Graph Transactions as Processes*

Paolo Baldan[1], Andrea Corradini[2], Luciana Foss[2,3,**], and Fabio Gadducci[2]

[1] Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy
[2] Dipartimento di Informatica, Università di Pisa, Italy
[3] Instituto de Informática, Universidade Federal do Rio Grande do Sul, Brasil

Abstract. *Transactional graph transformation systems* (T-GTSs) have been recently proposed as a mild extension of the standard DPO approach to graph transformation, equipping it with a suitable notion of *atomic* execution for computations. A typing mechanism induces a distinction between stable and unstable items, and a *transaction* is defined as a shift-equivalence class of computations such that the starting and ending states are stable and all the intermediate states are unstable.
The paper introduces an equivalent, 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. More specifically, we show that the functor mapping a T-GTS to a graph transformation system having as productions exactly the transactions of the original T-GTS is the right adjoint to an inclusion functor.

Keywords: Graph processes, refinement, transactions, zero-safe nets.

1 Introduction

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 [9, 10]. 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 as the application of suitable transformation rules. 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, e.g., [11, 14, 13]).

* Supported by the CNPq-CNR IQ-Mobile II, the EC RTN 2-2001-00346 SegraVis, the EU IST-2004-16004 SEnSOria and the MIUR PRIN 2005015824 ART.
** Supported by CAPES and CNPq.

In a *top-down* approach to the specification of a complex system, one can start describing each operation of the system as a single "abstract" rule. Then, each abstract rule is refined to a computation, describing in a more concrete way the activity performed and possibly the use of temporary resources. In order to guarantee that the behaviour of the refined system is correct with respect to the abstract specification, each computation corresponding to an abstract rule has to be executed "atomically", i.e., either it completes successfully, or the effects of a partial execution should not be visible at the abstract level: in one word, the computation refining an abstract rule must be a *transaction*.

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. Ideally, the following *ACID properties* should be guaranteed for each transaction

- *Atomicity*: either all of the tasks of a transaction are performed (and the transaction is *committed*) or none of them are;
- *Consistency*: the database is in a legal state when the transaction begins and when it ends;
- *Isolation*: no operation outside the transaction can see the data in an intermediate state;
- *Durability*: the effects of a committed transaction are persistent.

The above properties are also meaningful for characterising transactions in specification formalisms of concurrent/distributed systems, where the interaction now occurs with the environment: *atomicity*, *consistency* and *isolation* carry on with equal relevance, while only *durability* does not have a clear meaning anymore since no persistent repository of data is modelled.

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*, like rewriting formalisms and (possibly High-Level) 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 taken for *zero-safe nets* [4], which is a reference model for our work on transactional GTSs.

Zero-safe nets are Place/Transition Petri nets equipped with a distinguished subset of *zero places*. The places model resources that are consumed or produced by transitions and the zero places model resources that are invisible to the exterior of a step. A step in a zero-safe net starts at a *stable marking* (i.e., containing no zero places), evolves through *unstable markings* and ends in a stable marking. Stable tokens produced in a step are "frozen" and delivered at the end.

Inspired by the work on zero-safe nets, *transactional graph transformation systems* (T-GTSs), introduced in [1], are a mild 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 from 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.

In this paper we elaborate further on transactional GTSs. At first we obtain an alternative characterisation of transactions as *graph processes*, by exploiting the results in [2]. Next we show how the internal structure of transactions can be abstracted away, by considering an *abstract* GTS associated to the T-GTS: unstable items disappear and each distinct transaction becomes a single atomic production, which rewrites the starting stable state to the final stable state.

The main result of the paper shows that the operation mapping each T-GTS to its abstract counterpart is characterised as a universal construction in the categorical setting. More specifically, such construction is turned into a functor between the corresponding categories of systems, which is right adjoint to the inclusion functor in the opposite direction. The result is obtained by equipping T-GTSs with a notion of *implementation morphism*, allowing to map a single production to a whole transaction. This provides a solid theoretical justification to the notion of abstract GTS associated to a T-GTS: according to an intuitive interpretation of categorical adjunctions, it states that the constructed abstract GTS is the best approximation of the given T-GTS in the class of ordinary GTSs.

2 Double-Pushout Rewriting

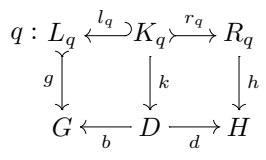
This section briefly summarises the basics of double-pushout (DPO) graph rewriting [8] for directed (multi-)graphs (but definitions and results of the paper generalise easily, for example, to hypergraphs, which are used indeed in the examples). Without loss of generality, as shown in [12], we consider rewriting with *injective matches* only. Graphs are equipped with a *typing morphism* to a fixed *type graph*, which plays an essential role when distinguishing between stable and unstable items in a given graph.

Formally, a *graph* is a tuple $\langle V, E, s, t\rangle$, where V and E are the (disjoint) sets of nodes and edges, and $s, t\colon E \to V$ are the source and target functions. Sometimes, abusing the notation, G denotes the disjoint union $V_G \uplus E_G$; e.g. writing $x \in G$ means that x is either a node or an edge of the graph G. Given a graph T, a *typed graph* G over T is a graph $|G|$, together with a graph morphism $t_G\colon |G| \to T$. A *morphism* between T-typed graphs $f\colon G_1 \to G_2$ is a graph morphism $f\colon |G_1| \to |G_2|$ respecting the typing, i.e., such that $t_{G_1} = t_{G_2} \circ f$. The category of T-typed graphs and typed graph morphisms is denoted by T-**Graph**.

Rewriting rules, called *T-typed productions*, are tuples $q\colon L_q \overset{l_q}{\leftarrow} K_q \overset{r_q}{\rightarrow} R_q$, where q is the name of the production, L_q, K_q and R_q are T-typed graphs (called the *left-hand side*, the *interface* and the *right-hand side* of the production, respectively), and l_q, r_q are injective morphisms. Without loss of generality, we always assume that l_q is an inclusion.

A rule q specifies that an occurrence of the left-hand side L_q in a larger graph can be rewritten into the right-hand side R_q, preserving the interface K_q.

Formally, given a typed graph G, a production q, and an injective *match* $g\colon L_q \to G$, a *direct derivation* δ *from G to H using q, g* exists, written $\delta\colon G \overset{q,g}{\Longrightarrow} H$, if the diagram to the right can be constructed, where both squares are pushouts in T-**Graph**.



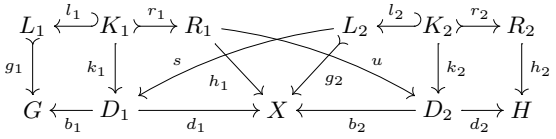
A graph transformation system is then defined as a collection of rules, over a fixed graph of types.

Definition 1 (graph transformation system). *A T-typed* graph transformation system *(GTS) is a tuple $\mathcal{G} = \langle T, P, \pi \rangle$, where T is a graph, P is a set of production names and π is a function mapping production names in P to T-typed productions.*

A *derivation* in a GTS $\mathcal{G}$ is a sequence of direct derivations via productions of $\mathcal{G}$

$$G_0 \overset{q_0,g_0}{\Longrightarrow} G_1 \overset{q_1,g_1}{\Longrightarrow} \ldots\ldots \overset{q_n,g_n}{\Longrightarrow} G_{n+1}.$$

A two-steps derivation $G \overset{q_1,g_1}{\Longrightarrow} X \overset{q_2,g_2}{\Longrightarrow} H$ as in the diagram below is called *sequential independent* [8, 12] if there are two morphisms $s\colon L_2 \to D_1$ and $u\colon R_1 \to D_2$ such that $d_1 \circ s = g_2$ and $b_2 \circ u = h_1$. Intuitively, the images in X of the left-hand side of q_2 and of the right-hand side of q_1 overlap only on items that are preserved by both derivation steps.

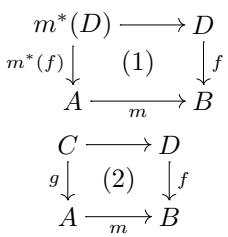


In this case, according to the Parallelism Theorem (Theorem 7.8 in [12]), we can apply to G a suitably defined *proper quotient* q of the parallel rule $q_1 + q_2$, obtaining an equivalent direct derivation from G to H via an injective match g. Furthermore, there is an equivalent derivation $G \overset{q_2,g_2'}{\Longrightarrow} X' \overset{q_1,g_1'}{\Longrightarrow} H$ where the two derivation steps are "switched". The equivalence on derivations induced by switchings of sequential independent direct derivations is called *shift-equivalence* [8].

We now equip GTSs with a suitable notion of morphism, allowing us to look at them as objects of a category. This is essential to provide a characterisation of some interesting constructions with universal properties, as shown in Section 5. We shall use a variant of the morphisms in [6, 3], where the type graphs are related by a partial morphism rather than by an arbitrary span.

A *partial morphism* $f\colon G_1 \rightharpoonup G_2$ is a total morphism from a subgraph of G_1, called $dom(f)$, to G_2, and is equivalently depicted as $G_1 \overset{l_f}{\hookleftarrow} dom(f) \overset{r_f}{\to} G_2$. 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 obiects; an arrow $h\colon f \to g$ in $\mathcal{C}{\downarrow}A$ is a $\mathcal{C}$-arrow h such that $g \circ h = f$.[4]

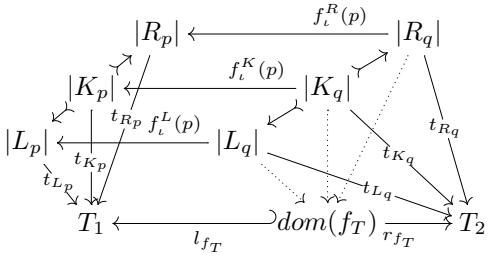
Let $m\colon A \to B$ be an arrow in a category $\mathcal{C}$ with pullbacks. Chosen a pullback square as (1) to the right for any $f\colon D \to B$, the *pullback functor* along $m\colon A \to B$, denoted $m^*\colon \mathcal{C}{\downarrow}B \to \mathcal{C}{\downarrow}A$, maps an object $(f\colon D \to B) \in \mathcal{C}{\downarrow}B$ 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 $g \cong m^*(f)$ if there exists an arrow $C \to D$ such that square (2) to the right is a pullback.



Definition 2 (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 GTSs. A* GTS morphism *$f\colon \mathcal{G}_1 \to \mathcal{G}_2$ is a pair $f = \langle f_T, f_P \rangle$, where*

- *$f_T\colon T_1 \rightharpoonup T_2$ is a partial graph morphism;*
- *$f_P\colon P_1 \to P_2 \cup \{\emptyset\}$ is a total function on production names, where $\emptyset\colon (\varnothing \leftarrow \varnothing \to \varnothing)$ is the empty production;*

such that productions are preserved, i.e., for all $p \in P_1$, with $f_P(p) = q$, there are morphisms $f_\iota^L(p)$, $f_\iota^K(p)$ and $f_\iota^R(p)$ such that the diagram to the right commutes, and $f_\iota^X(p) \cong t^*_{X_p}(l_{f_T})$ for $X \in \{L, K, R\}$.
The category with GTSs as objects and the corresponding morphisms as arrows is denoted by **GTS**.



Chosen a pullback functor $l^*_{f_T}$, the partial morphism $f_T\colon T_1 \rightharpoonup T_2$ induces a *retyping functor* $f_T^{\leftrightarrow}\colon T_1\text{-}\mathbf{Graph} \to T_2\text{-}\mathbf{Graph}$, defined on objects as $f_T^{\leftrightarrow}(t_G\colon |G| \to T_1) = r_{f_T} \circ l^*_{f_T}(t_G)$. 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 (see e.g. [6, 3]), meaning that, for a derivation ρ in $\mathcal{G}_1$, (any choice of) the retyped diagram $f_T^{\leftrightarrow}(\rho)$ is a derivation in $\mathcal{G}_2$.

3 Transactional Graph Transformation Systems

In this section we first recall the basics of *transactional* GTSs [1]. Next we introduce the notion of morphism between such systems and we show that morphisms preserve transactions. The basic idea underlying transactional GTSs consists in distinguishing between stable and unstable resources and defining transactions as "minimal" computations which start and end in stable states. The distinction between stable and unstable items in a graph is induced by specifying a subgraph of the type graph, which is intended to represent the stable types.

[4] Thus, for example, $T\text{-}\mathbf{Graph} = \mathbf{Graph}{\downarrow}T$.

Definition 3 (transactional GTS). *A* transactional GTS *(*T-GTS*) is a pair* $\mathcal{Z} = \langle \mathcal{G}, T_s \rangle$*, where* $\mathcal{G}$ *is a* T*-typed* GTS *(the* underlying GTS *of* $\mathcal{Z}$*) and* $i_s : T_s \hookrightarrow T$ *is a subgraph of the type graph of* $\mathcal{G}$*, called the* stable type graph.

We denote by $\mathcal{S} : T\text{-}\mathbf{Graph} \to T_s\text{-}\mathbf{Graph}$ 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 i_s^*.

The stabilising functor can be applied point-wise to any production of a given T-GTS, thus producing a GTS typed over the stable type graph.

Definition 4 (stabilised GTS). *Given a* 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* $q \in P$*.*

By construction, there is an obvious GTS morphism from a T-GTS $\mathcal{Z}$ to its stabilised GTS $\mathcal{S}(\mathcal{Z})$, given by the pair $\langle id_{T_s} : T \rightharpoonup T_s, id_P \rangle$. Since GTS morphisms are simulations, the following result trivially holds.

Proposition 1. *Let* $\mathcal{Z}$ *be a* T-GTS *and let* $\rho = G_0 \overset{q_1, g_1}{\Longrightarrow} G_1 \overset{q_2, g_2}{\Longrightarrow} \ldots \overset{q_n, g_n}{\Longrightarrow} G_n$ *be a derivation in* $\mathcal{G}$*. Then* $\mathcal{S}(\rho)$*, defined as below, is a derivation in* $\mathcal{S}(\mathcal{Z})$*.*

$$\mathcal{S}(\rho) = \mathcal{S}(G_0) \overset{q_1, \mathcal{S}(g_1)}{\Longrightarrow} \mathcal{S}(G_1) \overset{q_2, \mathcal{S}(g_2)}{\Longrightarrow} \ldots \overset{q_n, \mathcal{S}(g_n)}{\Longrightarrow} \mathcal{S}(G_n)$$

Let us come to the definition of transaction in a T-GTS. Inspired by the approach for Petri nets proposed in [4], and extended to nets with read arcs in [5], we introduce *stable steps*, *transactions* and *abstract transactions*. In the following, let $\mathcal{Z} = \langle \mathcal{G}, T_s \rangle$ be an arbitrary but fixed T-GTS.

Let us first define a graph G as *stable* if it consists only of stable items, i.e., if $|\mathcal{S}(G)| = |G|$, and *unstable* otherwise. A stable step is, intuitively, a computation which starts and ends in stable states. Moreover, stable items which are generated are "frozen", in the sense that they can not be preserved nor consumed by other productions inside the same step; similarly, stable items which are deleted cannot be preserved by other productions. Therefore, 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.

Definition 5 (stable step and transaction). *A* stable step *is a derivation* $\rho = G_0 \overset{q_1, g_1}{\Longrightarrow} G_1 \overset{q_2, g_2}{\Longrightarrow} \ldots \overset{q_n, g_n}{\Longrightarrow} G_n$ *which enjoys 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 rule* $q_1 + \ldots + q_n$ *and a suitable match* g*.*

A transaction *is a stable step additionally satisfying*

3. *the match* g *is an isomorphism;*
4. *no intermediate graph* G_i $(i \neq 0, n)$ *is stable.*

By condition 3, the start graph contains exactly what the transaction needs to reach a successful end. Notice that this condition defines what is a transaction, but then, in a computation, a transaction can be embedded into a larger context. By condition 4 no sub-derivation of ρ is a transaction.

Actually, since we are considering a concurrent model of computations, 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 shift-equivalent derivation.

When combining shift-equivalence with an equivalence which abstracts also with respect to the concrete identities of items in the involved graphs, i.e., which considers graphs up to isomorphism, we obtain the so-called *abstract truly-concurrent equivalence* [8]. The equivalence class of a derivation ρ with respect to such equivalence will be denoted by $[\rho]_a$ and called an *abstract trace*.

We are now able to introduce the notion of *abstract* transaction.

Definition 6 (abstract transaction). *An* abstract transaction *is an abstract trace $[\rho]_a$ such that any derivation $\rho' \in [\rho]_a$ is a transaction.*

A simple transactional GTS, presented in [1], tests the equality between integer expressions involving natural numbers, represented as sequences $\mathsf{S}(\mathsf{S}(\ldots\mathsf{S}(0)\ldots))$, and a sum operator. Figure 1 shows some of the productions, whose numbering refers to the original system. The type graph and its stable subgraph, not depicted here, can be inferred from the labeling observing that dashed items (hyper-edges depicted as boxed and nodes depicted as circles) are not stable.

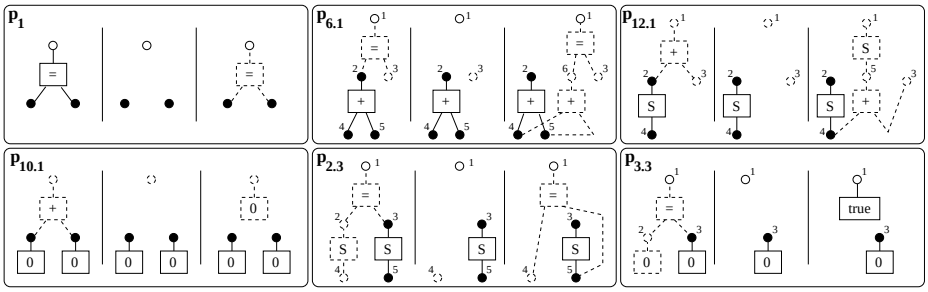


Fig. 1. Some productions of the T-GTS testing equality of natural numbers

Figure 2 shows the sequence of graphs of a derivation starting from the stable graph representing the expression $\mathsf{S}(0) + 0 = \mathsf{S}(0)$, and using the productions of Figure 1 in the given order. Intuitively, the derivation starts by making unstable the top operator $=$, and then triggering the evaluation of the sum operator. The evaluation of $+$ does not modify the stable part of the graph: it builds the result using unstable items, which are then consumed by the evaluation of the equality operator. The last graph is stable, and it includes the result of the evalutation on the node to which the original equality operator was attached.

It is not difficult to check that this derivation is a transaction, as it satisfies all conditions of Definition 5; furthermore its equivalence class is an abstract transaction, since all shift-equivalent derivations are transactions as well.

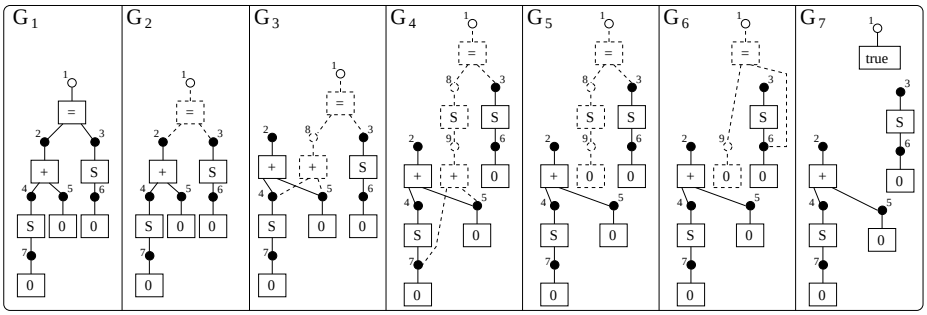


Fig. 2. A sample derivation evaluating $\mathsf{S}(\mathsf{0}) + \mathsf{0} = \mathsf{S}(\mathsf{0})$

We now extend the definition of GTS morphisms to transactional GTSs, explaining how morphisms behave with respect to the stable/unstable items.

Definition 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*

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 $f_T(z) \in T_{2s}$.*

The category having T-GTS*s as objects and the corresponding morphisms as arrows is denoted by* **TGTS**.

Note that we require that the type graph component of a morphism preserves both stable and unstable items. Additionally it must be total on unstable items.

In order to ensure that morphisms are simulations in this more general framework, we prove that T-GTS morphisms preserve abstract transactions.

Proposition 2 (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 $[f_T^{\leftrightarrow}(\rho)]_a$ (see the note after Definition 2) is an abstract transaction in $\mathcal{Z}_2$.*

4 Transactions as Processes

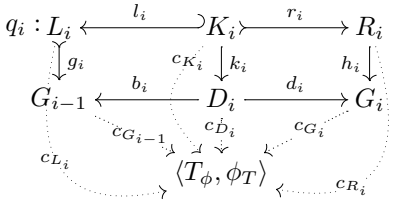
Inspired by the classical *non-sequential processes* for Petri nets, *graph processes* have been proposed in [7, 2] as a faithful representation of the derivations of a GTS up to shift-equivalence. A graph process for a T-GTS $\mathcal{Z}$ is defined as an "occurrence grammar" $\mathcal{O}$, i.e., a grammar satisfying suitable acyclicity constraints, equipped with a T-GTS morphism from $\mathcal{O}$ to $\mathcal{Z}$.

The derivations in $\mathcal{O}$ are mapped through the morphism to derivations in $\mathcal{Z}$, which are shown to be shift-equivalent. Vice versa, from each derivation in $\mathcal{Z}$ a process can be obtained by a simple colimit construction, and shift-equivalent derivations yield isomorphic processes. Since abstract transactions are defined as abstract traces, the corresponding processes provide a compact, handier representation for them, that will be exploited in the next section for the definition of *implementation morphisms* among T-GTSs.

In the present paper a process for a T-GTS $\mathcal{Z}$ is defined by an explicit colimit construction for any derivation in $\mathcal{Z}$. A more abstract characterisation based on structural properties can be provided as well, as in [2], but it is not needed here.

Definition 8 (process from a derivation). *Let $\mathcal{Z} = \langle\langle T, P, \pi\rangle, T_s\rangle$ be a* T-GTS*, and let $\rho = G_0 \overset{q_1,m_1}{\Longrightarrow} G_1 \overset{q_2,m_2}{\Longrightarrow} \dots \overset{q_n,m_n}{\Longrightarrow} G_n$ 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*

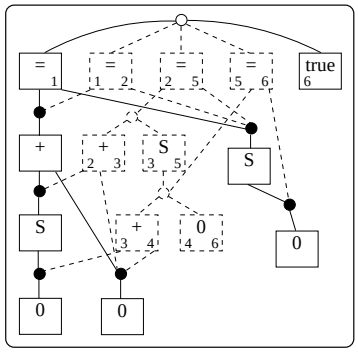
- *$\langle T_\phi, \phi_T\rangle$ is a colimit object (in T-***Graph***) of the diagram representing derivation ρ, as depicted (for a single derivation step) in the diagram below, where $c_{X_i} \colon X_i \to T_\phi$ is the induced injection for $X \in \{D, G, L, K, R\}$;*
- *$T_{\phi_s} \hookrightarrow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



Intuitively, the colimit construction applied to a derivation constructs the graph T_ϕ as a copy of the source graph plus the items created during the rewriting.

As an example, we show the type graph of the process associated to the derivation of Figure 2.

The injections from the graphs of the derivation are implicitly represented by indexing some edges with a *creation index* in the bottom-left corner, and a *deletion index* in the bottom-right one. The creation index is missing in the edges 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 $i \in \{1, \dots, 7\}$, contains all edges with creation index, if any, smaller than i, and deletion index, if any, larger than or equal to i.



Two processes ϕ and ϕ' for a T-GTS $\mathcal{Z}$ are *isomorphic* if there exists a T-GTS isomorphism $f \colon \mathcal{O}_\phi \to \mathcal{O}_{\phi'}$ such that $\phi' \circ f = \phi$. An *abstract process* for $\mathcal{Z}$ is an isomorphism class of processes for $\mathcal{Z}$ and it is denoted $[\phi]$ where ϕ is a representative in the class.

Since in a derivation all matches are assumed to be injective, it can be shown that in the associated process all rules are injectively typed in T_ϕ: referring to the diagram after Definition 8, all the morphisms c_{X_i} to T_ϕ are injective for $X \in \{G, D, L, K, R\}$. If $x \in T_\phi$ 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. $Min(\mathcal{O}_\phi)$ denotes the subgraph of T_ϕ consisting of the items x such that $^\bullet x = \emptyset$, and $^\bullet \phi$ the same graph typed over T by the restriction of ϕ_T. The graphs $Max(\mathcal{O}_\phi)$ and $\phi^\bullet$ are defined by duality.

Definition 9 (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 $((q_1{}^\bullet \cap \underline{q_2}) \cup (\underline{q_1} \cap {}^\bullet q_2)) \neq \emptyset$.*

It is easy to show that the causal relation is indeed a partial order.

Definition 10 (reachable set). *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*

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

We now introduce *transactional processes*, i.e., processes representing abstract transactions. For technical reasons we consider also a wider class of processes, the *unstable transactional processes*, which may start and end in unstable states.

Definition 11 (transactional process). *Let $\mathcal{Z} = \langle \langle T, P, \pi \rangle, T_s \rangle$ be a* T-GTS. *An* unstable transactional 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 unstable t-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 so.

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 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 after Definition 8 is transactional.

From the theory of graph processes (see [2]) 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 $TP_{\mathcal{Z}}$ 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 $\phi^{\leftrightarrow}_T(\rho)$ and $\phi^{\leftrightarrow}_T(\rho')$ of $\mathcal{Z}$ (see the observation after Definition 2) are abstract truly-concurrent equivalent, and thus belong to the same abstract trace. This defines a function $PT_{\mathcal{Z}}$ mapping the abstract processes for $\mathcal{Z}$ to abstract traces of $\mathcal{Z}$. Moreover, it can be proved that functions $TP_{\mathcal{Z}}$ and $PT_{\mathcal{Z}}$ are inverse to each other. By the next proposition they establish an isomorphism between abstract transactions and abstract t-processes: hence, these latter provide an alternative, equivalent characterisation of the former ones.

Proposition 3. *Let $\mathcal{Z}$ be a* T-GTS. *Then $[\phi]$ is an abstract t-process of $\mathcal{Z}$ iff $PT_{\mathcal{Z}}([\phi])$ is an abstract transaction.*

5 The Abstract System of a Transactional 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 GTS, where both stable and unstable states, and thus also the internal structure of transactions, are visible. But we can abstract away from the unstable states and observe only complete transactions. 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. This transformation defines a mapping from the objects of the category **TGTS** to those of **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.

We start by introducing the abstract GTS associated to a given T-GTS, where productions are abstract processes of the original T-GTS corresponding to transactions. For technical reasons, it is convenient to define productions as equivalence classes of t-processes which, roughly speaking, are isomorphic when forgetting the stable preserved part. We first define the span induced by a process.

Definition 12 (span underlying a process). *Given a process ϕ for a* T-GTS *$\mathcal{Z}$, the* underlying span *of ϕ is $\Pi(\phi) = {}^\bullet\phi \hookleftarrow {}^\bullet\phi \cap \phi^\bullet \hookrightarrow \phi^\bullet$ (intersection is taken component-wise).*

Given an ut-process ϕ, with $\mathcal{O}_\phi = \langle\langle T_\phi, P_\phi, \pi_\phi\rangle, T_{\phi_s}\rangle$, consider the structure $r(\phi)$, typed over the set of items $T_\phi - \mathit{Min}(\mathcal{O}_\phi) \cap \mathit{Max}(\mathcal{O}_\phi) \cap T_{\phi_s}$, where any component is restricted to such set of types (intuitively, the stable preserved part is forgotten). Then, two ut-processes ϕ_1 and ϕ_2 are *read-equivalent*, written $\phi_1 \simeq_r \phi_2$, if $\mathit{\Pi}(\phi_1) \simeq \mathit{\Pi}(\phi_2)$, i.e., they have the same associated span, and $r(\phi_1) \simeq r(\phi_2)$. A *read ut-process (rut-process)* is defined as an equivalence class of ut-processes with respect to read-equivalence, denoted as $[\phi]_r$ for a representative ϕ. The set of rut-processes of a T-GTS $\mathcal{Z}$ is denoted by $\mathbf{rutProc}(\mathcal{Z})$. The set of *read t-processes (rt-processes)* $\mathbf{rtProc}(\mathcal{Z})$ is defined in an analogous way.

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 13 (span underlying abstract process). *Let us assume for each* T-GTS $\mathcal{Z}$ *a choice function* $\mathsf{ch}_\mathcal{Z}$*, mapping each rut-process* $[\phi]_r$ *to a concrete representative* $\mathsf{ch}_\mathcal{Z}([\phi]_r) \in [\phi]_r$*. The* underlying span *of a rut-process* $[\phi]_r$ *is defined as* $\mathit{\Pi}_\mathcal{Z}([\phi]_r) = \mathit{\Pi}(\mathsf{ch}_\mathcal{Z}([\phi]_r))$*.*

We are now able to define the abstract system associated with a GTS.

Definition 14 (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 $\langle T_s, \mathbf{rtProc}(\mathcal{Z}), \mathit{\Pi}_\mathcal{Z}\rangle$ *where* $\mathbf{rtProc}(\mathcal{Z})$ *is the set of rt-processes of* $\mathcal{Z}$ *and* $\mathit{\Pi}_\mathcal{Z}$ *is as in Definition 13.*

For instance, in the abstract GTS of the T-GTS recalled in Section 3 the rt-process having the type graph shown after Definition 8 is a production. The corresponding span has graphs G_1 and G_7 as left- and right-hand side, respectively.

An implementation morphism is a T-GTS morphism that maps each given production of the source system to a read unstable *transactional* process of the target system, and also provides a triple of morphisms mapping the production underlying the process to the given production: this additional information is needed to compose implementation morphisms correctly.

Definition 15 (T-GTS **implementation morphisms).** *For a given* T-GTS $\mathcal{Z} = \langle\langle T, P, \pi\rangle, T_s\rangle$*, let* $\widehat{\mathcal{Z}} = \langle\langle T, \mathbf{rutProc}(\mathcal{Z}), \mathit{\Pi}_\mathcal{Z}\rangle, T_s\rangle$ *be the* T-GTS *having all read ut-processes as productions. An* implementation pre-morphism $f\colon \mathcal{Z}_1 \to \mathcal{Z}_2$ *is a triple* $f = \langle f_T, f_P, f_\iota\rangle$*, where* $\langle f_T, f_P\rangle\colon \mathcal{Z}_1 \to \widehat{\mathcal{Z}_2}$ *is a* T-GTS *morphism and* f_ι *is a family* $f_\iota = \{f_\iota(p) \mid p \in P_{\mathcal{Z}_1}\}$ *such that for each* $p \in P_{\mathcal{Z}_1}$*,* $f_\iota(p) = \langle f_\iota^L(p), f_\iota^K(p), f_\iota^R(p)\rangle$ *is a given choice of the three arrows whose existence is required in Definition 2.*

Given two pre-morphisms $\langle f_T, f_P, f_\iota\rangle, \langle f_T, f_P, g_\iota\rangle\colon \mathcal{Z}_1 \to \mathcal{Z}_2$*, let* $p \in P_1$ *such that* $f_P(p) = [\phi]_r$*. Then we write* $g_\iota(p) \approx f_\iota(p)$ *if there are a process automorphism* $\alpha\colon \mathsf{ch}_\mathcal{Z}([\phi]_r) \to \mathsf{ch}_\mathcal{Z}([\phi]_r)$ *and a span automorphism* $\eta : \mathit{\Pi}_\mathcal{Z}([\phi]_r) \to \mathit{\Pi}_\mathcal{Z}([\phi]_r)$ *which restricts to the identity over unstable items, such that* $g_\iota(p) = f_\iota(p) \circ \alpha_\mathit{\Pi} \circ \eta$*, component-wise (*$\alpha_\mathit{\Pi}$ *stands for the restriction of* α_T *to* $\mathit{\Pi}_\mathcal{Z}([\phi]_r)$*).*

An implementation morphism *is an equivalence class of pre-morphisms, where* $\langle f_T, f_P, f_\iota\rangle \approx \langle f_T, f_P, g_\iota\rangle$ *if* $g_\iota(p) \approx f_\iota(p)$ *for all* $p \in P_{\mathcal{Z}_1}$*.*

Roughly, implementation morphisms are classes of pre-morphisms up to the equivalence induced on the third component by process isomorphisms (note that the type component of an automorphism $\alpha : \mathsf{ch}_{\mathcal{Z}}([\phi]_r) \to \mathsf{ch}_{\mathcal{Z}}([\phi]_r)$ restricts to an automorphism over the span $\Pi_{\mathcal{Z}}([\phi]_r)$). The third component is further quotiented along isomorphisms of the stable subgraph: this is safe because, by the definition of transaction, stable items are not used in composing computations.

In order to provide a correct definition of the category having T-GTSs as objects and implementation morphisms as arrows, we first have to explain how implementation morphisms compose. This is summarised by the next lemma. 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 8) to the one-step derivation which applies p to its left-hand side L_p with the identity match.

Lemma 1 (composition and identity for implementation morphisms). *Given a* T-GTS $\mathcal{Z}$*, let* $\widehat{\mathcal{Z}}$ *be as in Definition 15. Then, the properties below hold.*

1. *Any* T-GTS *morphism* $f\colon \mathcal{Z}_1 \to \widehat{\mathcal{Z}_2}$ *extends to a* T-GTS *morphism* $\widehat{f}\colon \widehat{\mathcal{Z}_1} \to \widehat{\mathcal{Z}_2}$.
2. *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 the* T-GTS *morphism* $\widehat{g} \circ f\colon \mathcal{Z}_1 \to \widehat{\mathcal{Z}_3}$*. Then composition is associative.*
3. *For each* T-GTS $\mathcal{Z}$*, let* $id_{\mathcal{Z}} = \langle id_{\mathcal{Z}T}, id_{\mathcal{Z}P}, id_{\mathcal{Z}\iota}\rangle\colon \mathcal{Z} \to \widehat{\mathcal{Z}}$ *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* $[\phi_{id_p}]_r$*;*
   - *for each production* p*,* $id_{\mathcal{Z}\iota}(p)$ *is a triple of isomorphisms mapping the span* $\Pi_{\mathcal{Z}}([\phi_{id_p}]_r)$ *to* $L_p \hookleftarrow K_p \hookrightarrow R_p$ *and making the two resulting squares commute.*

   Then $id_{\mathcal{Z}}$ *is well-defined (any choice of* $id_{\mathcal{Z}\iota}$ *determines the same implementation morphism) and it is the identity on* $\mathcal{Z}$*.*

The proof of the lemma is long and involuted, and we give only some hints. Most interesting is the proof of point *1*. Let $f\colon \mathcal{Z}_1 \to \widehat{\mathcal{Z}_2}$ be a T-GTS morphism and ϕ a process for $\mathcal{Z}_1$. Thus ϕ has a set of productions mapped injectively into its type graph T_ϕ. Any such production p is mapped by f_P to a process of $\mathcal{Z}_2$, equipped with morphisms from its minimal and maximal graphs to the left- and right-hand sides of p (given by the component f_ι). Then the process $\widehat{f}_P(\phi)$ is obtained by "gluing" (with a colimit construction) all the processes which are images of productions in ϕ along the intersections in T_ϕ determined by the f_ι component.

The lemma allows to introduce a category with implementation morphims.

Definition 16 (category $\mathbf{TGTS}^{imp}$). *We denote by* $\mathbf{TGTS}^{imp}$ *the category having transactional* GTS*s as objects and implementation morphisms as arrows.*

Additionally, exploiting point *1* in Lemma 1 we can show that the extension $\widehat{f}\colon \widehat{\mathcal{Z}_1} \to \widehat{\mathcal{Z}_2}$ maps *stable* processes to stable processes, i.e., rt-processes of $\mathcal{Z}_1$ are mapped to rt-processes of $\mathcal{Z}_2$. This in turn can be used to prove that the abstraction function for T-GTS can be seen as a functor.

Proposition 4 (abstraction functor). *Function A, mapping a* T-GTS *to its abstract* GTS*, can be extended to a functor* $\mathcal{A}\colon \mathbf{TGTS}^{imp} \to \mathbf{GTS}$.

Quite obviously, a GTS $\mathcal{G} = \langle T, P, \pi \rangle$ can be seen as a T-GTS $\mathcal{I}(\mathcal{G}) = \langle \langle T, P, \pi \rangle, T \rangle$. This mapping can be extended to an inclusion functor $\mathcal{I}\colon \mathbf{GTS} \to \mathbf{TGTS}^{imp}$ in the following way: if $f = \langle f_T, f_P \rangle \colon \mathcal{G}_1 \to \mathcal{G}_2$ is a GTS morphism, then the T-GTS morphism $\mathcal{I}(f) = \langle g_T, g_P, g_\iota \rangle \colon \mathcal{I}(\mathcal{G}_1) \to \widehat{\mathcal{I}(\mathcal{G}_2)}$ is given as

- $g_T = f_T$;
- for each production $p \in P_{\mathcal{G}_1}$ its image $g_P(p)$ is the rt-process $[\phi_{f_P(p)}]_r$, where, as above, $\phi_{f_P(p)}$ is the process associated to the one-step derivation obtained by applying $f_P(p)$ to its left-hand side;
- for each production $p \in P_{\mathcal{G}_1}$, $g_\iota(p)$ is a triple of isomorphisms mapping the span $\Pi_{\mathcal{I}(\mathcal{G}_2)}([\phi_{f_P(p)}]_r)$ to $L_p \hookleftarrow K_p \hookrightarrow R_p$ and making the two resulting squares commute.

We are now ready to present the main result of the paper.

Theorem 1 (universality of abstraction). *The abstraction functor* $\mathcal{A} : \mathbf{TGTS}^{imp} \to \mathbf{GTS}$ *is right adjoint to the inclusion functor* $\mathcal{I}$.

Proof (Sketch). For each T-GTS $\mathcal{Z}$, we define the component at $\mathcal{Z}$ of the counit $\epsilon_{\mathcal{Z}} \colon \mathcal{I}(\mathcal{A}(\mathcal{Z})) \to \mathcal{Z}$. This is an implementation morphism, thus a T-GTS morphism $\epsilon_{\mathcal{Z}} \colon \mathcal{I}(\mathcal{A}(\mathcal{Z})) \to \widehat{\mathcal{Z}}$. Its type graph component is simply the inclusion of the stable type graph into the full type graph, while the component on productions maps each abstract rt-process of $\mathcal{Z}$ to itself. It remains to show that given a GTS $\mathcal{G}$ and a T-GTS $\mathcal{Z}$, for each implementation morphism $f \colon \mathcal{I}(\mathcal{G}) \to \mathcal{Z}$, there is a unique $h \colon \mathcal{G} \to \mathcal{A}(\mathcal{Z})$ such that $\epsilon_{\mathcal{Z}} \circ \mathcal{I}(h) = f$.

Now, observe that morphism f maps each production of $\mathcal{G}$ to a rt-process of $\mathcal{Z}$. Since productions in $\mathcal{A}(\mathcal{G})$ are exactly the rt-processes of $\mathcal{G}$, the morphism $h \colon \mathcal{G} \to \mathcal{A}(\mathcal{G})$ can be defined identically. The proofs of uniqueness and of the fact that $\epsilon_{\mathcal{Z}} \circ \mathcal{I}(h) = f$ are long, but routine. □

6 Conclusions

The present paper carried on the investigation on transactional graph transformation systems, introduced in [1], as a tool for expressing transactional activities in graph transformation. A transaction is defined as a shift-equivalence class of derivations such that the starting and ending states are stable and all the intermediate states are unstable. Thus unstable items are intended to represent temporary resources, only visible within a transaction, and the distinction between stable and unstable items is enforced by a typing mechanism.

The "indirect" definition of transactions based on the dichotomy between stable and unstable items, inspired by the work on zero-safe nets [4], is motivated by our understanding of graph transformation as a data-centered formalism, where the rules of a system are applied non-deterministically, and any form of control on the application of rules has to be encoded in the graphs.

As far as the realm of graph transformation is concerned, though, also more traditional notions of transaction have been considered, most importantly in the design of PROGRES [16]. 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. The PROGRES approach is therefore similiar to the way transactions are introduced in programming languages and other control-centered formalism, and as a consequence a direct comparison with our approach is not feasible.

Besides reviewing the basic definitions concerning graph transactions, enriching and streamlining the original proposal, the main 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. A key concept introduced in the present paper is that of implementation morphisms among T-GTSs, allowing to map productions to transactional processes. Such morphisms are similar to the refinement morphisms of [11], where productions can be mapped to arbitrary derivations: a deeper analysis of the relationships among the two notions will be a topic of future work.

As mentioned in the introduction, the ACID properties are a canonical way of characterising transactions, even if in our framework only the first three are relevant. Let us discuss informally how such properties are guaranteed by the notion of transaction presented in this paper. *Atomicity* is guaranteed by the fact that computations that do not represent a transaction are forgotten at the abstract level. *Consistency* is guaranteed because the initial and final states of a transaction are all stable, and at the abstract level only stable graphs are considered. *Isolation* is guaranteed by the fact that a transaction, besides starting and ending in stable states, is "minimal", in the sense that all derivations that are shift-equivalent to it are also transactions. Hence, intermediate unstable states are only accessible inside the transaction itself. This implies that, if two transactions can be applied in parallel to a stable graph, then all the direct derivations of either of them are independent of the direct derivations of the other one. Thus, as desired, the transactions can be interleaved in an arbitrary way.

Currently we are working on the definition of a notion of *graph transformation module*, based on the theory presented in this paper. The idea is that a T-GTS can be seen as the implementation of a module, and its abstract GTS as the exported interface. Module composition mechanisms defined as suitable colimits are under investigation, as well as the study of the precise relationship between the new notion of module and existing ones in the literature (as in [11, 14, 13]).

Acknowledgements. We are mostly indebted to Roberto Bruni and Leila Ribeiro for enlightening discussions about the topic of the paper, as well as to an anonymous referee for pointing out an inconsistency in the submitted version.

References

1. P. Baldan, A. Corradini, F.L. Dotti, L. Foss, F. Gadducci, and L. Ribeiro. Towards a notion of transaction in graph rewriting. In R. Bruni and D. Varró, editors, *Proceedings International Workshop on Graph Transformation and Visual Modeling Techniques*, Electr. Notes in Theor. Comp. Sci. Elsevier, 2006. To appear.
2. P. Baldan, A. Corradini, and U. Montanari. Concatenable graph processes: relating processes and derivation traces. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *Proceedings International Conference on Automata, Languages and Programming*, volume 1443 of *Lect. Notes in Comp. Sci.* Springer, 1998.
3. P. Baldan, A. Corradini, and U. Montanari. Unfolding of double-pushout graph grammars is a coreflection. In H. Ehrig, G. Engels, H.J. Kreowski, and G. Rozenberg, editors, *Proceedings International Workshop on Theory and Application of Graph Transformations*, volume 1764 of *Lect. Notes in Comp. Sci.*, pages 145–163. Springer, 1999.
4. R. Bruni and U. Montanari. Zero-safe nets: Comparing the collective and individual token approaches. *Info. & Comp.*, 156(1-2):46–89, 2000.
5. R. Bruni and U. Montanari. Transactions and zero-safe nets. In H. Ehrig, G. Juhás, J. Padberg, and G. Rozenberg, editors, *Advances in Petri Nets: Unifying Petri Nets*, volume 2128 of *Lect. Notes in Comp. Sci.*, pages 380–426. Springer, 2001.
6. A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and J. Padberg. The category of typed graph grammars and its adjunctions with categories of derivations. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Proceedings International Workshop on Graph Grammars and their Application to Computer Science*, volume 1073 of *Lect. Notes in Comp. Sci.* Springer, 1996.
7. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3/4):241–265, 1996.
8. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation I: Basic concepts and double pushout approach. In Rozenberg [15], chapter 3, pages 163–245.
9. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools*. World Scientific, 1999.
10. H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 3: Concurrency, Parallelism, and Distribution*. World Scientific, 1999.
11. M. Große-Rhode, F. Parisi-Presicce, and M. Simeoni. Formal software specification with refinements and modules of typed graph transformation systems. *Journal of Computer and System Science*, 64(2):171–218, 2002.
12. A. Habel, J. Müller, and D. Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.
13. R. Heckel, H. Ehrig, G. Engels, and G Täntzer. Classification and comparison of module concepts for graph transformation systems. In Ehrig et al. [9], chapter 17, pages 669–689.
14. H.-J. Kreowski and S. Kuske. Graph transformation units and modules. In Ehrig et al. [9], chapter 15, pages 607–638.
15. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*. World Scientific, 1997.
16. A. Schürr, A. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In Ehrig et al. [9], chapter 13, pages 487–550.