s^{L_q}, t^{L_q} \rangle$

$V_{L_q} = V_L \cup V \cup V_u$

$E_{L_q} = E_L \cup E_u$

$E_u = \{e \mid e \in E_{L_p} \wedge t_{L_p}(e) \notin T_s\}$ ▷ *set of unstable edges consumed or preserved by p*

$V = \{v \mid (v = s^{L_p}(e) \vee v = t^{L_p}(e)) \wedge e \in E_u\}$ ▷ *set of vertices that are source or target of edges in E_u*

$V_u = \{v \mid v \in V_{L_p} \wedge t_{L_p}(v) \notin T_s\}$ ▷ *set of unstable vertices consumed or preserved by p*

the functions s^{L_q} and t^{L_q} and the typing morphism are inherited from L_p

$K_q = \langle V_{K_q}, E_{K_q}, s^{K_q}, t^{K_q} \rangle$

$V_{K_q} = V_L \cup V \cup V'_u$

$E_{K_q} = E_L \cup E'_u$

$V'_u = \{v \mid v \in V_u \wedge v \in rng(l_p)\}$ ▷ *set of unstable vertices preserved by p*

$E'_u = \{e \mid e \in E_u \wedge e \in rng(l_p)\}$ ▷ *set of unstable edges preserved by p*

the functions s^{K_q} and t^{K_q} and the typing morphism are inherited from L_p

$R_q = \langle V_{R_q}, E_{R_q}, s^{R_q}, t^{R_q} \rangle$

$V_{R_q} = V_{\mathcal{S}(L_p)} \cup V'_u \cup V_s$

$E_{R_q} = E_{\mathcal{S}(L_p)} \cup E'_u \cup E_s$

$V_s = \{v \mid v \in V_{R_p} \wedge v \notin V_{\mathcal{S}(R_p)} \wedge v \notin rng(r_p)\}$ ▷ *set of unstable vertices created by p*

$E_s = \{e \mid e \in E_{R_p} \wedge e \notin E_{\mathcal{S}(R_p)} \wedge e \notin rng(r_p)\}$ ▷ *set of unstable edges created by p*

the functions s^{R_q} and t^{R_q} and the typing morphism are inherited from L_p and R_p

$\forall \in K_q \centerdot l_q(x) = id_{L_p}(x) \wedge r_q(x) = id_{L_p}(x)$ ▷ *map by l_q and r_q each preserved item to itself*

4: ***end for***

The function $Pr(p)$ always stops since the execution of its loop is realised for each subgraph of $\mathcal{S}(rng(l_p))$, that is a finite graph. The **while** loop (at lines $5 - 19$) of $GI(l)$ function is executed up to l becomes empty. Since one element of l is excluded in each iteration this loops eventually finishes. Therefore, since $Pr(p)$ and **while** loop finishes, the function $GI(l)$ always stops. The unique loop in $order(X')$ that is not controlled by a

finite structure is the **while** loop at lines $10-32$. This loop is repeated up to X' becomes empty. Each production $\langle p, k\rangle$ of X' is eliminated when the productions ordered in the previous iterations create all unstable items used by p (lines $24-29$). Since the productions in X' are balanced (i.e., all unstable items consumed/preserved by each production are created by others) all productions will be eliminated of X'. Therefore, the function $order(X')$ always finishes. Finally, since X and $GI(ord)$ are finite and $order(X')$ always finishes, the function $InitGraph(X)$ always stops.

Definition C.7 (transactions test) *Let $\phi : \mathcal{O} \to \mathcal{Z}$ be a graph process. $transaction(\phi)$ is* $\mathbf{true}$ *if ϕ is an transactional process and it is* false *otherwise. It is define by b as follows:*

1: $b = \mathbf{true}$ ▷ *initialise b indicating that ϕ is a transaction*
2: ***for all*** $x \in T_{\phi_s}$ ***do*** ▷ *for each stable item in the type graph of ϕ*
3: $pre_x = \{p \mid p \in P_\phi \wedge y \in L_p - dom(l_p) \wedge t_{L_p}(y) = x\}$ ▷ *assign all productions that consume x to pre_x*
4: $cont_x = \{p \mid p \in P_\phi \wedge y \in K_p \wedge t_{K_p}(y) = x\}$ ▷ *assign all productions that preserve x to $cont_x$*
5: $post_x = \{p \mid p \in P_\phi \wedge y \in R_p - rng(r_p) \wedge t_{R_p}(y) = x\}$ ▷ *assign all productions that create x to $post_x$*
6: ***if*** $pre_x \neq \varnothing$ ***then*** ▷ *if pre_x and*
7: ***if*** $cont_x \neq \varnothing$ ***then*** ▷ *$cont_x$ are not empty*
8: $b =$ false ▷ *then assign* false *to b indicating that ϕ is not a transaction*
9: ***else if*** $post_x \neq \varnothing$ ***then*** ▷ *if pre_x and $post_x$ are not empty*
10: $b =$ false ▷ *then assign* false *to b indicating that ϕ is not a transaction*
11: ***end if***
12: ***else if*** $cont_x \neq \varnothing$ ***then*** ▷ *if $cont_x$ and*
13: ***if*** $post_x \neq \varnothing$ ***then*** ▷ *$post_x$ are not empty*
14: $b =$ false ▷ *then assign* false *to b indicating that ϕ is not a transaction*
15: ***end if***
16: ***end if***
17: ***end for***
18: $Min = \varnothing$ ▷ *initialise the set of items of minimal graph of ϕ*
19: $Max = \varnothing$ ▷ *initialise the set of items of maximal graph of ϕ*
20: ***for all*** $x \in T_\phi$ ***do*** ▷ *for all item x in the type graph of ϕ*
21: ***if*** $\forall p \in P_\phi.\ \nexists y \in R_p - rng(r_p) \,.\, t_{R_p}(y) = x$ ***then*** ▷ *if there is no production that creates x*
22: $Min = Min \cup \{x\}$ ▷ *then include x in Min*
23: ***end if***
24: ***if*** $\forall p \in P_\phi.\ \nexists y \in L_p - dom(l_p) \,.\, t_{L_p}(y) = x$ ***then*** ▷ *if there is no production that consumes x*
25: $Max = Max \cup \{x\}$ ▷ *then include x in Max*
26: ***end if***
27: ***end for***
28: ***if*** $\exists x \in Min \wedge x \notin T_{\phi_s}$ ***then*** ▷ *if there is an unstable item in Min*
29: $b =$ false ▷ *then assign* false *to b indicating that ϕ is not a transaction*
30: ***end if***
31: ***if*** $\exists x \in Max \wedge x \notin T_{\phi_s}$ ***then*** ▷ *if there is an unstable item in Max*
32: $b =$ false ▷ *then assign* false *to b indicating that ϕ is not a transaction*
33: ***end if***

34: ***for all*** $x \in Min$ ***do*** ▷ *for all element x in Min*
35: ***if*** $\forall p \in P_\phi \,.\, (\nexists y \in L_p \,.\, t_{L_p}(y) = x)$ ***then*** ▷ *if there is no production preserving or consuming x*
36: $b = \textbf{false}$ ▷ *then assign* **false** *to b indicating that ϕ is not a transaction*
37: ***end if***
38: ***end for***
39: $Reachable = \varnothing$ ▷ *initialise the set containing sets with elements of each graph reachable from Min applying production of ϕ*
40: ***for all*** $P' \subseteq P_\phi$ ***do*** ▷ *for each subset P' of productions of ϕ*
41: ***if*** $\forall p \in P' \,.\, \forall x \in L_p .$
$((t_{L_p}(x) \in Min) \vee (\exists q \in P' \,.\, \exists y \in R_q - rng(r_p) \,.\, t_{R_q}(y) = t_{L_p}(x)))$ ***then***
▷ *if for all production in P', all consumed/preserved item are in Min or are created by another production in P'*
42: $S_{P'} = \{x \mid x \in T_\phi \wedge ((x \in Min \wedge \forall p \in P' .\ \nexists y \in L_p - dom(l_p) \,.\, t_{L_p}(y) = x) \vee$
$((\exists p \in P' \,.\, \exists y \in R_p - rng(r_p) \,.\, t_{R_p}(y) = x) \wedge$
$(\forall p' \in P' .\ \nexists z \in L_{p'} - dom(l_{p'}) \,.\, t_{L_{p'}}(z) = x)))\}$
▷ *then assign to $S_{P'}$ the set of all items in T_ϕ, such that these items are in Min and are not consumed by any production in P', or they are created by one production in P' and are not consumed by any other production in P'*
43: $Reachable = Reachable \cup \{S_{P'}\}$ ▷ *include the set $S_{P'}$ in $Reachable$*
44: ***end if***
45: ***end for***
46: ***for all*** $S \in Reachable$ ***do*** ▷ *for all set S in $Reachable$*
47: ***if*** $\forall x \in S \,.\, x \in T_{\phi_s}$ ***then*** ▷ *if all items in S are stable*
48: $b = \textbf{false}$ ▷ *then assign* **false** *to b indicating that ϕ is not a transaction*
49: ***end if***
50: ***end for***

All loops in function $transaction(\phi)$ are controlled by finite structures, then its computations always stop.

Definition C.8 (dep-weak-equivalence test) *Let ϕ_1 and ϕ_2 be two transactional process. $depEq(\phi_1, \phi_2)$ is* $\texttt{true}$ *if the processes are $dep-weak$-equivalent and it is* **false** *otherwise. $depEq(\phi_1, \phi_2)$ is defined by b as follows:*

1: $b = \textbf{false}$ ▷ *initialise b indicating that ϕ_1 and ϕ_2 are not dep-weak-equivalent*
2: ***if*** *there is an isomorphism $f: T_{\phi_1} \to T_{\phi_2}$ and there is a bijection $g: P_{\phi_1} \to P_{\phi_2}$ such that $rng(f_{|Min(\phi_1)}) = Min(\phi_2)$, $rng(f_{|Max(\phi_1)}) = Max(\phi_2)$ and $\phi_{2P} \circ g = \phi_{1P}$* ***then*** ▷ *if there is an isomorphism between type graphs such that minimal and maximal graphs are preserved and there is a bijection between productions such that two related productions must be mapped to the same production in the dT-GTS to which the processes are associated*
3: $R_{\phi_1} = \varnothing$ ▷ *initialise the relation resulting of translation of dependency relation of ϕ_1 into type of ϕ_2 w.r.t f*
4: ***for all*** $\langle a, b\rangle \in \prec_{\phi_1}$ ***do*** ▷ *for all pair in the dependency relation of ϕ_1*
5: $R_{\phi_1} = R_{\phi_1} \cup \{\langle f(a), f(b)\rangle\}$ ▷ *include in R_{ϕ_1} the pair of elements in type graph of ϕ_2 associated by the isomorphism f to a and b*
6: ***end for***

7: ***if*** $R_{\phi_1} = \prec_{\phi_2}$ ***then*** $b = \mathbf{true}$ ▷ *if the translation of* $\prec_{\phi_1}$ *is equal* $\prec_{\phi_2}$*, then assign* $\mathbf{true}$ *to* b *indicating that* ϕ_1 *and* ϕ_2 *are dep-weak-equivalent*

8: ***end if***

9: ***end if***

Since the loop in function defined above is controlled by a finite structure the computation of $depEq(\phi_1, \phi_2)$ *always stops.*

APPENDIX D RESUMO ESTENDIDO DA TESE

A complexidade dos sistemas atuais requer o uso de métodos de desenvolvimento que garantam correção e qualidade. Especificação formal é um importante instrumento usado para atingir estes objetivos. A especificação é uma descrição do comportamento de um sistema e/ou sua estrutura.

Sistemas de transformação de grafos (GTSs) é um formalismo adequado para a especificação de sistemas complexos, que podem levar em conta aspectos como orientação-à-objetos, concorrência, mobilidade e distribuição (EHRIG et al., 1999a). De fato, grafos podem ser naturalmente usados para fornecer uma representação estruturada dos estados de um sistema, a qual destaca seus subcomponentes e suas interconexões lógicas e físicas. Os eventos que ocorrem no sistema, que são responsáveis pela evolução de um estado para outro, são modelados por aplicações de regras de transformação adequadas, chamadas de produções (de grafos). Esta representação é suficientemente precisa para permitir análise formal do sistema em consideração, e também oferece uma representação intuitiva visual que pode ser facilmente compreendida por pessoal não-especialista.

Ao longo dos anos, o "framework" original foi sendo enriquecido, estendendo GTSs com conceitos de estruturação que são necessários para lidar com a complexidade de grandes especificações. Diversas noções de modularidade e refinamento foram propostas, proporcionando mecanismos básicos para encapsulamento, abstração e ocultação de informação – veja (HECKEL et al., 1999; SCHÜRR; WINTER, 2000; KREOWSKI; KUSKE, 1999; DREWES et al., 2000; EHRIG; ENGELS, 1996; HECKEL et al., 1998; TAENTZER; SCHÜRR, 1995; GROSSE-RHODE; PARISI-PRESICCE; SIMEONI, 1999; EHRIG; ENGELS, 1993). Contudo, pouca atenção tem sido dada à idéia de estender GTSs para permitir a especificação de atividades transacionais. Abstratamente, uma transação é uma atividade, envolvendo a execução de um grupo de eventos, os quais podem levar o sistema a um estado de sucesso ou falha. No último caso, a execução parcial da transação é descartada e não tem efeito no sistema. Em implementações concretas esta noção é obtida com mecanismos de "roll-back" que restaura o estado inicial quando a falha é detectada.

Em um trabalho introdutório (BALDAN et al., 2008), foi definida uma extensão de GTSs, chamadas *sistemas de transformação de grafos transacionais* (T-GTS), equipando-os com a noção de transação. Rudemente falando, estados (grafos) são divididos em partes estáveis e não-estáveis (instáveis), e uma transação é definida como uma computação que começa e termina em estados contendo somente ítens estáveis, na qual todos os estados intermediários tem alguma parte instável. Esta abordagem é motivado pela natureza "data-flow" deste formalismo, onde as produções do sistema são aplicadas não-deterministicamente, e qualquer forma de controle da aplicação das produções deve ser codificada nos grafos. Assim, transações são mais naturalmente definidas indiretamente,

através da identificação de partes do estado que representam recursos temporários (ou "instáveis"), visíveis somente dentro da transação. Transações podem ser vistas em dois diferentes níveis de abstração. Em um nível mais baixo, ambos os ítens estáveis e instáveis, e assim também a estrutura interna da transação, são visíveis. Mas em um nível mais abstrato, os ítens instáveis podem ser "esquecidos" e somente transações completas são observáveis. Intuitivamente, um novo GTS é obtido, onde transações do T-GTS original tornam-se produções que reescrevem diretamente o estado de origem no estado de destino.

Sistemas reativos, em contraste aos sistemas transformacionais, são caracterizados pela contínua reação a estímulos provenientes do seu ambiente. Se, além da reatividade, considerarmos que muitas aplicações o método de especificação deveria prover um modo de descrever a distribuição espacial dos estados, transformações de grafos parecem ser uma técnica de especificação adequada. Algumas aplicações com estas características são sistemas móveis e vias biológicas.

Diversos métodos para projeto e análise de sistemas reativos propõem linguagens síncronas como formalismo de especificação (BERRY, 2000; HALBWACHS et al., 1991; LEGUERNIC et al., 1991), onde o tempo de reação a um evento é nulo. Outros métodos (SECELEANU; SECELEANU, 2004; MAIA; IORIO; BIGONHA, 1998; RIESCO; TUYA, 2004) propõem usar linguagens assíncronas para especificar a comunicação entre os componentes e definem um mecanismo para descrever um conjunto (ou seqüência) de atividade que são realizadas atomicamente. Portanto, nos podemos usar a noção de transações para descrever, em um nível abstrato, estas atividades atômicas. Além disso, foi proposta, em (MAIA; IORIO; BIGONHA, 1998), uma abordagem para especificar explicitamente padrões de interação usando Máquinas de Estados Abstratas (ASMs), onde o projetista, além de especificar as operações do componente, pode descrever os sinais que são enviados e recebidos do ambiente. O projetista pode também especificar parcialmente o ambiente mostrando somente a especificação das interações.

Contudo, as abordagens para transformações de grafos não fornecem mecanismos para especificar explicitamente padrões de interação entre o sistema e seu ambiente. Algumas delas restringem o padrão de interação à funções (e assim na realidade descrevem sistemas transformacionais), e outras apenas permitem uma forma muito restrita de interações. Em (HECKEL, 1998), foi proposto o uso de sistemas de transformação de grafos para especificar sistemas reativos: a interação entre o sistema e seu ambiente não é explicitamente especificado, ao invés, ela é descrita em um nível semântico, onde os estados do sistema descrevem efeitos que não são determinados por aplicações de regras. Portanto, sistemas de transformação de grafos transacionais são estendidos para expressar interações explicitamente.

Nesta tese, é desenvolvido um trabalho mais elaborado em GTSs transacionais para permitir a abstração da estrutura interna das transações e demonstrar que um T-GTS e sua abstração têm o mesmo comportamento em termos de transações. Além disso, T-GTSs foram estendidos com um mecanismo para descrever padrões de interação entre um sistema e seu ambiente, o que permite especificar sistemas reativos. A idéia, nesta proposta, é que um componente interage com seu ambiente consumindo e criando elementos visíveis ao seu ambiente. Estas ações podem ser descritas como produções (abstratas) de grafos na especificação abstrata que são implementadas por uma série de outras produções em uma especificação mais concreta. Uma relação entre estes dois níveis de abstração foi definido, resultando em uma noção de refinamento.

Mais especificamente, os principais objetivos desta tese são:

- definir uma noção de atividade atômica para GTSs: alcançado através da definição da noção de transações;

- demonstrar que atividades atômicas são preservadas em um nível mais alto de abstração: atingido através da definição de morfismos de implementação e da demonstração da existência de uma adjunção entre as categorias dos T-GTS com morfismos de implementação e dos GTSs com morfismos padrão. Resultados preliminares deste trabalho foram publicados em (BALDAN et al., 2006);

- propor um mecanismo para descrever padrões de interação através de produções de grafos para especificar sistemas reativos: obtido através da definição de relações de dependência associadas às produções dos T-GTSs. Resultados preliminares deste trabalho foi publicado em (FOSS; MACHADO; RIBEIRO, 2007);

- definir uma noção de refinamento que leve em conta os padrões de interação: atingido através da definição de uma noção de refinamento baseada nos morfismos de implementação.

D.1 Contribuições

As contribuições desta tese estão relacionadas a duas diferentes áreas da Ciência da Computação: a área teórica, através da fundamentação teórica do conceito de transações para transformação de grafos; e a área de engenharia de software, propondo um formalismo para especificar sistemas reativos, onde as interações entre o sistema e seu ambiente podem ser descritas por relações associadas às produções. Além disso, usando morfismos de implementação, pode-se definir um método de desenvolvimento incremental, iniciando com uma visão abstrata do sistema e adicionando mais detalhes a cada passo de refinamento.

D.1.1 Contribuições para a área teórica

Nesta tese foi dada a fundamentação teórica da noção de atividades transacionais em transformação de grafos. Uma transação é definida como uma classe de derivações, onde os estados inicial e final são estáveis e todos os estados intermediários são instáveis. Assim, os ítens instáveis representam recursos temporários, visíveis somente dentro da transação. A distinção entre elementos estáveis e instáveis é forçada pelo mecanismo de tipagem. Esta definição de transações é inspirada no trabalho sobre *zero-safe nets* (BRUNI; MONTANARI, 2000) e é motivada pela natureza "data-driven" do formalismo de transformação de grafos, onde qualquer forma de controle na aplicação das produções é codificada nos grafos. Um dos principais resultados teóricos deste trabalho é a caracterização do sistema abstrato de um T-GTS (que contém todas as transações do T-GTS como produções) em termos de uma construção universal: um funtor adjunto à direita. Para obter este resultado, inicialmente, as transações foram caracterizadas como processos de grafos e após foi definida a noção de morfismo de implementação (outra importante contribuição teórica), permitindo associar produções a processos transacionais. Essa noção de morfismo de implementação é ainda mais geral que a noção definida em (BRUNI; MONTANARI, 2000), permitindo que produções instáveis possam ser implementadas por transações instáveis e, conseqüentemente, pode-se refinar também as implementações das produções.

Além dos resultados mencionados, a extensão de transformações de grafos transacionais, introduzida no Capítulo 4 (dT-GTS), enriquece a informação dada pelas produções de grafos das T-GTSs, tornando explícita a dependência entre elementos criados e consumidos/preservados. Esta relação pode ser usada para restringir o refinamento das transações, uma vez que a noção de morfismo de implementação para dT-GTSs deve respeitar as dependências. Assim, usando esta extensão podemos dar informações abstratas a respeito das dependências desejadas nas implementações.

D.1.2 Contribuições para a área de Engenharia de Software

A abordagem visual e dirigida a dados das GTSs, faz delas um formalismo natural para especificar sistemas reativos, onde o comportamento dos componentes é definido pelo fluxo dos sinais (dados) que são gerados pelo ambiente e não pelo fluxo de controle dos componentes. As noções de transação e relação de dependência introduzidas nesta tese tornam o formalismo ainda mais adequado para a especificação de sistemas reativos. Estas extensões incrementam o formalismo de GTSs com um mecanismo para especificar reações atômicas e, assim permite descrever, em um nível abstrato, sistemas síncronos. Nesta tese, também foi apresentada uma idéia inicial sobre estruturação vertical, propondo uma noção de refinamento para relacionar especificações abstratas e concretas, onde as dependências entre eventos/sinais de entrada e saída são dadas explicitamente como uma informação adicional a respeito do comportamento do sistema. Esta informação pode ser útil para especificar o comportamento do ambiente e para fins de verificação, como será discutida na seção 5.3.

Na seção a seguir, são apresentadas algumas idéias iniciais sobre como incrementar este framework.

D.1.3 Trabalhos futuros

As seguintes questões serão temas de trabalhos futuros:

- se este modelo for restringido a um tipo especial de GTSs chamado de Gramática de Grafos Baseadas em Objetos (OBGG), pode-se usar uma ambiente de desenvolvimento para especificação visual, simulação, verificação automática e geração de código (DOTTI et al., 2005, 2006). Em (DOTTI et al., 2006) foi proposta uma abordagem, para verificar sistemas parciais usando OBGGs, baseada na abordagem “assume-guarantee”. A idéia básica é ver cada parte do sistema como um sistema aberto - um sistema cujo comportamento não está completamente especificado e que depende das interações com seu ambiente. Para que seja possível usar esta abordagem, é necessário descrever ambas as interações de entrada e saída. As últimas são especificadas pela interface e são derivadas do comportamento do corpo do sistema. As interações de entrada (i.e., as reações provenientes do ambiente resultantes dos eventos/sinais do sistema) não podem ser derivadas da especificação do sistema, então é necessário incluir um mecanismo para descrever explicitamente este tipo de interação. Isto pode ser feito adicionando uma nova relação de dependência para cada produção na interface do módulo. Esta extensão permite restringir a semântica dos módulos, uma vez que tem mais dependências para considerar ao completar o comportamento do módulo.

- além disso, considerando uma dT-GTS e sua abstração relacionadas por um morfismo de implementação, como definido no Teorema 4.1, pode-se definir uma noção

de módulo, onde o sistema abstrato é a interface, descrevendo a interação entre o sistema e seu ambiente; e dT-GTS é o corpo que implementa a interface. Além disso, para definir noções de composição de módulos é necessário definir inicialmente uma noção de compatibilidade entre as interações, requerendo que os módulos compostos devem ter transações com padrões de interação duais. Além disso, para duas transações (de módulos diferentes) com compatibilidade de interações, deve existir elementos de sincronização, i.e., ítens estáveis que são criados por uma transação deve ser consumidos pela outra, antes do final da primeira. Em um módulo composto, estes elementos observáveis devem tornar-se inobserváveis pois as transações devem ser fundidas. É necessário ainda estudar melhor sobre a interface composta resultante, bem como, a semântica do módulo composto.

- outras noções de refinamento devem ser estudadas, por exemplos, considerando-se uma T-GTS sem relação de dependência como uma especificação abstrata e adicionar a relação de dependência no refinamento desta especificação. Outro tipo de refinamento que pode ser considerado é o refinamento da relação de dependência: em um nível abstrato ela pode ser menos restritiva e em um nível concreto ela pode ser mais restrita.

- O modelo das OBGGs é estendido, em (MICHELON; COSTA; RIBEIRO, 2006), com noções de tempo. Pode-se considerar aplicar esta extensão, para tentar expressar sincronia e noções de preempção no nível concreto.