Towards a Notion of Transaction in Graph Rewriting[1]

P. Baldan[a], A. Corradini[b], F.L. Dotti[c], L. Foss[b,d,2]
F. Gadducci[b], L. Ribeiro[d]

[a] *Università Ca' Foscari di Venezia, Italy*
[b] *Università di Pisa, Italy*
[c] *Pontifícia Universidade Católica do Rio Grande do Sul, Brasil*
[d] *Universidade Federal do Rio Grande do Sul, Brasil*

Abstract

We define *transactional graph transformation systems* (T-GTSs), a mild extension of the ordinary framework for the double-pushout approach to graph transformation, which allows to model transactional activities. Generalising the work on *zero-safe nets*, the new graphical formalism is based on a typing discipline which induces a distinction between stable and unstable items. A transaction is then a suitably defined minimal computation which starts and ends in stable states. After providing the basics of T-GTSs, we illustrate the expected results, needed to bring the theory to full maturity, and some possible future developments.

Keywords: Graph transformations, zero-safe nets, transactions.

1 Introduction

Graphs and graph transformations represent the core of most visual languages [2]. 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, can be modelled as the application of graph transformation rules. Such a representation is not only precise enough to allow the formal analysis of the system under scrutiny, but it is also amenable of an intuitive, visual representation, which can be easily understood also by a non-expert audience.

A *graph transformation system* (GTS) consists of a set of rewriting rules, also called graph productions [14]. In their basic formulation, GTSs do not provide

[1] Research partially supported by the CNPq-CNR Project IQ-Mobile II, the EC RTN 2-2001-00346 SegraVis, the EU IST-2004-16004 SENSORIA and the MIUR Project ART.
[2] Supported by CAPES and CNPq.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

mechanisms for synchronising or structuring computations, even if, since the left-hand side of productions can be arbitrarily large, a kind of synchronisation among items in the state (graph items) can be expressed.

Along the years several enrichments of the basic framework have been proposed, extending GTSs with mechanisms for expressing synchronisation between productions as well as for tackling modularity and refinement issues, which are features deemed necessary for a high-level specification formalism.

Instead, 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 this paper we face, from a foundational perspective, the problem of equipping graph transformation with mechanisms for modelling transactions. More precisely, we propose a mild extension to the double-pushout (DPO) approach to graph transformation, introducing *transactional graph transformation systems* (T-GTSs), a framework which provides a simple way of expressing transactional activities. Our formalism is deeply influenced and generalists the *zero-safe nets* proposal, introduced in [3] to solve an analogous modelling problem in the setting of Petri nets.

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. The "valid" computations of a T-GTS may start from a completely stable graph, evolve through graphs with unstable items and eventually end up in a new stable state; and the valid computations which are minimal, in a certain sense to be made precise in the paper, represent *transactions*.

The paper introduces the T-GTSs formalism, provides the basic concepts and illustrates a simple case study. In a concluding section we outline how the internal structure of transactions can be abstracted away by considering the so-called *abstract* GTS associated to the T-GTS, where unstable items disappear and each distinct transaction becomes a single atomic production, which rewrites the starting stable state to the final stable state. Thus "unfinished" transactions have no counterpart at the abstract level. Finally, we outline future venues of research, pointing out the technical issues which need to be further elaborated upon, such as the precise functorial correspondence between a T-GTS and its abstract counterpart.

2 Typed Graph Transformation Systems

In this section we introduce the basics of the double-pushout (DPO) algebraic approach to graph rewriting [9]. We remark that, although our approach will be developed for DPO rewriting over directed (multi-)graphs, it could have been easily adapted to other approaches to graph rewriting, e.g., to the single-pushout approach and to different notions of graph (e.g., to hypergraphs, which are used indeed in the

example in Section 4).

An essential ingredient of our theory is a typing discipline for graphs which will allow us to distinguish between stable and unstable items in a given graph. Typing for graphs (e.g., [5]) 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 labelling function is required to be a graph morphism.

Formally, a *graph* is a tuple $\langle V, E, s, t\rangle$, where V and E are sets of nodes and edges, and $s, t : E \to V$ are the source and target functions. Given a graph T, a *typed graph* G over T is a graph $|G|$, together with a total graph morphism $t_G : |G| \to T$. A *morphism* between T-typed graphs $f : G_1 \to G_2$ is a graph morphism $f : |G_1| \to |G_2|$ consistent with 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 of the kind $q = L_q \stackrel{l_q}{\leftarrow} K_q \stackrel{r_q}{\rightarrow} R_q$, where 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. A rule intuitively specifies that an occurrence of the left-hand side L_q in a larger graph can be rewritten into the right-hand side graph R_q, preserving the interface K_q. Formally, given a typed graph G, a production q, and a *match* $g : L_q \to G$, a *direct derivation δ from G to H using q, g* exists, written $\delta : G \stackrel{q,g}{\Longrightarrow} H$, if the diagram

$$\begin{array}{ccccc} q: L_q & \xleftarrow{l_q} & K_q & \xrightarrow{r_q} & R_q \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle k} & & \downarrow{\scriptstyle h} \\ G & \xleftarrow[b]{} & D & \xrightarrow[d]{} & H \end{array}$$

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 2.1 [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 DPO productions.

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

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

3 Transactional Graph Transformation Systems

In this section we introduce the basics of transactional graph transformation systems. After discussing the typing discipline which allows to distinguish between stable and unstable items in a given typed graph, we show how this can be used to define a notion of transaction.

The distinction between stable and unstable items is induced by specifying a subgraph of the type graph, which is intended to represent the stable types.

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*.

Given a graph G typed over T we can single out its stable part $\mathcal{S}(G)$, i.e., the subgraph consisting of its stably-typed items only. Formally, $\mathcal{S}(G)$ can be defined as the graph typed over T_s obtained by considering the pullback

$$\begin{array}{ccc} |\mathcal{S}(G)| & \overset{\iota}{\hookrightarrow} & |G| \\ \downarrow & & \downarrow \\ T_s & \hookrightarrow & T \end{array}$$

Without loss of generality, we will assume a concrete choice for $\mathcal{S}(G)$, by imposing that the morphism ι in the pullback diagram above is an inclusion.

We say that a graph is stable if it consists only of stable items. This is formalised in the next definition.

Definition 3.2 [stable graph] A T-typed graph G is called *stable* if $|\mathcal{S}(G)| = |G|$ (i.e., if the morphism ι in the pullback diagram is the identity). It is called *unstable* otherwise.

It can be shown that the above transformation is functorial: given a morphism of T-typed graphs $f : G \to H$, the transformation above uniquely induces a morphism $\mathcal{S}(f) : \mathcal{S}(G) \to \mathcal{S}(H)$ (which is, given the concrete choice for $\mathcal{S}(G)$, the restriction of f to $\mathcal{S}(G)$). The corresponding functor $\mathcal{S} : T\text{-}\mathbf{Graph} \to T_s\text{-}\mathbf{Graph}$ is called *stabilising functor*.

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 3.3 [stabilised GTS] Given a T-typed T-GTS $\langle \mathcal{G}, T_s \rangle$, the *stabilised* GTS $\mathcal{S}(\mathcal{G})$ 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 $\langle \mathcal{G}, T_s \rangle$, produces a derivation in $\mathcal{S}(\mathcal{G})$. An indirect proof of this fact can be obtained by observing that there exists a typed GTS morphism $f : \mathcal{G} \to \mathcal{S}(\mathcal{G})$, in the sense of [1], which essentially forgets about the non-stable items. Then, using the fact that GTS morphisms are simulations, one can infer the result below.

Proposition 3.4 *Let $\langle \mathcal{G}, T_s \rangle$ be a* T-GTS *and let $d = G_0 \overset{q_1,g_1}{\Longrightarrow} G_1 \overset{q_2,g_2}{\Longrightarrow} \dots \overset{q_n,g_n}{\Longrightarrow} G_n$ be a derivation in $\mathcal{G}$. Then*

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

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

Let us come to the definition of a transaction in a T-GTS $\langle \mathcal{G}, T_s \rangle$. Inspired by the approach for Petri nets proposed in [3] and extended to nets with read arcs in [4], we introduce *stable steps*, *stable transactions* and *abstract stable transactions*. In the following $\langle \mathcal{G}, T_s \rangle$ is a fixed T-GTS.

A stable step is, intuitively, a computation which starts and ends in stable states. Moreover, once generated, stable items are "frozen", in the sense that they

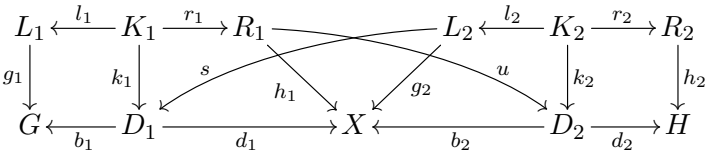


Fig. 1. Sequential independent derivations.

cannot be read or consumed by other productions inside the same step. 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 will be applicable in parallel.

To give a formal definition we need to briefly review some notions. A derivation $G \overset{q_1,g_1}{\Longrightarrow} X \overset{q_2,g_2}{\Longrightarrow} H$ as in Figure 1 is called *sequential independent* [6] if there are two morphisms $s : L_2 \to D_1$ and $u : R_1 \to D_2$ 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 we can apply the two productions in the reverse order, obtaining derivation $G \overset{q_2,g_2'}{\Longrightarrow} X' \overset{q_1,g_1'}{\Longrightarrow} H$ and we can also apply them concurrently, obtaining a parallel direct derivation $G \overset{q_1+q_2,g}{\Longrightarrow} H$.

Definition 3.5 [stable step] A *stable step* is a derivation $d = G_0 \overset{q_1,g_1}{\Longrightarrow} G_1 \overset{q_2,g_2}{\Longrightarrow} \ldots \overset{q_n,g_n}{\Longrightarrow} G_n$ which enjoys the following properties:

(i) G_0 and G_n are stable graphs;

(ii) the derivation $\mathcal{S}(d)$ is equivalent to a parallel direct derivation $\mathcal{S}(G_0) \overset{q_0+\ldots+q_n,\mathcal{S}(g)}{\Longrightarrow} \mathcal{S}(G_n)$ in $\mathcal{S}(\mathcal{G})$.

Definition 3.6 [stable transaction] A *stable transaction* is a stable step $d = G_0 \overset{q_1,g_1}{\Longrightarrow} G_1 \overset{q_2,g_2}{\Longrightarrow} \ldots \overset{q_n,g_n}{\Longrightarrow} G_n$ such that, if $\mathcal{S}(G_0) \overset{q_0+\ldots+q_n,\mathcal{S}(g)}{\Longrightarrow} \mathcal{S}(G_n)$ in $\mathcal{S}(\mathcal{G})$ is the induced parallel derivation, then

(i) g is an epimorphism;

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

By condition (i), the start graph contains exactly what the transaction needs to be brought to a successful end, while by condition (ii) no sub-derivation of d is a transaction, thus guaranteeing atomicity.

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., any *shift-equivalent* (see, e.g., [13,6]) 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* [6]. The equivalence class of a derivation d with respect to such equivalence will be denoted by $[d]_c$ and called *abstract trace*.

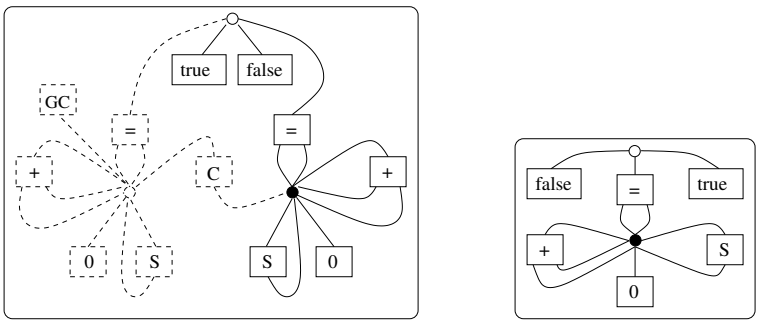


Fig. 2. The type graph (left) and its stable component (right).

Definition 3.7 [abstract stable transaction] An *abstract stable transaction* is an abstract trace $[d]_c$, such that for any $d' \in [d]_c$ the derivation d' is a stable transaction.

It follows from the definition that if two abstract stable 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.

Clearly, a more manageable characterisation of abstract stable transactions would be desirable: even if the corresponding theory is not yet completely developed, we will sketch in the concluding section how such a characterisation could be obtained by means of suitable graph processes.

4 A simple example on integer equality

We now present a simple GTS for testing the equality between integer expressions involving natural numbers represented as sequences $\mathsf{S}(\mathsf{S}(\ldots\mathsf{S}(\mathsf{0})\ldots))$ and a sum operator. Despite its small size we hope that this example will pinpoint the key features of our approach.

The type (hyper-)graph and its stable subgraph are depicted in Figure 2. Explicitly stated, the dashed items (dashed boxes representing (hyper-)edges and dashed circles for nodes) are not stable. Notice that, as usual for hypergraphs, each edge is connected to an ordered list of nodes. The order is implicit in our drawings: the first connection leaves the edge from the top, and the others follow counter-clockwise.

As a sample expression to be evaluated we consider $\mathsf{S}(\mathsf{S}(\mathsf{0})) + \mathsf{S}(\mathsf{0}) = \mathsf{S}(\mathsf{0})$, as represented by the stable graph G_0 on the left of Figure 5. For the sake of simplicity, G_0 is a tree, but the system also works for acyclic graphs, where subexpressions can be shared. In order to ensure that a shared subexpression is not affected by the evaluation of an outer expression, some rules duplicate the part of the structure that needs to be accessed in a destructive manner.

Let us consider the production p_1 in Figure 3: the graph on the left (center, right) represents the left-hand side (interface and right-hand side, respectively) of the production. Note that, according to the shape of the type graph, an unstable operator can be connected to a stable node only through an additional unstable node and a C-labelled edge. In order to simplify the presentation, such node and the C-labelled edge will be omitted in the figures. For instance, production p_1

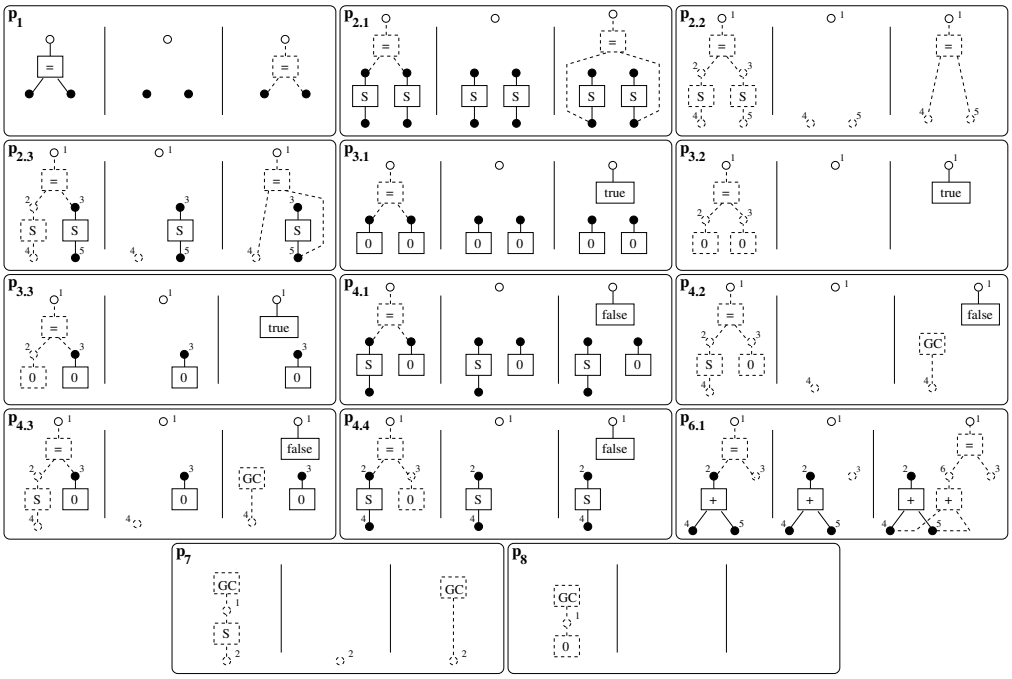
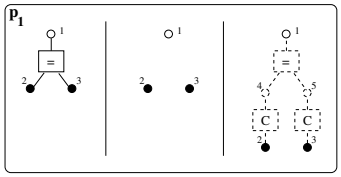


Fig. 3. Productions for the equality operator and for garbage collection.

should be read as



Intuitively, a computation proceeds as follows. The only production that can be applied to a stable graph like G_0 is p_1, which starts a transaction by replacing the stable edge $\boxed{=}$ with its unstable, dashed counterpart $\boxed{=}$. Next, conceptually, the equality operator traverses the expression (see production $p_{2.1}$), triggering, whenever it is needed, the evaluation of the sum operators by generating an unstable copy of them (production $p_{6.1}$). In turn, the evaluation of the sum generates as its result a chain of unstable successor operators (see productions $p_{12.1}$ and $p_{11.1}$ in Figure 4), recursively triggering the evaluation of nested additions (as in production $p_{9.1}$), and stopping when both arguments are zero (as in production $p_{10.1}$). The equality operator can then proceed, consuming the chain of unstable successors generated by the sum, till when either one or two zeros are reached. At this point the boolean result is generated (as in productions $p_{3.2}$), and, if needed, the "garbage collection" of the remaining unstable items is started (productions $p_{4.2}$, p_7 and p_8).

The presence of stable and unstable versions of both operators and constants motivates the existence of several variants for each production. For example, all the productions $p_{2.1}$, $p_{2.2}$ and $p_{2.3}$ (as well as its symmetric version $p_{2.4}$ which is not depicted) basically replace, conceptually, the subexpression $\mathsf{S}(\mathsf{x}) = \mathsf{S}(\mathsf{y})$ by the equivalent one $\mathsf{x} = \mathsf{y}$. Such productions do not have the same structure, though, because stable S-edges have to be preserved, as they may belong to a shared subex-

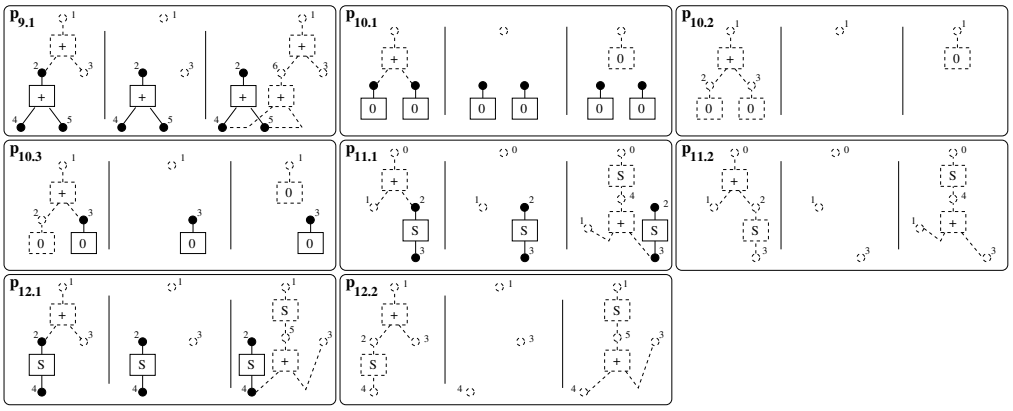


Fig. 4. Productions for the sum operator.

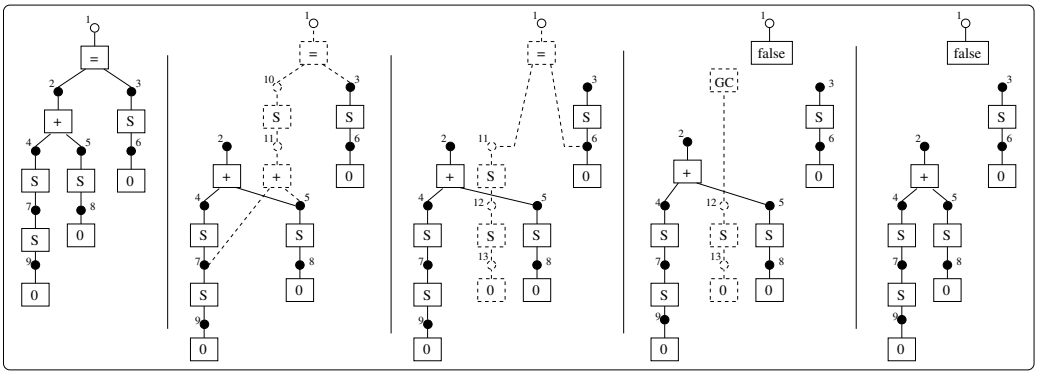


Fig. 5. An expression (left), some unstable states (center), and the result (right).

pression, while unstable S-edges have to be deleted, as they should not appear in the final state: this can be done safely, because unstable S-edges are generated by the productions in a way that guarantees that they are never shared.

The same observation applies to each other group of productions, like $p_{3\_}$ (modelling the rule $0 = 0 \rightsquigarrow \mathsf{true}$), $p_{4\_}$ (modelling $\mathsf{S}(x) = 0 \rightsquigarrow \mathsf{false}$), and so on. Notice that several rules have a symmetric version (exchanging the left and right arguments of the main binary operator) which are not depicted. For example the productions in the missing $p_{5\cdot\_}$ family model the evaluation of $0 = \mathsf{S}(x)$ to false. They are obtained from the $p_{4\cdot\_}$ productions by exchanging the arguments of the equality operator.

Some states of the derivation starting from $\mathsf{S}(\mathsf{S}(0)) + \mathsf{S}(0) = \mathsf{S}(0)$ and reaching the final state, which represents the result false, are depicted in Figure 5. From the starting state productions p_1, $p_{6.1}$ and $p_{12.1}$ are applied, reaching the second state; next, applying productions $p_{12.1}$, $p_{11.1}$, $p_{10.1}$ and $p_{2.3}$ the third state is reached; then, applying $p_{4.3}$ the fourth state is reached; and finally the application of p_7 and p_8 produces the final state. Note that all the intermediate states are unstable, due to the presence of at least one unstable item. Hence, the only visible states in the derivation, which can be shown to be a stable transaction, are the initial and final ones.

The corresponding abstract stable transaction includes all the derivations which are obtained by switching sequential independent direct derivations, such as the one

which applies the productions in the order p_1, $p_{6.1}$, $p_{12.1}$, $p_{2.3}$, $p_{12.1}$, $p_{4.3}$, $p_{11.1}$, $p_{10.1}$, p_7 and p_8. It can be shown that each abstract stable transaction in the proposed system performs the evaluation of exactly one equality operation, building as an unstable intermediate structure the result of the sum operators, and destroying them at the end.

5 Future perspectives

This paper introduces *transactional graph transformation systems*, a formalism which is intended to enrich the classical DPO approach to graph rewriting with a built-in notion of transaction. Our work so far outlined the basic notions underlying the framework, and further results are now needed to bring the theory to full maturity.

Abstract GTS associated to a transactional GTS
A first line of research concerns the definition of the abstract GTS associated to a T-GTS. As discussed in the paper, 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 states, and thus also the internal structure of transactions, are visible. But we can also abstract away from the unstable states and observe only complete transactions. Formally, this gives rise to another GTS, whose definition requires the notion of the production *induced* by a derivation sequence, a known construction in the literature. The production induced by a derivation $d : G_0 \Rightarrow^* G_n$ 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 5.1 [Abstract GTS] Let $\langle \mathcal{G}, T_s \rangle$ be a T-GTS. Given an abstract stable transactions $[d]_c$, a production induced by d is called *abstract production* for the transaction $[d]_c$.

The *abstract* GTS associated to the given T-GTS, denoted by $\mathcal{A}(\langle \mathcal{G}, T_s \rangle)$, is the GTS $\langle T_s, P', \pi' \rangle$ where P' is the set of abstract stable transactions $[d]_c$ and $\pi'([d]_c)$ is an abstract production for the transaction $[d]_c$.

As an example, the abstract production that corresponds to the transaction depicted in Figure 5 is shown in Figure 6.

As it should be evident from the proposed example, the abstract GTS associated to a T-GTS can have, in general, an infinite number of productions. Indeed, the notion of transaction allows one to model an abstract system with infinitely many productions by means of a lower level system, with a finite number of productions.

From a theoretical point of view the definition of the abstract GTS associated to a T-GTS might not be yet fully satisfactory, since it lacks an extensional presentation, as it is offered by categorical means in terms of adjunctions.

However, note that any GTS $\mathcal{G}$ can be naturally seen as a T-GTS $\langle \mathcal{G}, T \rangle$ by considering the entire type graph T as stable. Hence, turning the classes of GTSs and of T-GTSs into categories **GTS** and **TGTS**, respectively, there would be an obvious inclusion functor of **GTS** into **TGTS**. Thus, a solid justification for the construc-

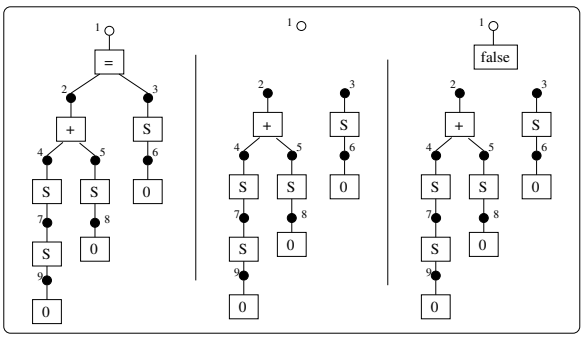


Fig. 6. The abstract production induced by the transaction of Figure 5.

tion of the abstract GTS associated to a T-GTS could come from a characterisation of the mapping $\mathcal{A}$ (Def. 5.1) as a functor from the category of T-GTSs to the category of GTSs, and from its characterisation as the right adjoint to the inclusion functor in the opposite direction. Intuitively, this would mean that, given a T-GTS $\langle \mathcal{G}, T_s \rangle$, the abstract GTS $\mathcal{A}(\langle \mathcal{G}, T_s \rangle)$, given in Definition 5.1, is the "best approximation" of $\langle \mathcal{G}, T_s \rangle$ in the category **GTS**.

We foresee two possible ways of proving a result of this kind:

(i) Freely generated category of systems with transactions as productions. Inspired by the work on zero-safe Petri nets [3], the idea consists of freely generating complex computations of a given T-GTS, starting and ending in stable states, by suitably composing its original productions. The considered composition operation should act differently on the stable and unstable items, composing the former in parallel and the latter sequentially. Moreover, it should be subject to axioms which identify computations differing only for the order of independent steps. In this setting transactions would be identified as computations that cannot be decomposed as the parallel composition of (non trivial) computations.

(ii) Transactions as special processes. Graph processes [5] are structures which provide a truly concurrent representation of a deterministic computation in a given GTS, by explicitly representing the start and ending state, as well as all the intermediate items produced in the computation and their causal dependencies. A transaction can be characterised as a process which starts and ends in stable states, where only direct causal dependencies between stable items exist, and which satisfies suitable atomicity properties.

In both cases, it seems that the appropriate choice of morphisms in the category of T-GTS should be that of implementation or refinement morphisms [10,11], which allow to map a single production into a computation.

Multi-level transactional GTSs

Another issue to be addressed concerns the "binary" distinction between stable and unstable items, which can be unsatisfactory in certain situations. In fact, a system can be viewed at several levels of abstractions, and what appears to be as an atomic production can be refined to a computation at a lower level and can be the building block of more complex transactions at a higher-level. In the proposed framework, the situation could be recast by replacing the stable/unstable dichotomy by a multi-

layered structure, consisting of a set of graphs T_0, T_1, ..., T_n such that T_{i+1} is a subgraph of T_i, representing the stable types for layer i.

The functorial characterisation of abstract GTSs that we envision could also be helpful to provide a modular semantics to the multi-layered T-GTSs.

Relations with refinement and modularity for GTSs
We do believe that our semantical analysis of transactional mechanisms in graph transformation is original. However, the same notion of abstract GTS calls for a comparison with the approaches to refinement and modularisation proposed in the literature (see [12] and the references therein).

Transactions could be exploited to simulate modules, since the atomicity of some computations is induced by the fact that some states are classified as non-observable or unstable at the abstract level. We leave to future work the further elaboration of these ideas, as well as a comparison with the literature.

Acknowledgments. We are mostly grateful to Roberto Bruni for insightful discussions and his careful reading of a preliminary version of this paper.

References

[1] P. Baldan, A. Corradini, and U. Montanari. Unfolding of double-pushout graph grammars is a coreflection. In Ehrig et al. [7], pages 145–163.

[2] R. Bardohl, M. Minas, A. Schurr, and G. Täntzer. Application of graph transformation to visual languages. In Ehrig et al. [8], chapter 3, pages 105–180.

[3] R. Bruni and U. Montanari. Zero-safe nets: Comparing the collective and individual token approaches. *Info. & Co.*, 156(1-2):46–89, 2000.

[4] 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 *LNCS*, pages 380–426. Springer, 2001.

[5] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3/4):241–265, 1996.

[6] 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 [14], chapter 3, pages 163–245.

[7] G. Ehrig, G. Engels, H.J. Kreowski, and G. Rozenberg, editors. *Proceedings of the International Workshop on Theory and Application of Graph Transformations*, volume 1764 of *LNCS*. Springer, 1999.

[8] 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.

[9] H. Ehrig, M. Pfender, and H.J. Schneider. Graph-grammars: An algebraic approach. In R.V. Book, editor, *IEEE Conf. on Automata and Switching Theory*, pages 167–180. IEEE Computer Society Press, 1973.

[10] M. Grosse-Rhode, F. Parisi Presicce, and M. Simeoni. Refinement of graph transformation systems via rule expressions. In Ehrig et al. [7], pages 368–382.

[11] R. Heckel, A. Corradini, H. Ehrig, and M. Löwe. Horizontal and vertical structuring of graph transformation systems. *Mathematical Structures in Computer Science*, 6(6):613–648, 1996.

[12] R. Heckel, H. Ehrig, G. Engels, and G Täntzer. Classification and comparison of module concepts for graph transformation systems. In Ehrig et al. [8], chapter 17, pages 669–689.

[13] H.-J. Kreowski. *Manipulation von Graphmanipulationen*. PhD thesis, Technische Universität Berlin, 1977.

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