Counterexample-guided Abstraction Refinement for the Analysis of Graph Transformation Systems*

Barbara König and Vitali Kozioura

Institut für Formale Methoden der Informatik, Universität Stuttgart, Germany
{koenigba,koziouvi}@fmi.uni-stuttgart.de

Abstract. Graph transformation systems are a general specification language for systems with dynamically changing topologies, such as mobile and distributed systems. Although in the last few years several analysis and verification methods have been proposed for graph transformation systems, counterexample-guided abstraction refinement has not yet been studied in this setting.
We propose a counterexample-guided abstraction refinement technique which is based on the over-approximation of graph transformation systems (GTS) by Petri nets. We show that a spurious counterexample is caused by merging nodes during the approximation. We present a technique for identifying these merged nodes and splitting them using abstraction refinement, which removes the spurious run. The technique has been implemented in the AUGUR tool and experimental results are discussed.

1 Introduction

In the last years verification techniques based on counterexample-guided abstraction refinement [10] have been very successful. The idea behind this approach is to start with a coarse initial abstraction or over-approximation of a system and to check whether a certain property can be verified using this abstraction. If it can not be verified, one obtains a run in the approximation that violates this property, also called counterexample. Now either this counterexample is real or it is spurious, i.e., it has been introduced by the approximation. In the latter case the approximation is refined in such a way that the counterexample disappears. This process is repeated, however in the case of infinite-state systems there is in general no guarantee that it will terminate, since the properties to be verified are usually undecidable. The technique has been used successfully in several tools such as SLAM [8], BLAST [16] or MAGIC [9].

Abstraction is also important for graph structures that can arise in several applications, for instance as evolving pointer structures on the heap, as object graphs or as networks with mobile processes. So far, little work has been done in

* Research supported by DFG project SANDS and SFB 627 (NEXUS).

this area concerning abstraction refinement. We are only aware of [21, 20] where models of a 3-valued logics representing pointer structures are refined in the framework of shape analysis [27] by generating new instrumentation relations.

Here we are working in a different framework where we are using graph transformation systems (GTS)—instead of 3-valued logics—in order to represent and transform graph structures. Graph transformation systems (GTSs) are an expressive and useful specification formalism, allowing to describe dynamic properties of concurrent and distributed systems [26, 14]. They can be used to model systems such as pointer structures [25], object-oriented languages [11, 29] and mobile processes [22, 15].

In this paper the technique of counterexample-guided abstraction refinement is applied to the verification of graph transformation systems. Our approach is based on a (partial order) technique that approximates GTSs by Petri nets via an unfolding construction [3, 6]. More specifically, in this approach a finite over-approximation called Petri graph is constructed, which consists of a graph and a Petri net having the edges of the graph as places. The important property of the approximation obtained in this way is that each graph reachable from the start graph in the GTS can be mapped, by merging some of its nodes, to a reachable marking of the over-approximating Petri net. On the other hand there may be some markings reachable in the obtained Petri graph, which have no counterpart in the original GTS. The sequence of events in the approximation leading to such a graph is called a spurious run.

In our case spurious runs are caused by the merging of graph nodes in the construction of the over-approximation. This is similar to the concept of summary nodes in shape analysis [27]. This paper describes how to construct a more exact over-approximation by separating merged nodes for which these spurious runs disappear. This procedure can be performed repeatedly for any number of spurious runs.

We believe that the technique of identifying the reason for the spurious run is independent of the abstraction mechanism used in this paper and could also be used in other frameworks dealing with approximations of graph structures.

The techniques presented here are implemented as an extension of the tool AUGUR[1]. The experimental part of the paper compares this approach with an already existing abstraction refinement technique which reduces the number of spurious examples by constructing an over-approximation which is exact up to some pre-defined depth [5]. It is shown experimentally that counterexample-guided abstraction refinement is faster and produces smaller Petri graphs.

2 Basic Notions

In this section we describe the notions of hypergraph, GTS, Petri net and Petri graph and also show in an informal way how to construct over-approximating Petri graphs.

[1] Available from `http://www.fmi.uni-stuttgart.de/szs/tools/augur/`

For a set A we denote by A^* the set of strings over A and for a function $f : A \to B$ we denote by $f^* : A^* \to B^*$ its extension to strings.

We will in the following work with hypergraphs (also called graphs), a generalization of directed graphs, which are often more convenient for modelling.

Definition 1 (hypergraphs and hypergraph morphisms). *Let Λ be a set of labels where each label $l \in \Lambda$ has an arity $ar(l) \in \mathbb{N}$. A* labelled hypergraph *G is a tuple (V_G, E_G, c_G, l_G), where V_G is a finite set of nodes, E_G is a finite set of edges, $c_G : E_G \to V_G^*$ is a connection function and $l_G : E_G \to \Lambda$ is the labeling function satisfying $ar(l_G(e)) = |c_G(e)|$ for every $e \in E_G$. The nodes are not labelled.*

Let G and G' be two labelled hypergraphs. A hypergraph morphism *(or simply* morphism*) $\varphi : G_1 \to G_2$ consists of a pair of total functions $\varphi_V : V_{G_1} \to V_{G_2}$ and $\varphi_E : E_{G_1} \to E_{G_2}$ such that for every $e \in E_{G_1}$ it holds that $l_{G_1}(e) = l_{G_2}(\varphi_E(e))$ and $\varphi_V^*(c_{G_1}(e)) = c_{G_2}(\varphi_E(e))$. A morphism is called* edge-bijective (edge-injective) *whenever it is bijective (injective) on edges. It is an* isomorphism *whenever it is bijective on nodes and edges.*

Usually we are interested only in the structure of graphs, i.e., in graphs up to isomorphism. Furthermore we will in the following also abstract from isolated nodes, i.e., nodes not connected to any edge.

Hypergraphs can be rewritten using rules of the following kind.[2]

Definition 2 (rewriting rule). *A* rewriting rule *r is a triple (L, R, α), where L and R are hypergraphs, called left-hand side and right-hand side respectively and $\alpha : V_L \to V_R$ is an injective mapping, indicating how nodes are preserved.*

We demand that there are no isolated nodes in the left-hand side L and no isolated nodes in $V_R - \alpha(V_L)$. Additionally E_L must not be empty.

The first condition says that we abstract from isolated nodes, whereas the second is a standard requirement for unfolding-based techniques, where every rule must be consuming. Note furthermore that we do not consider rules that preserve edges of the left-hand side.

For convenience we will in the following often assume that α is an inclusion denoted by *id*, which can be enforced by renaming the nodes of the left or right-hand side appropriately, and that the node and edge sets of L and R are disjoint otherwise. That is, we demand that $V_L \subseteq V_R$ and $E_L \cap E_R = \emptyset$ which implies that the union $L \cup R$ is well-defined.

Given a hypergraph, a rewriting rule and a match of the left-hand side, we can apply this rule and replace the left-hand side by the right-hand side in the following way. Additionally we define a partial morphism ν from the original graph to the rewritten graph, keeping track of preserved nodes and edges.

[2] Although our rewriting rules can be seen within the framework of the DPO approach [12], we are using simpler rules with only discrete interfaces, where additionally the deletion of nodes is forbidden.

Definition 3 (rewriting step). *Let $r = (L, R, id)$ be a rewriting rule. A* match *of r in a hypergraph G is any morphism $\varphi : L \to G$ injective on edges. We can apply r to G according to the match φ and obtain a new graph H, written $G \Rightarrow_r H$, which is defined as follows:*

$$V_H = V_G \uplus (V_R - V_L) \qquad E_H = (E_G - \varphi(E_L)) \uplus E_R$$

and, defining $\overline{\varphi}: V_R \to V_H$ by $\overline{\varphi}(v) = \varphi(v)$ if $v \in V_L$ and $\overline{\varphi}(v) = v$ otherwise, the connection and labelling functions are given by

$$\begin{aligned} e \in E_G - \varphi(E_L) \quad &\Rightarrow \quad c_H(e) = c_G(e), \quad l_H(e) = l_G(e) \\ e \in E_R \quad &\Rightarrow \quad c_H(e) = \overline{\varphi}^*(c_R(e)), \quad l_H(e) = l_R(e) \end{aligned}$$

We also define an injective partial morphism $\nu : G \to H$ where $\nu_V : V_G \to V_H$ and $\nu_E : (E_G - \varphi(E_L)) \to E_H$ with $\nu(x) = x$ for every node or edge x.

We are now ready to define the notion of graph transformation system.

Definition 4 (graph transformation system). *A* graph transformation system (GTS) $\mathcal{G} = (\mathcal{R}, G_0)$ *is a finite set of rules together with a start hypergraph (also called initial graph).*

Example: We illustrate the definitions of this chapter with an example describing a firewall system similar to the one introduced in [4]. This system contains an (arbitrarily large) set of processes running behind a firewall (safe processes) and one process in a public area (unsafe process). Any number of safe processes (SP) and connected locations (L) can be generated during runtime. The property to verify is that the unsafe process from the public area does not penetrate the firewall. If this situation is detected, rule "Error" will be applied and an edge labelled $Error$ is created.

Fig. 1 and Table 1 depict the initial graph[3] and the rules of the firewall system. A double-headed arrow in a rule means that the rule can be applied in both directions. Numbers close to the nodes indicate the mapping α. The private and public areas are connected by the firewall (F), and initially there is one unsafe processes (UP) in the public area. Only safe processes will be generated and the firewall can be crossed in one direction only. Our aim is to show that no reachable graph contains the 0-ary edge $Error$.

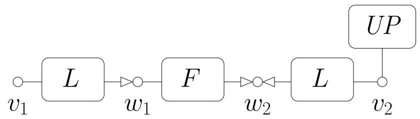

Fig. 1. Initial graph of the firewall system.

[3] The nodes of the graph are supplied with identities v_1, v_2, w_1, w_2. They will be needed later in order to refer to these nodes.

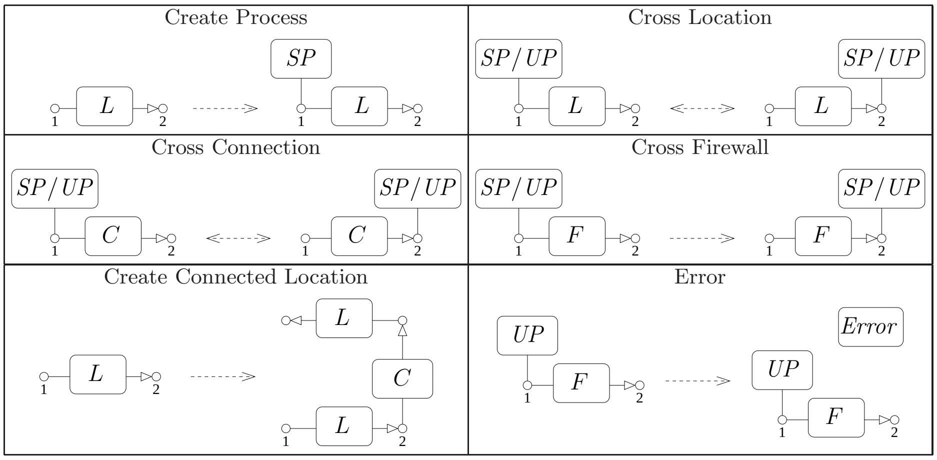

Table 1. Rules of the firewall system.

In order to approximate GTSs we will employ Petri nets, which, as multiset rewriting systems, can be seen as a special case of graph rewriting. Petri nets are an easier model than GTS and hence more amenable to analysis, several algorithms and tools are available for their verification. Furthermore, by approximating with Petri nets we will be able to preserve nice properties of the GTS model, such as locality (state changes are only described locally) and concurrency (no unnecessary interleaving of events) in the approximation.

In order to introduce Petri nets, we need the following notation: By $A^{\oplus}$ we denote a multiset over A and for a function $f : A \to B$ we denote by $f^{\oplus} : A^{\oplus} \to B^{\oplus}$ its extension to multisets. Furthermore for $m \in A^{\oplus}$ and $a \in A$ we denote by $m(a)$ the multiplicity of a in m.

Definition 5 (Petri net). *Let Δ be a finite set of labels. A Δ-*labelled Petri net *is a tuple $N = (S, T, {}^{\bullet}(), ()^{\bullet}, p)$, where S is the set of places, T is a set of transitions, ${}^{\bullet}(), ()^{\bullet} : T \to S^{\oplus}$ assign to each transition its pre-set and post-set and $p : T \to \Delta$ assigns a label to each transition. A marked Petri net is a pair (N, m_N), where N is a Petri net and $m_N \in S^{\oplus}$ is the initial marking.*

3 Approximated Unfolding

In this section we will give a short overview of a technique that approximates a graph transformation system by a structure that is both a Petri net and a hypergraph [3–6].

First we define the notion of Petri graph which will be used to represent an over-approximation for a given GTS. Note that the edges of the graph are at the same time the places of the net and that the transitions are labelled with rules of the GTS.

Definition 6 (Petri graph). *Let $\mathcal{G} = (\mathcal{R}, G_0)$ be a* GTS. *A* Petri graph *(over $\mathcal{R}$) is a tuple $P = (G, N, \mu)$, where G is a hypergraph, $N = (E_G, T_N, {}^\bullet(), ()^\bullet, p_N)$ is an $\mathcal{R}$-labelled Petri net where the places are the edges of G and μ associates to each transition $t \in T_N$, with $p_N(t) = (L, R, id)$, a hypergraph morphism $\mu(t) : L \cup R \to G$ such that ${}^\bullet t = \mu(t)^\oplus(E_L)$ and $t^\bullet = \mu(t)^\oplus(E_R)$.*

A Petri graph for the GTS *$\mathcal{G}$ is a pair (P, ι), where $P = (G, N, \mu)$ is a Petri graph over $\mathcal{R}$ and $\iota : G_0 \to G$ is a graph morphism. A marking is reachable (coverable) in Petri graph if it is reachable (coverable) in the underlying Petri net with the multiset $\iota^\oplus(E_{G_0})$ as the initial marking.*

We view Petri graphs as symbolic representations of transition systems with graphs as states. Specifically each marking m of a Petri graph can be seen as representation of a graph, denoted by $\mathit{graph}(m)$, according to the following definition. We take the marked subgraph of G and duplicate each edge as indicated by the marking.

Definition 7 (graph generated by a marking). *Let $P = (G, N, m_0)$ be a Petri graph and let $m \in E_G^\oplus$ be a marking of N. The* graph generated by m, *denoted by $\mathit{graph}_G(m)$ or $\mathit{graph}(m)$, is the graph H defined as follows: $V_H = \{v \in V_G \mid \exists e \in m \,\exists i : (c_G)_i(e) = v\}$, $E_H = \{(e, i) \mid e \in m \wedge 1 \leq i \leq m(e)\}$, $s_H((e, i)) = s_G(e)$, $t_H((e, i)) = t_G(e)$ and $l_H((e, i)) = l_G(e)$. (Note that by $(c_G)_i(e)$ we denote the i-the node in the sequence $c_G(e)$.)*

Alternatively one can define $\mathit{graph}(m)$ as the unique graph H, up to isomorphism, such that H has no isolated nodes and there exists a morphism $\psi : H \to G$ injective on nodes with $\psi^\oplus(E_H) = m$. Furthermore, whenever there exists a morphism $\varphi : G' \to G$ such that $\varphi^\oplus(E_{G'}) \leq m$, then there exists an edge-injective morphism $e_{m,\varphi} : G' \to \mathit{graph}(m)$ such that $\psi \circ e_{m,\varphi} = \varphi$. This morphism $e_{m,\varphi}$ will be used later in the paper.

This morphism $e_{m,\varphi}$ is not unique, since we may have several parallel edges from which an image can be chosen, but the resulting diagram consisting of $e_{m,\varphi}, \varphi, \psi$ is unique up to isomorphism.

In order to obtain a Petri graph approximating a GTS, we first need—as building blocks—Petri graphs that describe the effect of a single rule.

Definition 8 (Petri graph for a rewriting rule). *Let $r = (L, R, id)$ be a rewriting rule. By $P(t, r) = (G, N, \mu)$ we denote a Petri graph with $G = L \cup R$ and N is a net with places $S_N = E_L \cup E_R$ and one transition t such that $p_N(t) = r$, ${}^\bullet t = E_L$ and $t^\bullet = E_R$. Furthermore the morphism $\mu(t) : L \cup R \to G$ is the identity.*

Given a GTS $\mathcal{G} = (\mathcal{R}, G_0)$ one can construct an over-approximating Petri graph $\mathcal{C}_\mathcal{G}$ (also called the *covering* of $\mathcal{G}$), using the following algorithm (see [3]). It starts with a Petri graph P_0 that consists only of the start graph and computes $\mathcal{C}_\mathcal{G}$ iteratively. It is based on an unfolding technique which is combined with over-approximating folding steps which guarantee a finite approximation.

Algorithm 9 (approximated unfolding) We set $P_0 = (G_0, N_0, m_0)$, where N_0 contains no transitions, $m_0 = E_{G_0}$ and let $\iota_0 \colon G_0 \to G_0$ be the identity. As long as one of the following steps is applicable, transform P_i into P_{i+1} according to the possibilities given below (where folding steps take precedence over unfolding steps).

Unfolding: Find a rule $r = (L, R, id) \in \mathcal{R}$ and a match $\varphi : L \to G_i$ that has not yet been unfolded. Then choose a new transition t and extend P_i by attaching $P(t, r)$, i.e., take the disjoint union of both Petri graphs and factor through the equivalence $\equiv$ generated by $e \equiv \varphi(e)$ for every $e \in E_L$.

Folding: Find a rule $r = (L, R, id) \in \mathcal{R}$ and two matches $\varphi, \varphi' : L \to G_i$ such that $\varphi^{\oplus}(E_L)$ and $\varphi'^{\oplus}(E_L)$ are coverable in N_i and the second match is causally dependent on the transition unfolding the first match. Then merge the two matches by setting $\varphi(e) \equiv \varphi'(e)$ for each $e \in E_L$ and factoring through the resulting equivalence relation $\equiv$.

If neither possibility applies the Petri graph P_i obtained in the last step is returned. The result is denoted by $\mathcal{C}_{\mathcal{G}}$. In [3] it has been shown that the algorithm always terminates with a result unique up to isomorphism.

Example: We illustrate the algorithm using the rules of the firewall example (see Table 1) by starting with an initial graph consisting of a single binary edge labelled L. Table 2 shows the Petri graphs obtained after the first two steps of Algorithm 9. In the first step the edge labelled L is unfolded using rule "Create Process" and in the second step the two L-labelled edges are merged since they are causally dependent on each other and are both matches of rule "Create Process". Note that transitions modelling the consumption and production of tokens can be seen as specifying the deletion and creation of edges.

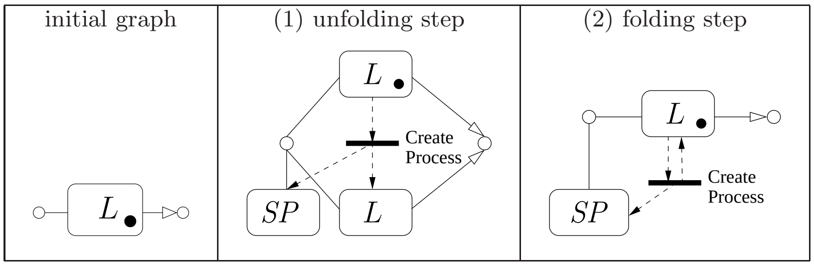

Table 2. The first two steps of the unfolding algorithm for the firewall example (with a modified initial graph).

In our running example, the constructed over-approximation consists of the hypergraph in Fig. 2 and the Petri net in Fig. 3. (Ignore the two highlighted transitions for the moment.) Note that the set of edges of the graph corresponds exactly to the set of places of the net (the correspondence is indicated by giving indices to the labels). Furthermore the graph generated by the initial marking is given on the left-hand side of Fig. 4.

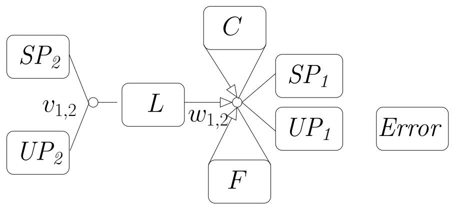

Fig. 2. Hypergraph component of the approximating Petri graph (firewall example).

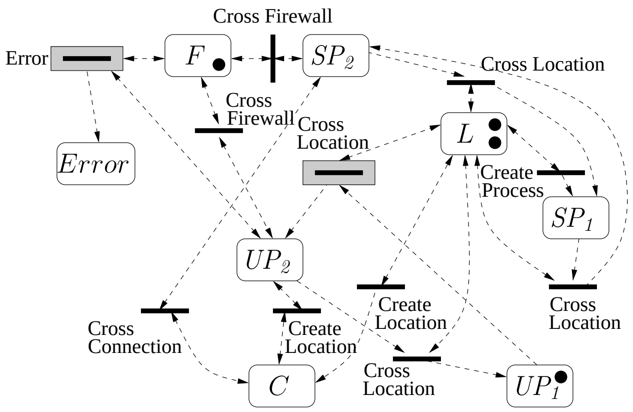

Fig. 3. Petri net component of the approximating Petri graph (firewall example).

Before we can show in what way Petri graphs can be considered as abstractions of GTSs and before we discuss how they can be analyzed, we first need the definition of an abstract run of a GTS and a notion of correspondence of two abstract runs. Then we can define how Petri graphs can be seen as abstractions of GTSs.

Definition 10 (Abstract run). *An* abstract run *of a* GTS $(\mathcal{R}, G_0)$ *is a sequence of hypergraphs* $\mathcal{J} = (J_0 \Rrightarrow_{r_1} J_1 \Rrightarrow_{r_2} \dots \Rrightarrow_{r_n} J_n)$, *where* r_i *is a rule name, together with morphisms* $\varphi_i : L_{i+1} \to J_i$ *for each* $i = 1, \dots, n-1$, *where* L_i *is the left-hand side of rule* $r_i \in \mathcal{R}$.

Note that we do not demand that J_i *can be derived from* J_{i-1} *by applying rule* r_i *at match* φ_i. *If this is the case* $\mathcal{J}$ *will be called a* real run *and we will also use the symbol* $\Rightarrow$ *instead of* $\Rrightarrow$.

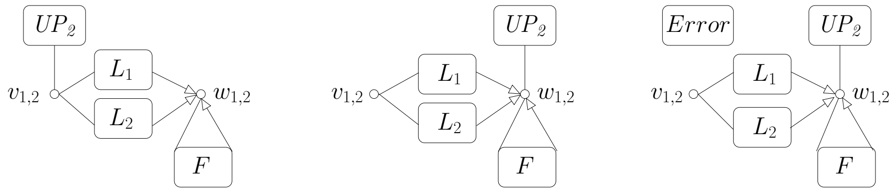

Fig. 4. Left: Graph $graph(m_0)$ generated by the initial marking. Middle: $graph(m_1)$, where $m_0[\text{“Cross Location”}\rangle m_1$. Right: $graph(m_2)$, where $m_1[\text{“Error”}\rangle m_2$. (See also the highlighted transitions in Fig. 3)

Let $\mathcal{J}' = (J_0' \Rrightarrow_{r_1} J_1' \Rrightarrow_{r_2} \ldots \Rrightarrow_{r_n} J_n')$ be another abstract run with morphisms $\varphi_i' \colon L_{i+1} \to J_i'$ for each $i = 1, \ldots, n-1$. We say that $\mathcal{J}'$ weakly corresponds *to $\mathcal{J}$ (in symbols $\mathcal{J}' \ll \mathcal{J}$) if there exist edge-bijective morphisms $\xi_i : J_i' \to J_i$ for $i = 0, \ldots, n$. If furthermore the following diagram commutes for $i = 0, \ldots, n-1$ we say that $\mathcal{J}'$* corresponds *to $\mathcal{J}$ and write $\mathcal{J}' \lll \mathcal{J}$.*

$$\begin{tikzcd} L_{i+1} \arrow[r, "\varphi_i'"] \arrow[rr, bend right, "\varphi_i"'] & J_i' \arrow[r, "\xi_i"] & J_i \end{tikzcd}$$

Petri graphs can, as mentioned above, be seen as symbolic representations of graph transition systems and also as representations of sets of abstract runs.

Definition 11 (Abstract runs of a Petri graph). *Let (P, ι) with $P = (G, N, \mu)$ be a Petri graph for a* GTS *$(\mathcal{R}, G_0)$. Furthermore let $m_0[t_1\rangle \ldots [t_n\rangle m_n$ be a firing sequence of the net N and let $r_i = p_N(t_i)$ be the rules corresponding to the transitions. We define morphisms $\varphi_i = e_{m_i, \mu(t_{i+1})|_{L_{i+1}}} : L_{i+1} \to graph(m_i)$, where L_{i+1} is the left-hand side of rule r_{i+1}. The sequence $graph(m_0) \Rrightarrow_{r_1} graph(m_1) \Rrightarrow_{r_2} \ldots \Rrightarrow_{r_n} graph(m_n)$ together with the morphisms φ_i is an* abstract run. *We denote by $Run_A(P, \iota)$ the set of all abstract runs of the Petri graph (P, ι).*

Each real run $\mathcal{J}_r = (G_0 \Rightarrow_{r_1} G_1 \Rightarrow_{r_2} \ldots \Rightarrow_{r_n} G_n)$ of the GTS $(\mathcal{R}, G_0)$ can be considered as an abstract run where the $\varphi_i : L_{i+1} \to G_i$ represent the matches of the left-hand sides of the rules r_i.

Proposition 1. *Let $\mathcal{C}_\mathcal{G}$ be an over-approximation for a* GTS *$\mathcal{G}$ computed by Algorithm 9. Then, for every real run $\mathcal{J}_r$ of the graph transformation system there exists an abstract run $\mathcal{J} \in Run_A(\mathcal{C}_\mathcal{G})$ such that $\mathcal{J}_r$ corresponds to $\mathcal{J}$, i.e., $\mathcal{J}_r \lll \mathcal{J}$.*

An abstract run $\mathcal{J}$ for which there does not exist a real run corresponding to $\mathcal{J}$ is called *spurious*. If, at the same time, it violates the property we attempt to verify, it is called a *counterexample* or *error trace*.

We can now verify the GTS by analyzing the Petri graph underlying the Petri net. For instance, in order to show that no reachable graph contains a subgraph

G_s we add a new rule to the GTS with G_s as left-hand side and an edge with a new label *Error* in the right-hand side (see rule "Error" in Table 1). If we can show that either no place labelled *Error* exists in the net or every such place is not coverable (this can be done using coverability graphs [17] or backward reachability algorithms [1]), then we can deduce that this property holds.

However, if the approximation is too coarse, we might not be able to verify the property. The approximated unfolding can then be refined by forbidding certain folding steps. In the extension described in [5] we forbid to merge items in the unfolding the depth of which is smaller than some fixed constant k. This eliminates all spurious runs of length smaller than k and gives us a sequence of better and better approximations (called k-coverings) which, in the limit, converge to the full unfolding which is obtained by unfolding without folding and which is in general infinite. However this kind of refinement of the approximation does not take into account the property which should be checked and the subsequently better approximations grow in size fairly rapidly. Therefore we will now show how to successfully apply the technique of counterexample-guided abstraction refinement in our framework.

4 Abstraction Refinement

In order to eliminate spurious runs, we will show that they are always caused by the fact that certain nodes were merged. We will identify these nodes and show how to avoid their being merged in the next iteration, thereby avoiding this particular spurious run and all other abstract runs corresponding to it in a sense made precise later. Merging of nodes is harmful since it might produce new left-hand sides, thereby leading to additional rewriting steps. On the other hand, merging of edges is harmless as long as it does not cause the merging of nodes, since we count multiplicities of edges using tokens and so no information can be lost in this way.

4.1 Spurious Runs

For a given abstract run $\mathcal{J} = (\mathit{graph}(m_0) \Rrightarrow_{r_1} \mathit{graph}(m_1) \Rrightarrow_{r_2} \dots \Rrightarrow_{r_n} \mathit{graph}(m_n))$ of the Petri graph with morphisms $\varphi_i : L_{i+1} \to \mathit{graph}(m_i)$ we define $\mathcal{H}$ to be the set of real runs corresponding to the prefixes of $\mathcal{J}$. Furthermore let $\mathcal{H}_i$ be the set of hypergraphs reachable after i steps in a real run $\mathcal{J}_r \in \mathcal{H}$. It holds that $\mathcal{H}_0 = \{G_0\}$.

An abstract run $\mathcal{J}$ is spurious if $\mathcal{H}_n = \emptyset$. If the run is spurious, there exists a k such that $\mathcal{H}_k \neq \emptyset$, but $\mathcal{H}_{k+1} = \emptyset$ (and therefore also $\mathcal{H}_l = \emptyset$ for $l > k$). It will be shown in the following how to construct a new refined over-approximation $\mathcal{C}'_{\mathcal{G}}$, which does not contain $\mathcal{J}$ and some other spurious runs corresponding to $\mathcal{J}$.

Example: We illustrate the idea of a spurious abstract run with the run corresponding to the firing of the highlighted transitions "Cross Location" and "Error" in Fig. 3. We obtain markings m_1, m_2 and two graphs $\mathit{graph}(m_1)$, *graph*(m_2) generated by these markings (see Fig. 4). One can see that—due to over-approximation and the presence of the "looping firewall"—the unsafe process is now located in front of and behind the firewall at the same time in graph *graph*(m_1). This is the reason why the second transition "Error" can be applied in the over-approximation while this is not possible in the GTS. Hence the run is spurious and no real run corresponds to it.

4.2 Relations on Nodes for Refining Abstract Runs

According to Algorithm 9 and Definitions 7 and 10 it holds that $\mathcal{H}_k \neq \emptyset$ and $\mathcal{H}_{k+1} = \emptyset$ if and only if for each $G \in \mathcal{H}_k$ there exists *no* edge-injective morphism $\eta : L_{k+1} \to G$ such that the following diagram commutes, where ξ_k is an edge-bijective morphism derived from the correspondence property (see Definition 10). In other words: there is no way to find a match of the left-hand side in G that agrees with the abstract run.

$$\begin{tikzcd} L_{k+1} \arrow[r, dotted, "\eta"] \arrow[rr, bend right, "\varphi_k"'] & G \arrow[r, "\xi_k"] & graph(m_k) \end{tikzcd}$$

For if there were such a match morphism η, we could rewrite G to G' with rule r_{k+1} corresponding to the transition transforming m_k to m_{k+1}. Because of the construction of the Petri graph, where the right-hand side of r_{i+1} has been attached during an unfolding step, we would then be able to find an edge-bijective morphism $\xi_{k+1}\colon G' \to graph(m_{k+1})$ thus continuing the correspondence.

Such a situation is only possible if ξ_k is non-injective on some nodes of G, i.e., these nodes were merged during construction of the over-approximation $\mathcal{C}_\mathcal{G}$, which is the reason for the spurious run.

Example: In our running example the nodes v_1 and v_2 as well as w_1 and w_2 of the initial hypergraph have been merged by the over-approximation, becoming $v_{1,2}$ and $w_{1,2}$ (see Fig. 1 and 2). This led to the spurious abstract run depicted in Figure 4, which was obtained by firing the transitions "Cross Location" and "Error" of the Petri net in Fig. 3.

Fig. 5 shows this abstract run, together with the left-hand sides of the rules, the real graphs contained in $\mathcal{H}$ and the corresponding morphisms. Note that $\mathcal{H}_2 = \emptyset$, which corresponds to the fact that the run is spurious. Specifically there exists no morphism $L_2 \to G_1$ that makes the diagram commute.

We will now show how to determine the node merges which caused the spurious run. Consider, for a fixed graph G and a morphism ξ_k, the set Θ of possible equivalence relations $\sim$ on nodes of a graph $G \in \mathcal{H}_k$ such that, after merging the nodes in each equivalence class, we can find an appropriate match of the left-hand side L_{k+1} in the graph $G/_\sim$. More formally, we demand the existence of an edge-injective morphism $\eta' : L_{k+1} \to G/_\sim$ such that the following diagram commutes, where $\xi'_k : G/_\sim \to graph(m_k)$ is obtained by quotienting ξ_k according to $\sim$.

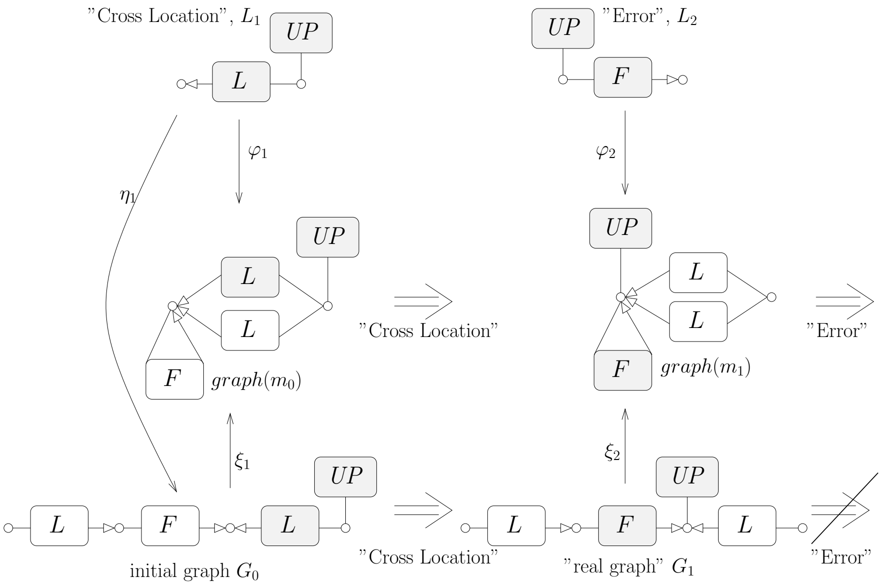

Fig. 5. Abstract run $\mathcal{J} = graph(m_0) \Rrightarrow_{r_1} graph(m_1) \Rrightarrow_{r_2} graph(m_2)$, real run $G_0 \Rightarrow_{r_1} G_1$ and the corresponding morphisms.

$$\begin{tikzcd} L_{k+1} \arrow[r, "\eta'"] \arrow[rr, bend right, "\varphi_k"'] & G/{\sim} \arrow[r, "\xi'_k"] & graph(m_k) \end{tikzcd}$$

In order to characterize the smallest equivalence in Θ consider a node v of the left-hand side and determine a set Q_v of nodes in G which have to be fused into one node which is the image of v under η'. Let $v \in V_{L_{k+1}}$ and let e be an edge of L_{k+1} with $c_i(e) = v$ for some i. For every edge e' in G with $\xi_k(e') = \varphi_k(e)$ we require that $c_i(e') \in Q_v$.

Consider the relation $\mathcal{Q}$, where for each $v \in V_{L_{k+1}}$ all nodes in Q_v are related and the relation $\widehat{\mathcal{Q}}$ which is the smallest equivalence containing $\mathcal{Q}$.

Proposition 2. *The equivalence $\widehat{\mathcal{Q}}$ constructed above is the smallest equivalence contained in Θ.*

Proof. First note that the set Q_v can also be defined as follows:

$$Q_v = \{c_i(\xi_k^{-1}(\varphi_k(e))) \mid e \in E_{L_{k+1}}, v = c_i(e)\}.$$

Let $\widehat{\mathcal{Q}}_v$ be the equivalence class in $\widehat{\mathcal{Q}}$ containing $\mathcal{Q}_v$. Note that for each $v_1, v_2 \in L$: $\widehat{\mathcal{Q}}_{v_1} \cap \widehat{\mathcal{Q}}_{v_2} = \emptyset$ or $\widehat{\mathcal{Q}}_{v_1} = \widehat{\mathcal{Q}}_{v_2}$.

We assume that there is a morphism $\mu\colon G \to G/\widehat{\mathcal{Q}}$ mapping every edge to itself and every node $w \in G$ to the equivalence class $\widehat{\mathcal{Q}}_v$, where $w \in \widehat{\mathcal{Q}}_v$. Furthermore there is a morphism $\xi'_k\colon G/\widehat{\mathcal{Q}} \to \mathit{graph}(m_k)$ mapping every edge e to $\xi_k(\mu^{-1}(e))$ and every node $\widehat{Q_v}$ to $\xi_k(w)$ where $w \in Q_v$. This is well-defined since $w_1 \,\mathcal{Q}\, w_2$ implies $\xi_k(w_1) = \xi_k(w_2)$ and $\widehat{\mathcal{Q}}$ is a transitive closure of $\mathcal{Q}$. Therefore $w_1 \,\widehat{\mathcal{Q}}\, w_2$ implies $\xi_k(w_1) = \xi_k(w_2)$.

Now let us define a morphism $\eta'\colon L_{k+1} \to G/\widehat{\mathcal{Q}}$ and show that it is a hypergraph morphism. Set $\eta'(v) = \widehat{Q}_v$ for each $v \in V_L$ and $\eta'(e) = \xi_k^{-1}(\varphi_k(e))$ for each $e \in E_L$. Since φ_k is injective on edges and ξ_k is bijective on edges, η' is also injective on edges. In order to show that η' is a morphism we have to prove that $\eta'(c_i(e)) = c_i(\eta'(e))$. Let $v = c_i(e)$. We have $\eta'(c_i(e)) = \eta'(v) = \widehat{Q}_v$. On the other hand $c_i(\eta'(e)) = c_i({\xi'}_k^{-1}(\varphi_k(e))) = c_i(\mu(\xi_k^{-1}(\varphi_k(e)))) = \mu(c_i(\xi_k^{-1}(\varphi_k(e)))) = \widehat{Q}_v$, since $c_i(\xi_k^{-1}(\varphi_k(e))) \in Q_v$.

We now have the following situation:

$$\begin{tikzcd}
L_{k+1} \arrow[r, "\eta'"'] \arrow[rr, bend left, "\varphi_k"] & G/\widehat{\mathcal{Q}} \arrow[r, "\xi'_k"'] & \mathit{graph}(m_k) \\
& G \arrow[u, "\mu"] \arrow[ur, "\xi_k"'] &
\end{tikzcd}$$

Next we show that the diagram above commutes, i.e., $\xi'_k \circ \eta' = \varphi_k$. By definition it commutes on edges. For nodes we show that $\xi'_k(\widehat{Q}_v) = \varphi_k(v)$ for each $v \in V_L$. Let $w \in \mathcal{Q}_v$. According to the definition $w = c_i(\xi_k^{-1}(\varphi_k(e)))$ for an edge e and an index i where $c_i(e) = v$. We have $\xi'_k(\eta'(v)) = \xi'_k(\widehat{Q}_v) = \xi_k(w) = \xi_k(c_i(\xi_k^{-1}(\varphi_k(e)))) = \varphi_k(c_i(e)) = \varphi_k(v)$

We have proved that $\widehat{\mathcal{Q}} \in \Theta$. Now we show that $\widehat{\mathcal{Q}}$ is the smallest equivalence relation in Θ. Let $\widetilde{\mathcal{Q}}$ be another equivalence relation from Θ where $\tilde{\eta}\colon L_{k+1} \to G/\widetilde{\mathcal{Q}}$, $\tilde{\mu}\colon G \to G/\widetilde{\mathcal{Q}}$ and $\tilde{\xi}_k\colon G/\widetilde{\mathcal{Q}} \to \mathit{graph}(m_k)$ such that $\tilde{\xi}_k \circ \tilde{\eta} = \varphi_k$ and $\tilde{\xi}_k \circ \tilde{\mu} = \xi_k$.

Let $w_1 \,\mathcal{Q}\, w_2$ where $w_1, w_2 \in V_G$ and $w_1 \neq w_2$. That means that there are edges $e_1, e_2 \in E_L$ and indexes j_1, j_2 such that $w_i = c_{j_i}(\xi_k^{-1}(\varphi_k(e_i)))$ and there exists a node $v \in V_L$ such that $c_{j_i}(e_i) = v$ for $i = 1, 2$. It holds that

$$\begin{aligned}
\tilde{\mu}(w_i) &= \tilde{\mu}(c_{j_i}(\xi_k^{-1}(\varphi_k(e_i)))) = c_{j_i}(\tilde{\mu}(\xi_k^{-1}(\varphi_k(e_i)))) = c_{j_i}(\tilde{\xi}_k^{-1}(\varphi_k(e_i))) \\
&= c_{j_i}(\tilde{\eta}(e_i)) = \tilde{\eta}(c_{j_i}(e_i)) = \tilde{\eta}(v)
\end{aligned}$$

Note that $\hat{\xi}_k^{-1}(\varphi_k(e_i)) = \hat{\eta}(e_i)$ only holds since $\hat{\eta}, \varphi_k$ are injective on edges and $\hat{\xi}_k^{-1}$ is bijective on edges.

Hence we have $\tilde{\mu}(w_1) = \tilde{\mu}(w_2)$ and w_1, w_2 must be in the same equivalence class according to $\widetilde{\mathcal{Q}}$. This means $\mathcal{Q} \subseteq \widetilde{\mathcal{Q}}$. As we know $\widehat{\mathcal{Q}}$ is the the smallest equivalence relation containing $\mathcal{Q}$. Therefore $\widehat{\mathcal{Q}} \subseteq \widetilde{\mathcal{Q}}$. $\square$

Example: We consider again the abstract error trace $\mathcal{J}$ which can be obtained by firing transitions "Cross Location" and "Error". However, this error trace has no real runs that correspond to it, which can be seen by computing the set $\mathcal{H}$ of runs corresponding to prefixes of $\mathcal{J}$. Here, the set $\mathcal{H}_0$ consists of the initial hypergraph and the set $\mathcal{H}_1$ contains one graph G_1. The next rule "Error" cannot be applied to G_1 in such a way that the corresponding diagram commutes and therefore the set $\mathcal{H}_2$ is empty.

Fig. 6 shows the left-hand side of rule "Error", $G_1 \in \mathcal{H}_1$ and $graph(m_1)$, the graph corresponding to the marking reached after one step (see also Fig.4 and Fig. 5). One notices that no appropriate morphism η can be found unless the nodes w_1 and w_2 in G_1 are merged. Therefore we have $Q_{w'_1} = \{w_1, w_2\}$, $Q_{w'_2} = \{w_2\}$ and the smallest equivalence relation $\widehat{\mathcal{Q}}$ relates the nodes w_1 and w_2 and no other nodes. Note for instance that w_2 must be contained in $Q_{w'_1}$ since both are attached to the unary edge labelled UP.

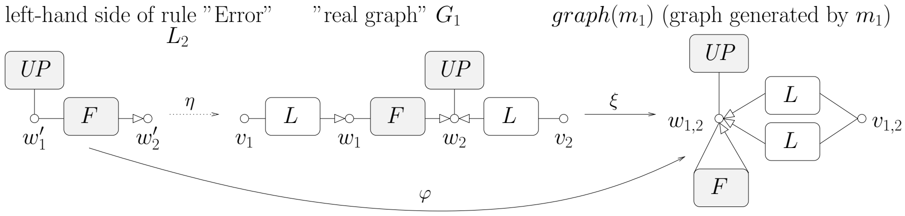

Fig. 6. Hypergraphs $G_1 \in \mathcal{H}_1$, L_2 and $graph(m_1)$ from the firewall example.

4.3 Elimination of Spurious Runs

The general idea for destroying spurious runs is to avoid the merging of nodes from the same equivalence class of $\widehat{\mathcal{Q}}$. For this reason we assign colours to the nodes of the graphs contained in $\mathcal{H}$ and disallow the merging of nodes corresponding to nodes with the same colour. For reasons that will become clear below a node may have several colours, i.e., a node v is associated to a set $cols(v)$ of colours.

For each $G \in \mathcal{H}_k$ and each morphism $\xi_k : G \to graph(m_k)$ we consider the corresponding relation $\mathcal{Q}_{G,\xi_k}$. Then we assign colours to nodes in such a way that there exists at least one pair v_1, v_2 of nodes such that $v_1 \,\mathcal{Q}_{G,\xi_k}\, v_2$ and $cols(v_1) \cap cols(v_2) \neq \emptyset$. There are several ways to do this and all of them will help to eliminate the counterexample. In our implementation we choose a color for each set of nodes Q_v and assign it to all nodes contained in Q_v.

In order to catch "bad" mergings as early as possible, these colours have to be distributed to the remaining graphs contained in $\mathcal{H}$. Let us recall here that according to Definition 3 for each real run $\mathcal{J}_r = (G_0 \Rightarrow_{r_1} G_1 \ldots \Rightarrow_{r_k} G_k)$ from $\mathcal{H}$ we have injective partial morphisms $\nu_i : G_i \to G_{i+1}$ for $i = 0, \ldots, k-1$. Using these partial morphisms we assign the colours of G_k to the remaining graphs G_i contained in $\mathcal{H}$. We start from G_k and proceed as follows: if a node $v \in G_{i+1}$ has a colour then we also assign this colour to the node $\nu^{-1}(v)$ if such a node exists. In this way a node may obtain several colours, due to the branching structure of the runs contained in $\mathcal{H}$. We denote by $cols(v)$ the set of colours of the node $v \in V_{G_j}$ where $G_j \in \mathcal{H}_j$.

We are now ready to present the algorithm for computing the refined over-approximation.

Algorithm 12 (Refined approximated unfolding)

Input: A GTS $\mathcal{G}$, a set $\mathcal{H}$ of runs corresponding to prefixes of the counterexample and a function *cols* assigning sets of colours to the nodes of the graphs in $\mathcal{H}$.

Output: The refined over-approximation $\mathcal{C}'_{\mathcal{G}}$.

We start constructing the new over-approximation $\mathcal{C}'_{\mathcal{G}}$ with the initial graph G_0. Unfolding steps will be performed as described in Algorithm 9.

For a folding step we disallow the merging of nodes corresponding to nodes in $\mathcal{H}$ having the same colour. More specifically, consider the over-approximation $\mathcal{C}'_{\mathcal{G}}$, which is currently being constructed. Now for each run $\mathcal{J}_r = G_0 \Rrightarrow_{r_1} \ldots \Rrightarrow_{r_\ell} G_\ell$ in $\mathcal{H}$ where $\ell < k$ check the following:

We consider all abstract runs $\mathcal{J} = graph(m_0) \Rrightarrow_{r_1} \ldots \Rrightarrow_{r_\ell} graph(m_\ell)$ of the current Petri graph $\mathcal{C}'_{\mathcal{G}}$ for which $\mathcal{J}_r \ll \mathcal{J}$ and all edge-bijective morphisms $\xi\colon G_i \to graph(m_i)$ for $i = 0, \ldots, \ell$. Whenever there are two nodes v_1, v_2 in G_i with $cols(v_1) \cap cols(v_2) \neq \emptyset$ and $\xi(v_1) = \xi(v_2)$, we have erroneously merged two nodes in the approximation which should not have been merged. Consequently this folding step is undone.

Previously rejected folding steps are recorded and are not any more considered by the algorithm.

In this way we will eliminate not only the spurious run but several more runs which are characterized below (see Proposition 5).

Before continuing with the running example, the following two remarks are in order: First note that the check in Algorithm 12 which is performed for each folding step can be done in an efficient way by following the branching structure of the runs instead of enumerating all runs. Second, refining the abstraction directly without constructing it again from scratch as we have done, is a non-trivial undertaking since Petri graphs are very compact descriptions of the state space in which states can not be easily separated. Doing this in an efficient way is a direction of future work.

Example: Fig. 7 depicts the hypergraph obtained for the firewall example after the abstraction refinement procedure. As one can see, the "critical nodes" of the hypergraph, namely the nodes w_1 and w_2, are now separated.

4.4 Correctness

In the following we will show that Algorithm 12 terminates and that the refined over-approximation is correct and more exact than the previous one.

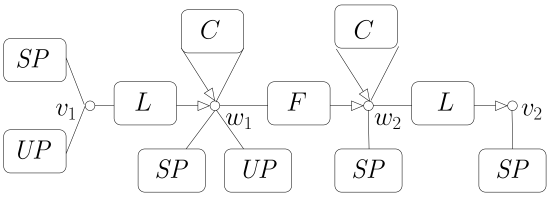

Fig. 7. Hypergraph obtained after abstraction refinement.

Let $\mathcal{C}_\mathcal{G}$ be an over-approximation with a spurious run $\mathcal{J}$ and let $\mathcal{C}'_\mathcal{G}$ be the corresponding refined over-approximation. In [3] it is shown that the algorithm constructing the over-approximating Petri graph terminates. We modified the algorithm by forbidding some of the folding steps and hence we have to reprove termination for the new version of the algorithm.

Proposition 3. *The algorithm computing the refined over-approximation $\mathcal{C}'_\mathcal{G}$ for a given* GTS *$\mathcal{G}$ and a (spurious) abstract run $\mathcal{J}$ of $\mathcal{C}_\mathcal{G}$ terminates.*

Proof (Sketch). By a slight modification of the termination proof for the approximated unfolding algorithm in [5]. □

Furthermore the new over-approximation is still a valid over-approximation as before.

Proposition 4. *Let $\mathcal{C}'_\mathcal{G}$ be the refined over-approximation of the* GTS*. Then, for every real run $\mathcal{J}_r$ of the graph transformation system there exists an abstract run $\mathcal{J} \in Run_A(\mathcal{C}'_\mathcal{G})$ such that $\mathcal{J}_r$ corresponds to $\mathcal{J}$, i.e., $\mathcal{J}_r \lll \mathcal{J}$.*

Proof. This follows directly from the construction of the over-approximation, since the construction starts with the initial graph and every coverable left-hand side is unfolded at some point. □

In the following two propositions we will show that we have eliminated the given spurious counterexample and have not added any new ones. First we should answer the following question: what kind of runs have we eliminated by abstraction refinement? It is easy to see that in the refined over-approximation we have lost the initial spurious counter-example $\mathcal{J}$. In fact we have not only eliminated $\mathcal{J}$, but some more runs as described below.

Definition 13 (Correspondence with respect to runs). *Let (P, ι) and (P', ι') be two Petri graphs for a* GTS *$(\mathcal{R}, G_0)$. Furthermore let $\mathcal{J} \in Run_A(P, \iota)$ and $\mathcal{J}' \in Run_A(P', \iota')$ be two abstract runs of these Petri graphs and let $\mathcal{H}$ be the set of real runs considered earlier. We say that $\mathcal{J}'$* corresponds to $\mathcal{J}$ with respect to $\mathcal{H}$ *if $\mathcal{J}'$ corresponds to $\mathcal{J}$ and a run $\mathcal{J}'' \in \mathcal{H}$ of maximal length weakly corresponds to a prefix of $\mathcal{J}'$, i.e., $\mathcal{J}' \lll \mathcal{J}$ and $\mathcal{J}'' \ll pr_{|\mathcal{J}''|}(\mathcal{J}')$ for some $\mathcal{J}'' \in \mathcal{H}$. (By $pr_\ell(\mathcal{J})$ we denote the prefix of length ℓ of a run $\mathcal{J}$.)*

Using this definition we can now state and prove the following propositions.

Proposition 5. *The refined over-approximation $\mathcal{C}'_{\mathcal{G}}$, constructed above does not contain any run $\mathcal{J}'$ corresponding to the spurious run $\mathcal{J}$ of $\mathcal{C}_{\mathcal{G}}$ with respect to $\mathcal{H}$.*

Proof. Let us consider a respect to $\mathcal{H}$ and let k be the maximal index such that $\mathcal{H}_k$ is not empty.

We consider the following diagram where L_{k+1} is the left-hand side of the rule r_{k+1} and $G_k \in \mathcal{H}_k$ is a reachable hypergraph from a maximal run $\mathcal{J}''$ weakly corresponding to a prefix of $\mathcal{J}'$ ($\mathcal{J}'' \ll pr_{|\mathcal{J}''|}(\mathcal{J}')$).

$$\begin{tikzcd}
L_{k+1} \arrow[r, "\varphi'_k"'] \arrow[rr, bend left, "\varphi_k"] & \mathit{graph}(m'_k) \arrow[r, "\xi"'] & \mathit{graph}(m_k) \\
 & G_k \arrow[u, "\psi'"] \arrow[ur, "\psi"'] &
\end{tikzcd}$$

The existence of ψ'_k is implied by the weak correspondence $\mathcal{J}'' \ll pr_{|\mathcal{J}''|}(\mathcal{J}')$. The sub-diagram $\{L_{k+1},\ \mathit{graph}(m'_k),\ \mathit{graph}(m_k)\}$ is taken from the condition $\mathcal{J}' \lll \mathcal{J}$. We define $\psi = \xi \circ \psi'$. This means that the sub-diagram $\{G_k,$ $\mathit{graph}(m'_k),\ \mathit{graph}(m_k)\}$ commutes.

The morphism ψ' is edge-bijective. This implies that $\mathit{graph}(m_k)$ is isomorphic to the graph G_k factorized through an equivalence $\sim$ on nodes. This equivalence $\sim$ must be an element of the set of equivalences Θ defined in Section 4.2 of which $\widehat{\mathcal{Q}}_{G_k,\psi}$ is the smallest element according to Proposition 2. This implies $\sim \supseteq \mathcal{Q}_{G_k,\psi}$ and we can conclude that ψ' maps at least two nodes having the same colour to the same node.

Hence we have a situation where $\mathcal{J}'' \ll pr_{|\mathcal{J}''|}(\mathcal{J}')$ and there is a morphism $\psi' : G_k \to \mathit{graph}(m'_k)$ with $\psi'(v_1) = \psi'(v_2)$, where $v_1 \neq v_2$ and $\mathit{cols}(v_1) \cap \mathit{cols}(v_2) \neq \emptyset$. This is a contradiction, since this situation should have been detected by the abstraction refinement algorithm.

□

We can also show that no new spurious runs have appeared, which means that the new approximation is strictly better than the old one.

Proposition 6. *If the refined over-approximation $\mathcal{C}'_{\mathcal{G}}$ contains a spurious run $\mathcal{J}'$, then it corresponds to some spurious run $\mathcal{J}$ in $\mathcal{C}_{\mathcal{G}}$.*

Proof. This follows from the fact, that from the Petri graph P' of the refined approximation there exists a morphism $\beta : P' \to P$ to the original Petri graph P, which can be shown by induction on the number of steps of the algorithm. □

We remark that the considered abstraction refinement approach can also be implemented in the case of any number of spurious counterexamples as follows: We store the set $\mathcal{H}$ with the internal structure and check the obtained over-approximation. If again a counterexample is found and it is spurious, then we apply the algorithm above to the new set $\mathcal{H}'$ and the set $\mathcal{H}$, obtained in the previous step. This procedure can be repeated. Naturally, due to undecidability and the fact that GTSs are in general Turing-complete, there is no guarantee that it will ever terminate.

In the future we plan to show that whenever a property can be proved by forbidding folding steps up to a certain depth, then it can also be shown using counterexample-guided abstraction refinement. However, some care has to be taken concerning the order in which counterexamples are destroyed and also concerning the colours which are assigned to nodes. In this way we plan to get a result which is remotely similar to one presented in [7].

5 Implementation and Experimental Results

In this section we consider examples of GTSs and compare the experimental results obtained by refining the approximation by forbidding folding steps up to a certain depth (see [5]) and counterexample-guided abstraction refinement as presented in this paper. It is shown that for practical purposes the new technique is usually more efficient.

The algorithm was implemented in C++ under Linux and the computer parameters are 2*Xeon 2.4 GHz, 2 GB RAM. The implementation [18] is still in a prototype stage with a lot of room for improvement, for instance the detection of the matches of left-hand sides is currently implemented in a very straightforward and inefficient way. We are currently working on a new version [19].

Case study 1: The first case study treats the firewall example that we used as a running example in this paper. The property we want to verify is "an unsafe process does not penetrate the firewall".

In order to verify this property we computed the over-approximation $\mathcal{C}_\mathcal{G}$ using the tool AUGUR. Counterexamples in the underlying Petri net, i.e., runs which finally cover the place *Error*, can be found with an adaptation of the backward reachability algorithm described in [1]. The experimental results for the two forms of abstraction refinement that we consider are given in Tables 3 and 4 which show the size (number of nodes, edges and transitions) of the constructed over-approximation, the runtime and truth values indicating whether the properties under consideration can be verified. Times for successful verifications are highlighted by using boldface.

The 0-covering (where all folding steps are allowed) contains a spurious error trace and the same is true for the 1-covering (where folding of items of depth 0 is forbidden). In fact one can show that the property cannot be verified with any k-covering. The reason for this is that new locations of arbitrary depth are created that are being merged by the approximation, which also holds for locations in front of and behind the firewalls. In this way processes running at locations will—in the approximation—"move around" firewalls without actually crossing them. In the case of counterexample-guided abstraction refinement newly created locations will be merged with existing locations and this effect does not appear, which means that the property can be verified.

The second version of the firewall example has a different initial graph and the same rules as the first version. The choice of taking an alternative initial graph was made in order to show how slight variations can affect the previous abstraction refinement technique (exact unfolding up to depth k), an effect that disappears with the method presented here. The initial graph is depicted in Fig. 8 in Appendix A, where the image is automatically generated using the Graphviz package.

Case study 2: The second case study is called "public/private servers", the rules are given in Appendix A. Again the rules are automatically generated by AUGUR. In this system any number of public servers and one private server can be generated. The servers can produce processes (internal processes by the private and external by the public servers) and arbitrary connections can be created between them along which processes may move. Furthermore the private server may at some point decide to transform itself into a public server. We consider two variants of this system by using two different initial graphs.

The properties we want to verify are:

(NC) No connection will ever be created from a public to a private server (only connections going in the other direction and connections between public servers are allowed).
(EP) External processes will never access private servers.

Again we compute the over-approximation $\mathcal{C}_\mathcal{G}$ using AUGUR and we attempt to verify both properties at once using two rules (one for each property) generating the *Error*-edge (see also the rules in Appendix A, Fig. 9).

The 0-covering contains spurious error runs for both properties (NC) and (EP). Hence, these properties cannot be verified using this approximation, but they can be verified using the 1-covering. For the second variant of the "public/private servers" system the 1-depth approximation is still insufficient. In this case the properties can only be verified using the 2-covering.

If we compare the results in Tables 3 and 4 it can be seen that in the case of counterexample-guided abstraction refinement we have an advantage both in runtime for computing the approximation and in the size of the over-approximations, which are consequently easier to analyze. The difference is especially pronounced for versions II, which use larger start graphs.

The efficiency of the abstraction refinement approach can be explained by the fact that we forbid to merge only those parts of the unfolding which are responsible for the spurious counterexample. This means that the over-approximation remains rather compact compared to the depth-based (or k-covering) approach, where we are not allowed to merge all items having depth smaller than k.

Furthermore note that not only the runtime of approximated unfolding is better in the case of counterexample-guided abstraction refinement, but also the Petri net tools checking coverability on the net are substantially faster. For instance in the case study Firewall II our coverability checker (based on backward reachability) runs for more than one day in the case of the 2-covering.

example	k (depth)	nodes	edges	transitions	time (sec)	verified
Public/private servers I	0	1	9	13	0.05	no
Public/private servers I	1	2	19	34	**0.72**	**yes**
Public/private servers II	0	1	10	14	0.05	no
Public/private servers II	1	1	11	16	0.07	no
Public/private servers II	2	3	31	63	**7.16**	**yes**
Firewall I	0	2	8	13	0.05	no
Firewall I	1	6	25	50	2.4	no
Firewall I	2	10	51	148	138.18	no
Firewall II	0	2	8	13	0.14	no
Firewall II	1	8	39	82	13.7	no
Firewall II	2	14	79	242	858.4	no

Table 3. Verification results (abstraction refinement by forbidding folding steps up to a certain depth k, i.e., by computing k-coverings).

example	nodes	edges	transitions	time (sec)	verified
Public/private servers I	2	16	25	**0.67**	**yes**
Public/private servers II	2	17	26	**0.68**	**yes**
Firewall I	4	11	17	**0.16**	**yes**
Firewall II	4	12	18	**0.33**	**yes**

Table 4. Verification results (counterexample-guided abstraction refinement).

6 Conclusion

In this paper we have shown how counterexample-guided abstraction refinement can be applied to the analysis of dynamically evolving graphical structures in a fully automatic way. In this case we are not concerned with the abstraction of data values, but rather with graphs that are abstracted by merging nodes and edges, using the concept of graph morphisms. Hence, abstraction refinement can in this case be described by exploiting commutativity or rather non-commutativity of morphisms as described in Section 4. Also, since we are dealing with the approximation of graph structures rather than data values, no theorem prover is needed in order to determine the initial abstraction, instead we use techniques for approximated unfolding developed in [3].

Apart from smaller case studies we have used our approximated unfolding technique to verify a mutual exclusion protocol [13] and to verify insertion of elements into red-black trees [2]. We are currently working on an encoding of simple pointer programs into graph rewriting which will enable us to directly verify operations on pointer structures.

Research concerned with the verification of dynamically evolving graph structures which can be used to model distribution and mobility is fairly recent. There are contributions coming from the area of dataflow analysis such as shape analysis [27] as well as work directed more specifically towards the analysis of graph transformation systems [24, 23, 28, 13, 25]. We believe that introducing counterexample-based abstraction refinement is an important step in order to make such verification techniques usable in practice. We also think that some of the techniques presented here can be employed in fairly general settings.

Compared to shape analysis [27, 21] which is also concerned with over-approximation techniques for graphical structures and which represents these structures as models of a 3-valued logic, we follow a different approach where graphs are represented directly and graph morphisms are used as a convenient abstraction mechanism. Furthermore we approximate with Petri nets, which enable us to talk about multiplicities of edges and can be conveniently analyzed using a variety of existing Petri net tool.

Acknowledgments: We would like to thank Tobias Heindel, Paolo Baldan and Andrea Corradini for many interesting discussions on the topics of this paper.

References

1. P.A. Abdulla, B. Jonsson, M. Kindahl, and D. Peled. A general approach to partial order reductions in symbolic verification. In *Proc. of CAV '98*, pages 379–390. Springer, 1998. LNCS 1427.
2. P. Baldan, A. Corradini, J. Esparza, T. Heindel, B. König, and V. Kozioura. Verifying red-black trees. In *Proc. of COSMICAH '05*, 2005. Proceedings available as report RR-05-04 (Queen Mary, University of London).
3. P. Baldan, A. Corradini, and B. König. A static analysis technique for graph transformation systems. In *Proc. of CONCUR '01*, pages 381–395. Springer, 2001. LNCS 2154.

4. P. Baldan, A. Corradini, and B. König. Static analysis of distributed systems with mobility specified by graph grammars - a case study. In *Proc. of IDPT '02*. Society for Design and Process Science, 2002.
5. P. Baldan and B. König. Approximating the behaviour of graph transformation systems. In *Proc. of ICGT '02*, pages 14–29. Springer, 2002. LNCS 2505.
6. P. Baldan, B. König, and B. König. A logic for analyzing abstractions of graph transformation systems. In *Proc. of SAS '03*, pages 255–272. Springer, 2003. LNCS 2694.
7. T. Ball, A. Podelski, and S.K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *Proc. of TACAS '02*, pages 158–172. Springer, 2002. LNCS 2280.
8. T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. of SPIN '01*, pages 103–122. Springer, 2001. LNCS 2057.
9. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proc. of ICSE '03*, pages 385–395. IEEE Computer Society, 2003.
10. E. Clarke, S. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of CAV '00*, pages 154–169. Springer, 2000. LNCS 1855.
11. A. Corradini, F.L. Dotti, L. Foss, and L. Ribeiro. Translating Java code to graph transformation systems. In *Proc. of ICGT '04*, LNCS 3256, pages 383–398. Springer, 2004.
12. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation—part I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations*, chapter 3. World Scientific, 1997.
13. F.L. Dotti, L. Foss, L. Ribeiro, and O. Marchi Santos. Verification of distributed object-based systems. In *Proc. of FMOODS '03*, pages 261–275. Springer, 2003. LNCS 2884.
14. 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.
15. F. Gadducci and U. Montanari. A concurrent graph semantics for mobile ambients. In S. Brookes. and M. Mislove, editors, *Proceedings of the 17th MFPS*, volume 45 of *ENTCS*. Elsevier Science, 2001.
16. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. of POPL '02*, pages 58–70. ACM, 2002.
17. R.M. Karp and R.E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969.
18. Barbara König and Vitali Kozioura. Augur—a tool for the analysis of graph transformation systems. *EATCS Bulletin*, 87:125–137, November 2005. Appeared in The Formal Specification Column.
19. Barbara König and Vitali Kozioura. Augur 2—a new version of a tool for the analysis of graph transformation systems. In *Proc. of GT-VMT '06 (Workshop on Graph Transformation and Visual Modeling Techniques)*, 2006. ENTCS. to appear.
20. A. Loginov, T. Reps, and M. Sagiv. Abstraction refinement for 3-valued-logic analysis. Technical Report 1504, Comp. Sci. Dept., Univ. of Wisconsin, 2004.
21. A. Loginov, T. Reps, and M. Sagiv. Abstraction refinement via inductive learning. In *Proc. of CAV '05*, pages 519–533. Springer, 2005. LNCS 3576.
22. U. Montanari and M. Pistore. Concurrent semantics for the π-calculus. *Electronic Notes in Theoretical Computer Science*, 1, 1995.

23. A. Rensink. Canonical graph shapes. In *Proc. of ESOP '04*, pages 401–415. Springer, 2004. LNCS 2986.
24. A. Rensink. State space abstraction using shape graphs. In *Proc. of AVIS '04*, ENTCS, 2004. to appear.
25. A. Rensink and D. Varró. Model checking graph transformations: A comparison of two approaches. In *Proc. of ICGT '04*, pages 226–241. Springer, 2004. LNCS 3256.
26. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*. World Scientific, 1997.
27. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.
28. D. Varró. Towards symbolic analysis of visual modeling languages. In *Proc. of GT-VMT '02*, volume 72 of *ENTCS*. Elsevier, 2002.
29. A. Wagner and M. Gogolla. Defining operational behaviour of object specifications by attributed graph transformation. *Fundamenta Informaticae*, 26:407–431, 1996.

A Additional Material for the Case Studies

Here we give some additional material that is needed to fully understand the case studies treated in Section 5. The figures were generated automatically by our tool AUGUR with the help of the Graphviz package.

Figure 8 shows the alternative start graph for the firewall example.

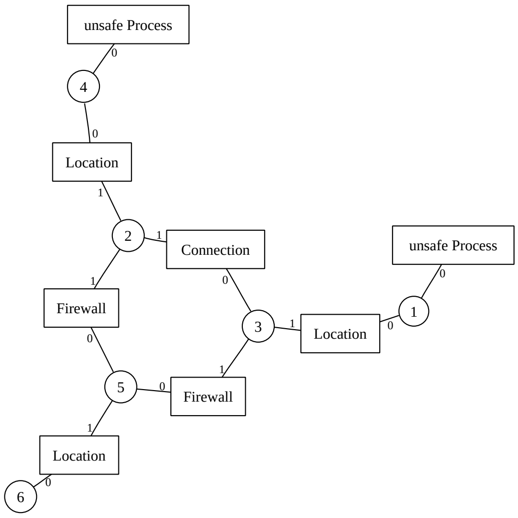

Fig. 8. Initial graph of the *Firewall II* example

In Fig. 9 we present two versions of the system "Public/Private Servers", which are being used as a case study in Section 5 of the paper. Note that the versions I and II have only two differences. First, the second version has a different initial graph. Furthermore, the second version has an additional rule ("Create Private Server"). All other rules are identical for both versions.

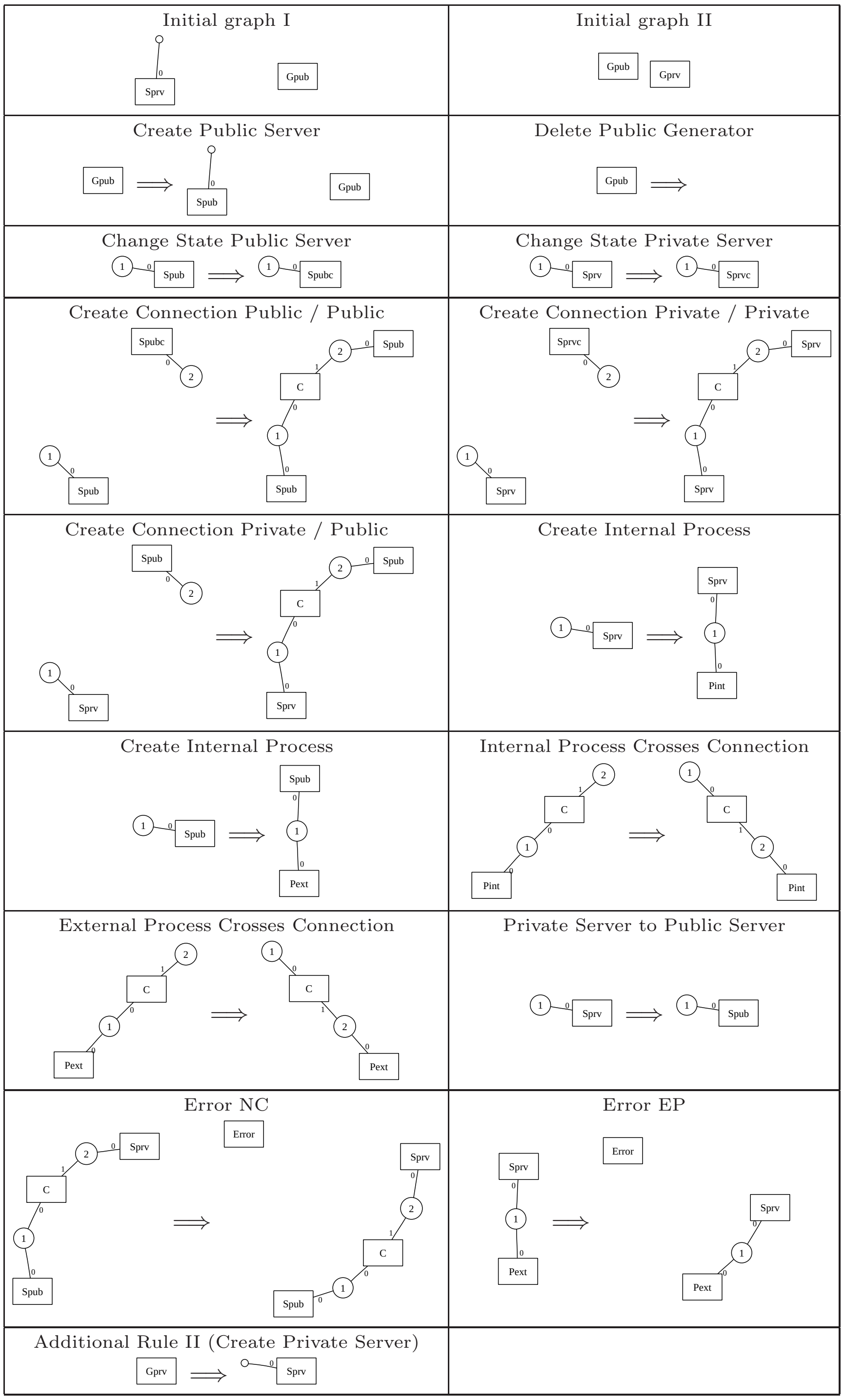

Fig. 9. Public/Private Servers I and II