Action Planning for Graph Transition Systems

Stefan Edelkamp[1] **and Shahid Jabbar**[2]
Department of Computer Science
Baroper Str. 301
University of Dortmund,
44221 Dortmund, Germany
[1]stefan.edelkamp@cs.uni-dortmund.de
[2]shahid.jabbar@cs.uni-dortmund.de

Alberto Lluch Lafuente
Dipartimento di Informatica
Università di Pisa, Pisa, Italy
lafuente@di.unipi.it

Abstract

Graphs are suitable modeling formalisms for software and hardware systems involving aspects such as communication, object orientation, concurrency, mobility and distribution. State spaces of such systems can be represented by graph transition systems, which are basically transition systems whose states and transitions represent graphs and graph morphisms. In this paper, we propose the modeling of graph transition systems in PDDL and the application of heuristic search planning for their analysis. We consider different heuristics and present experimental results.

Introduction

Graphs are a suitable modeling formalism for software and hardware systems involving issues such as communication, object orientation, concurrency, distribution and mobility. The graphical nature of such systems appears explicitly in approaches like graph transformation systems (Rozenberg 1997) and implicitly in other modeling formalisms like algebras for communicating processes (Milner 1989). The properties of such systems mainly regard aspects such as temporal behavior and structural properties. They can be expressed, for instance, by logics used as a basis for a formal verification method, like model checking (Clarke, Grumberg, & Peled 1999), which main success is due to the ability to find and report errors.

Finding and reporting errors in model checking and many other analysis problems can be reduced to state space exploration problems. In most cases the main drawback is the *state explosion problem*. In practice, the size of state spaces can be large enough (even infinite) to exhaust the available space and time resources. Heuristic search has been proposed as a solution in many fields, including model checking (Edelkamp, Leue, & Lafuente 2003), planning (Bonet & Geffner 2001) and games (Korf 1985). Basically, the idea is to apply algorithms that exploit the information about the problem being solved in order to guide the exploration process. The benefits are twofold: the search effort is reduced, i.e., errors are found faster and by consuming less memory, and the solution quality is improved, i.e., counterexamples are shorter and thus may be more useful. In some cases, like wide area networks with Quality of Service (QoS), one might not be interested in short paths, but in cheap or optimal ones based on some notion of cost. Therefore, we generalize our approach by considering an abstract notion of costs.

Our work is mainly inspired by approaches to directed model checking (Edelkamp, Leue, & Lafuente 2003), logics for graphs (like the *monadic second order logic* (Courcelle 1997)), spatial logics used to reason about the behavior and structure of processes calculi (Caires & Cardelli 2003) and graphs (Cardelli, Gardner, & Ghelli 2002), and approaches for the analysis of graph transformation systems (Baldan *et al.* 2004; Rensink 2003; Varrò 2003). At the theoretical front, our approach is very much inspired by cost-algebraic search algorithms (Sobrinho 2002; Edelkamp, Jabbar, & Lluch-Lafuente 2005a).

The work also relates to (Edelkamp 2003a) that compiled protocol software model checking domains in Promela to PDDL. Two of such domains have served as a benchmark for the 4th international planning competition in 2004 (Hoffmann *et al.* 2005). We extend the work of (Edelkamp, Jabbar, & Lluch-Lafuente 2005b) that applies heuristic search for graph transition systems in the context of the experimental model checker HSF-SPIN (Edelkamp, Leue, & Lafuente 2003). To the best of our knowledge this is the first work on action planning for the analysis of graphically described systems, probably with the exception of one currently running master's thesis (Golkov 2005).

The goal of our approach is to formalize structural properties of systems modeled by graph transition systems. We believe that our work additionally illustrates the benefits of applying heuristic search in state space exploration systems. Heuristic search is intended to reduce the analysis effort and, in addition, to deliver shorter or optimal solutions. We consider a notion of optimality with respect to a certain cost or weight associated to system transitions. For instance, the cost of a transition in network systems can be a certain QoS value associated to the transition.

The next section introduces the running example that is used along the paper to illustrate some of the concepts and methods. Next, we define our modeling formalism, namely graph transition systems. We then consider the kind of properties we are interested in verifying and discuss their PDDL model. We study two planning heuristics for the analysis of properties in graph transition systems and present experimental results obtained with a heuristic search planner. Finally, we conclude the paper and outline future research avenues. For the sake of readability the PDDL model for the arrow distributed protocol is included in an appendix that follows the bibliography.

The Arrow Distributed Directory Protocol

The *arrow distributed directory protocol* (Demmer & Herlihy 1998) is a solution to ensure exclusive access to mobile objects in a distributed system. The distributed system is given as an undirected graph G, where vertices and edges respectively represent nodes and communication links. Costs are associated with the links in the usual way, and a mechanism for optimal routing is assumed.

The protocol works with a minimal spanning tree T of G. Each node has an arrow which, roughly speaking, indicates the *direction* in which the object lies. If a node owns the object or is requesting it, the arrow points to itself; we say that the node is *terminal*. The directed graph induced by the arrows is called $\mathcal{L}$. Roughly speaking, the protocol works by propagating requests and updating arrows such that at any moment the paths induced by arrows, called *arrow paths*, either lead to a terminal owning the object or waiting for it.

More precisely, the protocol works as follows: Initially $\mathcal{L}$ is set such that every path leads to the node owning the object. When a node u wants to acquire the object, it sends a request message *find*(u) to $a(u)$, the target of the arrow starting at u, and sets $a(u)$ to u, i.e., it becomes a terminal node. When a node u whose arrow does not point to itself receives a *find*(w) message from a node v, it forwards the message to node $a(u)$ and sets $a(u)$ to v. On the other hand, if $a(u) = u$ (the object is not necessarily at u but will be received if not) the arrows are updated as in the previous case but this time the request is not forwarded but enqueued. If a node owns the object and its queue of requests is not empty, it sends the object to the (unique) node u of its queue sending a *move*(u) message to v. This message goes *optimally* through G. A formal definition of the protocol can be found in (Demmer & Herlihy 1998).

Figure 1 illustrates three states of a protocol instance with six nodes $v_0, \ldots, v_5$. The state on the left is the initial one: node v_0 has the object and all paths induced by the arrows lead to it. The state on the right of the figure is the result of two steps: node v_4 sends a request for the object through its arrow; and v_3 processes it by updating the arrows properly, i.e., the arrow points now to v_4 instead of v_2.

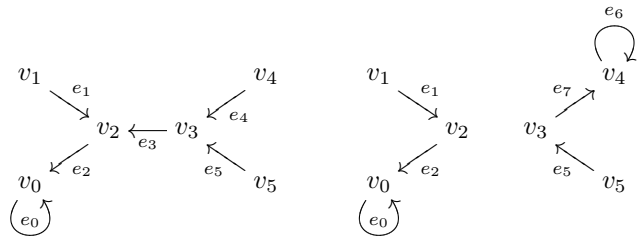

Figure 1: Three states of the directory.

One could be interested in properties like *Can node v_i be a terminal?* (Property 1), *Can node v_i be terminal and all arrow paths end at v_i?* (Property 2), *Can a node v be terminal?* (Property 3), *Can a node v be terminal and all arrow paths end at v?* (Property 4).

Graph Transition Systems

This section presents our modeling formalism. First, an algebraic notion of costs is defined. It shall be used as an abstraction of costs or weights associated to edges of graphs or transitions of transition systems. For a deeper treatment of the cost algebra we refer to (Edelkamp, Jabbar, & Lluch-Lafuente 2005a).

Definition 1 *A* cost algebra *is a 6-tuple $\langle A, \bigsqcup, \times, \preceq, \boldsymbol{0}, \boldsymbol{1}\rangle$, such that*

1. *$\langle A, \times\rangle$ is a monoid with $\boldsymbol{1}$ as identity element and $\boldsymbol{0}$ as its absorbing element, i.e., $a \times \boldsymbol{0} = \boldsymbol{0} \times a = \boldsymbol{0}$;*
2. *$\preceq \subseteq A \times A$ is a total ordering with $\boldsymbol{0} = \bigsqcap A$ and $\boldsymbol{1} = \bigsqcup A$;*
3. *A is* isotone*, i.e., $a \preceq b$ implies both $a \times c \preceq b \times c$ and $c \times a \preceq c \times b$ for all $a, b, c \in A$ (Sobrinho 2002).*

In the rest of the paper $a \prec b$ abbreviates $a \preceq b$ and $a \neq b$. Moreover, $a \succeq b$ abbreviates $b \preceq a$, and $a \succ b$ abbreviates $a \succeq b$ and $a \neq b$. The *least* element c in A is defined as $\bigsqcup A$, if $c \in S$ and $c \preceq a$ for all $a \in A$. The *greatest* element c in A is defined as $\bigsqcap A$, if $c \in A$ and $c \preceq a$ for all $a \in A$.

Intuitively, A is the domain set of cost values, $\times$ is the operation used to cumulate values and $+$ is the operation used to select the best (the least) amongst two values. Consider for example, the following instances of cost algebras, typically used as cost or QoS formalisms:

- $\langle\{true, false\}, \vee\wedge, \Rightarrow, false, true\rangle$ (Network and service availability)
- $\langle\mathbb{R}^+ \cup \{+\infty\}, \min, +, \leq, +\infty, 0\rangle$ (Price, propagation delay)
- $\langle\mathbb{R}^+ \cup \{+\infty\}, \max, \min, \geq, 0, +\infty\rangle$ (Bandwidth).

In the rest of the paper, we consider a fixed cost algebra $\langle A, \bigsqcup, \times, \preceq, \mathbf{0}, \mathbf{1}\rangle$.

Definition 2 *An (*edge-weighted graph*) G is a tuple $\langle V_G, E_G, src_G, tgt_G, \omega_G\rangle$ where V_G is a set of nodes, E_G*

is a set of edges, $src_G, tgt_G : E_G \to V_G$ are a source and target functions, and $\omega_G : E_G \to A$ is a weighting function.

Graphs usually have a distinguished start state which we denote with s_0^G, or just s_0 if G is clear from the context.

Definition 3 *A* path *in a graph G is an alternating sequence of nodes and edges represented as $u_0 \xrightarrow{e_0} u_1 \ldots$ such that for each $i \geq 0$ we have $u_i \in V_G$, $e_i \in E_G$, $src_G(e_i) = u_i$ and $tgt_G(e_i) = u_{i+1}$, or, shortly $u_i \xrightarrow{e_i} u_{i+1}$.*

An initial path is a path starting at s_0^G. Finite paths are required to end at states. The length of a finite path p is denoted by $|p|$. The concatenation of two paths p, q is denoted by pq, where we require p to be finite and end at the initial state of q. The cost of a path is the cumulative cost of its edges. Formally,

Definition 4 *Let $p = u_0 \xrightarrow{e_0} \ldots \xrightarrow{e_{k-1}} u_k$ be a finite path in a graph G. The* path cost *$\omega_G(p)$ is $\omega_G(e) \times \omega_G(q)$ if $p = (u \xrightarrow{e} v)q$ and $\mathbf{1}$ otherwise. If $|q| = 0$, $\omega_G(q) = \mathbf{1}$.*

Let $\gamma(u)$ denote the set of all paths starting at node u. In the sequel, we shall use $\omega_G^*(u, V)$ to denote the cost of the optimal path starting at a node u and reaching a node v in a set $V \subseteq V_G$. Formally, $\omega_G^*(u, V) = \bigsqcup_{p \in \gamma(s) | (p \cap V) \neq \emptyset} \omega_G(p)$. For ease of notation, we write $\omega_G^*(u, \{v\})$ as $\omega_G^*(u, v)$.

Graph transition systems are suitable representations for software and hardware systems and extend traditional transition systems by relating states with graphs and transitions with partial graph morphisms. Intuitively, a partial graph morphism associated to a transition represents the relation between the graphs associated to the source and the target state of a transition. More specifically, it models the merging, insertion, addition and renaming of graph items (nodes or edges). In case of a merge, the cost of merged edges is the least one amongst the edges involved in the merging.

Definition 5 *A* graph morphism *$\psi : G_1 \to G_2$ is a pair of mappings $\psi_V : V_{G_1} \to V_{G_2}$, $\psi_E : E_{G_1} \to E_{G_2}$ such that we have $\psi_V \circ src_{G_1} = src_{G_2} \circ \psi_E$, $\psi_V \circ tgt_{G_1} = tgt_{G_2} \circ \psi_E$,[1] and for each $e \in E_{G_2}$ we have, $\omega_{G_2}(e) = \bigsqcup\{\omega_{G_1}(e') \mid \psi_E(e') = e\}$.*
A graph morphism $\psi : G_1 \to G_2$ is called injective *if so are ψ_V and ψ_E;* identity *if both ψ_V and ψ_E are identities, and* isomorphism *if both ψ_E and ψ_V are bijective. A graph G' is a* subgraph *of graph G, if $V_{G'} \subseteq V_G$ and $E_{G'} \subseteq E_G$, and the inclusions form a graph morphism.*
A partial graph morphism *$\psi : G_1 \to G_2$ is a pair $\langle G_1', \psi_m \rangle$, where G_1' is a subgraph of G_1, and $\psi_m : G_1' \to G_2$ is a graph morphism.*

The composition of (partial) graph morphisms results in (partial) graph morphisms. Now, we define a notion of transition system that enriches the usual ones with weights.

Definition 6 *A* transition system *is a graph $M = \langle S_M, T_M, in_M, out_M, \omega_M \rangle$ whose nodes and edges are respectively called* states *and* transitions*, with in_M, out_M representing the source and target of an edge respectively.*

Finally, we are ready to define graph transition systems, which are transition systems together with morphisms mapping states into graphs and transitions into partial graph morphisms.

Definition 7 *A* graph transition system *(GTS) is a pair $\langle M, g \rangle$, where M is a weighted transition system and $g : M \to \mathcal{U}(\boldsymbol{G}_p)$ is a graph morphism from M to the graph underlying $\boldsymbol{G}_p$, the category of graphs with partial graph morphisms. Therefore $g = \langle g^S, g^T \rangle$, and the component on states g^S maps each state $s \in S_M$ to a graph $g^S(s)$, while the component on transitions g^T maps each transitions $t \in T_M$ to a partial graph morphism $g^T(t) : g^S(in_M(t)) \Rightarrow g^S(out_M(t))$.*

In the rest of the paper we shall consider a GTS $\langle M, g \rangle$ modeling the state space of our running example, where g maps states to $\mathcal{L}$, i.e., the graph induced by the arrows, and transitions to the corresponding partial graph morphisms. Consider Figure 1, each of the three graphs depicted, say G_1, G_2 and G_3 corresponds to three states s_1,s_2,s_3, meaning that $g(s_1) = G_1$, $g(s_2) = G_2$ and $g(s_3) = G_3$. The figure illustrates a path $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3$, where $g(t_1)$ is the identity restricted to all items but edge e_4. Similarly, $g(t_2)$ is the identity restricted to all items but edge e_3. Thus, in both transitions all other items are preserved (with their identity) except the edges mentioned.

Properties of Graph Transition Systems

The properties of a graph transition system can be expressed using different formalisms. One can use, for instance, a temporal graph logic like the ones proposed in (Baldan *et al.* 2004; Rensink 2003), which combine temporal and graph logics. A similar alternative are spatial logics (Caires & Cardelli 2003), which combine temporal and structural aspects. In graph transformation systems (Corradini *et al.* 1997), one can use rules to find certain graphs: the goal might be to find a match for a certain transformation rule. For the sake of simplicity and generality, however, we consider that the problem of satisfying or falsifying a property is reduced to the problem of finding a set of *goal states* characterized by a goal graph and the existence of an injective morphism.

Definition 8 *Given a GTS $\langle M, g \rangle$ and a graph G, the* goal function *$\mathrm{goal}_G : S_M \to \{\mathrm{true}, \mathrm{false}\}$ is defined such that $\mathrm{goal}_G(s) = \mathrm{true}$ iff there is an injective graph morphism $\psi : G \to g(s)$.*

Intuitively, *$goal_G$* maps a state s to *true* if and only if G can be injectively matched with a subgraph of $g(s)$.

[1] $\circ$ is the function composition operator. In other words $f \circ g = f(g(\cdot))$

Figure 2: Three graphs illustrating various goal criteria.

It is worth saying that most graph transformations approaches consider injective rules, for which a match is precisely given by injective graph morphisms, and that the most prominent graph logic, namely the Monadic Second-Order (MSO) logic by (Courcelle 1997) and its first-order fragment (FO) can be used to express injective graph morphisms. The graph G will be called *goal graph*. It is of practical interest identifying particular cases of goal functions as the following goal types:

1. ψ is an identity - the exact graph G is looked for. In our running example, this corresponds to Property 2 mentioned in Section . For instance, we look for the exact graph depicted in left of Figure 2.
2. ψ is a restricted identity - an exact subgraph of G is looked for. This is precisely Property 1. For instance, we look for a subgraph of the graph depicted in left of Figure 2. The graph in center of Figure 2 satisfies this.
3. ψ is an isomorphism - a graph isomorphic to G is looked for. This is precisely Property 4. For instance, we look for a graph isomorphic to the one depicted in left of Figure 2. The graph in the right of Figure 2 satisfies this.
4. ψ is any injective graph morphism - we have the general case. This is precisely Property 3. For instance, we look for an injective match of the graph depicted in center of Figure 2. The graph in the right of Figure 2 satisfies this.

Note that there is a type hierarchy, since goal type 1 is a subtype of goal types 2 and 4, which are of course subtypes of the most general goal type 4.

The computational complexity of the goal function varies according to the above cases. For goals of type 1 and 2, the computational efforts needed are just $O(|G|)$ and $O(|\psi(G)|)$, respectively. Unfortunately, for goal types 3 and 4, due to the search for isomorphisms, the complexity increase to a term exponential in $|G|$ for the graph isomorphism case and to a term exponential in $|\psi(G)|$ for the subgraph isomorphism case. The general problem of subgraph isomorphism (SI) can be reduced polynomially to graph isomorphism. Subgraph isomorphism is NP-complete, as CLIQUE $\leq_p$ SI. The general problem of graph isomorphism is not completely classified. It is expected not to be NP-complete (Wegener 2003).

Now we state the two analysis problems we consider. The first one consists on finding a goal state.

Definition 9 *Given a GTS $\langle M, g\rangle$ and a graph G (the goal graph), the* reachability problem *of our approach consists on finding a state $s \in S_M$ such that* goal(s) *is* true.

The second problem aims at finding an optimal path to a goal state.

Definition 10 *Given a GTS (M, g) and a graph G (the goal graph), the* optimality problem *of our approach consists on finding a finite initial path p ending at a state $s \in S_M$ such that such that* $\text{goal}_G(s)$ *is* true *and* $\omega(p) = \omega^*_M(s_0^M, S')$*, where* $S' = \{s \in S_M \mid \text{goal}_G(s) = \text{true}\}$.

For the sake of brevity, in the following $\omega^*_M(s)$ abbreviates $\omega^*_M(s, S')$ with $S' = \{s \in S_M \mid goal_G(s) = true\}$, when $goal_G$ is clear from the context.

The two problems defined in the previous section can be solved with traditional graph exploration and shortest-path algorithms[2]. For the reachability problem, for instance, one can use, amongst others, depth-first search, hill climbing, best-first search, Dijkstra's algorithm (and its simplest version breadth-first search) or A*. For the optimality problem, only the last two are suited.

Nevertheless, Dijkstra's algorithm and A* are traditionally defined over a simple instance of our cost algebra A, namely algebra $\langle \mathbb{R}^+ \cup \{+\infty\}, \min, +, \leq, +\infty, 0\rangle$. Fortunately, the results that ensure the *admissibility* of Dijkstra's algorithm or A*, i.e., the fact that both algorithms correctly solve the optimality problem, have been generalized for the cost algebra (Edelkamp, Jabbar, & Lluch-Lafuente 2005a).

Encoding of the Arrow Distributed Directory Protocol

To simplify the discussion, we assume a uniform transition weight leading to pure propositional planning problems. But with the extensions that are available in current planning description languages such as PDDL2.1 (Fox & Long 2003), the current setting can be extended to numerical weights. Note that, the formal treatment of the problem presented earlier in this paper is capable of dealing with non-uniform weights.

In propositional planning, for each state we have atomic propositions that can either be true or false. Planning operators or actions change the truth values of atomic propositions AP. An action a in STRIPS consists of three lists: precondition, add, and delete lists, commonly denoted as $pre(a)$, $add(a)$, and $del(a)$, respectively (Fikes & Nilsson 1971). Each list consists of atomic propositions and the application of a to a state $S \subseteq 2^{AP}$ with $pre(a) \subseteq S$ yields the successor state $(S \setminus del(a)) \cup add(a)$.

To apply a planner to graph transition systems, we first need a propositional description of graph transition systems in PDDL. The graph is modeled with the help

[2]We refer here to a slight modification of the original algorithms, consisting of terminating the algorithm when a goal state is reached and returning the corresponding path.

of predicates defining the edges. We use `(link u v)` predicate to denote an edge between two nodes u and v. Since a node cannot exist on its own, we do not provide any predicate to declare a node. The predicate `(find-pending u v w)` is true if the node u receives a request from its neighbour v to find the object for the node w. Similarly, the predicate `(move-pending u v w)` is true, if the node u receives an object from the node w to be forwarded to the requesting node. The parameter v is actually not in use and its just for the sake of uniformity with the original model and with the `find-pending` predicate.

The predicate `(not-request-send u)` is used to control the requests generated by the nodes so that a node cannot request more than once. The contents of a queue attached with a particular node u are controlled through the `(queue u v)` predicate. The ownership of the object is determined through the `(owner u)` predicate.

Due to the parametric description facility provided by planning formalism, it is easier to define morphisms and partial morphisms as actions. For example, a morphism operation that inverses an edge can easily be defined as a very simple action as follows:

```
(:action morphism-inverse
:parameters(?u ?v - node)
:precondition
  (link ?u ?v)
:effect
  (and
    (not (link ?u ?v))
    (link ?v ?u)))
```

An example description for the Arrow Protocol is provided in Annex.

Problem description in PDDL

A GTS problem can be described with the help of predicates defining the graph in the initial state. The whole graph can be described by the use of `link` predicates defining the edges between different nodes of the graph. The owner node, i.e., the node that currently owns the object is define by the use of `owner` predicate.

A PDDL problem description for an instance of *star*-shaped network topology is shown in the appendix.

Goal Specification in PDDL

Fortunately, PDDL provides a very neat and elegant mechanism to formulate our goals' criteria. In the following we explain various methods to describe different types of goals.

Property 1 goal (subgraph): Perhaps the most simple to describe are the type 1 goals as we only search for a specific subgraph. As is evident from the PDDL specification of the domain, the subgraph can easily be declared by using the `(link u v)` predicates. If the subgraph to be searched for actually asks for an ownership predicate to be true for some node w, we simply declare the `(owner w)` predicate as our goal criteria. In Appendix, we see an example problem description in PDDL where a goal of type 1 is searched for.

Property 2 goal (exact graph): For a *Property 2* goal, we look for an exact matching of the goal graph in our state space. Just like for the previous type, we can describe the whole graph with `(link u v)` predicates. Note that it is true only for the current domain, since a spanning tree property of the graphs is preserved through out the search space, i.e., there cannot be a reachable state where the graph is a superset of the goal graph. This might not be the case in other GTS domains. In such cases we have to describe the non-existence of all the other edges too.

Property 3 goal (subgraph isomorphism): Given a goal graph G, the state space is searched for a state that contains a subgraph isomorphic to G. In such case goals are strictly more expressive and need an existential quantification over all the nodes to be described succinctly. Existential quantification can be incorporated in STRIPS through ADL (Pednault 1989) by the following construct:

```
(:goal <existential-expression> <goal-condition>)
```

A goal of type 3 can then be included in our problem specification as:

```
(:goal (exists (?n - node) (owner ?n))
```

Property 4 goal (isomorphism): Given a goal graph G, the state space is searched for a node that contains a graph isomorphic to G. Having the existential quantifier in our hands, we can describe G using `(link u v)` predicates. For our example in Figure 2, a type 4 goal will have the form:

```
(:goal (exists ?v0 ?v1 ?v2 ?v3 ?v4 ?v5 - node)
 (and (link ?v0 ?v0)  (link ?v1 ?v0)
      (link ?v2 ?v0)  (link ?v3 ?v1)
      (link ?v4 ?v0)  (link ?v5 ?v4)
      (owner ?v3)))
```

Given actions with ADL expressivity, it is not difficult to transform an existential goal description to a non-existential one by adding the following special operator to the domain description:

```
(:action goal-achieving-action
 :precondition <old-goal-condition>
 :effects (and (goal-achieved)))
```

The modified goal condition then simplifies to

```
(:goal (goal-achieved))
```

With the extended expressivity of PDDL2.2 (Edelkamp & Hoffmann 2004) goal achievement is best introduced in form of domain axioms, so-called derived predicates. They are inferred in form of a fix-point computation with rules that do not belong to a plan. For this case we include

```
(:derived (goal-achieved) <old-goal-condition>)
```

to the domain description.

Planning Heuristics for Graph Transition Systems

Heuristic search algorithms use heuristic functions to guide the state space exploration as apposed to blind search algorithms that do not utilize any information about the search space. Two of the most famous heuristic search algorithms are A* and IDA*. A* utilizes a heuristic estimate for the distance from a state to the goal, to prioritize states' expansion. The result is a reduced search space; consequently, less consumption of memory with gain in speed. A* is guaranteed to produce optimal results in case of admissible and consistent heuristic.

Most of the modern planners (for example, FF (Hoffmann & Nebel 2001) or MIPS (Edelkamp 2003b)) utilize various heuristics to guide the planner. Two of such heuristics that have performed very good in planning domains are *relaxed planning heuristic* and *planning pattern databases*.

Relaxed Planning Heuristic

A *relaxed planning heuristic* (Hoffmann & Nebel 2001) is computed by solving a *relaxed version* of a planing problem. The *relaxation* a^+ of a STRIPS action $a = (pre(a), add(a), del(a))$ is defined as $a^+ = (pre(a), add(a), \emptyset)$. The relaxation of a planning problem is the one in which all actions are substituted by their relaxed counterparts. Any solution that solves the original plan also solves the relaxed one; and all preconditions and goals can be achieved if and only if they can be in the relaxed task. Value h^+ is defined as the length of the shortest plan that solves the relaxed problem.

Solving relaxed plans optimally is still computationally hard (Bylander 1994), but the decision problem to determine, if a relaxed planning problem has at least one solution, is computationally tractable. The optimization task can efficiently be approximated by counting the number of operators in a *parallel plan* that solves the relaxed problem. Note that optimal parallel and optimal sequential plans may have a different sets of operators, but good parallel plans are at least informative for sequential plan solving, and can, therefore, be used for the design of a heuristic estimator function.

The extension to the *numerical relaxed planning heuristic* is a polynomial-time state evaluation function for mixed integer domain-independent planning problems (Hoffmann 2003). It has been extended to nonlinear tasks (Edelkamp 2004).

Planning Pattern Databases

Abstraction is one of the most important issues to cope with large and infinite state spaces, and to reduce the exploration efforts. Abstracted systems should be significantly smaller than the original one while preserving some properties of concrete systems. The study of abstraction formalisms for graph transition systems is, however, out of the scope of this paper. We refer to (Baldan *et al.* 2004) for an example of such a formalism. Assuming that abstractions are available, we state the properties necessary for abstractions to preserve our two problems (reachability and optimization) and propose how to use abstraction to define informed heuristics.

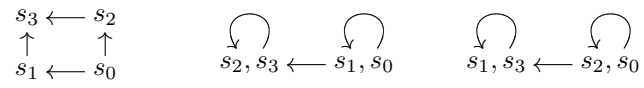

Figure 3: A transition system (leftmost) with two different abstractions.

The preservation of the reachability problem means that the existence of an initial goal path in the concrete system must entail the existence of a corresponding initial goal path in the abstract system. Note that this does not mean the existence of *spurious* initial goal paths in the abstract system, i.e., abstract paths that do not correspond to any concrete path. Similarly, the preservation of the optimization problem means that the cost of the optimal initial goal path in the concrete system should be greater or equal to the cost of the optimal initial goal path in the abstract system.

Abstractions have been applied in combination with heuristic search in single-agent games (Culberson & Schaeffer 1998; Korf 1997), in model checking (Edelkamp & Lluch-Lafuente 2004) and planning (Edelkamp 2001) approaches. The main idea is that the abstract system is explored in order to create a database that stores the exact distances from abstract states to the set of abstract goal states. The exact distance between abstract states is an admissible and consistent estimate of the distance between the corresponding concrete states. The distance database is thus used as heuristics for analyzing the concrete system.

When different abstractions are avaliable, we can combine the different databases in various ways to obtain better heuristics. The first way is to trivially select the best value delivered by two heuristic databases, which trivially results in a consistent and admissible heuristic. Figure 3 depicts a concrete transition system (left) with three abstractions (given by node mergings). The two abstractions are mutually disjoint.

Experimental Results

We validate our approach by presenting initial experimental results obtained with the heuristic search planning system FF. We have implemented the *arrow distributed directory protocol* in PDDL2.1, Level 1, i.e. in the specification language STRIPS/ADL. We performed our experiments on a Pentium IV 3.2 GHz. machine with Linux operating system and 2 gigabytes of internal memory. In all our experiments we set a memory bound of 2 GB.

When running the planner on the instances, we obtain the results as shown in Table 1 in comparison with

	HSF-SPIN		FF
star	DFS	BFS_{h_f}	$EHC + RPH$
Stored nodes	6,253	30	6
Sol. length	134	58	5
chain	DFS	BFS_{h_f}	$EHC + RPH$
Stored nodes	78,112	38	6
Sol. length	118	74	5
tree	DFS	BFS_{h_f}	$EHC + RPH$
Stored nodes	24,875	34	6
Sol. length	126	66	5

Table 1: Comparison of results between HSF-SPIN and FF.

	# Nodes	*Stored Nodes*	*Sol. Length*
star	10	6	5
	25	7	6
	50	7	6
	70	7	6
chain	10	6	5
	25	33	28
	50	100	73
	70	138	101
tree	10	6	5
	25	22	16
	50	47	25
	70	61	31

Table 2: Scaling behaviour of the model.

the results that we have obtained in the model checking domain through our experimental model checker HSF-SPIN. The goal searched for is of type 2. Column DFS shows the results while running HSF-SPIN with depth-first search as the exploration algorithm. The gain in HSF-SPIN by employing a heuristic guided exploration as apposed to DFS is noticeable in column BFS_{h_f}. The heuristic estimate used here is based on original *formula-based heuristic* (Edelkamp, Leue, & Lafuente 2003) that exploits the length of the specification of goal states to guide the search algorithm. A discussion on this heuristic is out of the scope of this paper and we refer the reader to (Edelkamp, Leue, & Lafuente 2003) for a detailed treatment.

For all three topologies, namely, *star*, *tree*, and *chain*, the planner resulted in much lesser expansions of nodes. Note that, though the results through the use of planner seem by far better than the one by model checker, we cannot actually compare the two approaches with each other for several reasons. A crucial difference is the dynamic creation of nodes during exploration. PDDL specifications currently do not support such kind of dynamism in models. For a limited case, we can utilize the visibility paradigm of domain specification by providing a pool of invisible nodes to the planner along with the model. These nodes can be made visible whenever a new node is required to be created. The other crucial difference is the modeling of finite and bounded channels - one of the main component of a concurrent system. Such channels can be defined in a model checker but not in PDDL.

In Table 2, we depict the scaling behaviour of the problem for different topologies. We generated random graphs with random `owner`s and with random goals. The second column shows the number of nodes that composed the graph. Column 3, *Stored Nodes*, shows the number of nodes stored during the serach. The length of the solution obtained is shown in the fourth column. For *star* topology, the problem was quite simple. But a major shift in space and time requirement was noted when we switched to the *chain* topology. The longest running example in *chain* topology was with 70 nodes that took about 139 secs to be solved. The scaling factor of memory usage turned out to be very sharp. A 50 nodes problem required about 0.5 GB that jumped to 1.9 GB for 70 nodes in all the topologies. Unfortunately this was also the capacity of our machine - the reason that we are unable to show the results for bigger models.

Conclusion

We have presented an abstract approach for the analysis of graph transitions systems, which are traditional transition systems where states and transitions respectively represent graphs and partial graph morphisms. It is a useful formalism to represent the state space of systems involving graphs, like communication protocols, graph transformations, and visually described systems.

The analysis of such systems is reduced to exploration problems consisting on finding certain states reachable from the initial one. We analyze two problems: finding just one path and finding the optimal one, according to a certain notion of optimality. As specification formalism, we propose the use of ADL. It is capable of expressing all four types of goals that we have suggested. In addition, we have proposed the use of abstraction-based heuristics which exploit abstraction techniques in order to obtain informed heuristics.

We have illustrated our approach with a scenario in which one is interested in analyzing structural properties of communication protocols. As a concrete example we used the *arrow distributed directory protocol* (Demmer & Herlihy 1998) which ensures exclusive access to a mobile service in a distributed system. We implemented our approach in a heuristic search planning system, and presented experiments validating our approach. The PDDL specification presented in this paper is one of the first steps towards modeling *arrow distributed directory protocol* and still has some of the specifications unmodeled such as a bounded queue to prioritize the requests.

In the 2004 International Planning Competition (IPC-4)[3], a Promela domain was used for the first time

[3] `http://ipc.icaps-conference.org/`

as an AI planning problem. This opened new horizons to bridge model checking with AI planning. This paper is one of the first efforts to model the systems represented by Graph Transition Systems as an AI planning problem. There is still a lot of room for expansion for the ideas presented in this paper.

In future work we would like to investigate further scenarios for the analysis of graph transformation systems to planning problems. One such direction is to model other more complicated protocols than the Arrow Distributed Directory protocol. With more challenging problem instances, we expect that graph transition system can serve as a challenging benchmark for upcoming planning competitions.

References

Baldan, P.; Corradini, A.; König, B.; and König, B. 2004. Verifying a behavioural logic for graph transformation systems. In *CoMeta'03*, ENTCS.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1–2):5–33.

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artificial Intelligence* 165–204.

Caires, L., and Cardelli, L. 2003. A spatial logic for concurrency (part I). *Inf. Comput.* 186(2):194–235.

Cardelli, L.; Gardner, P.; and Ghelli, G. 2002. A spatial logic for querying graphs. In *ICALP'2002*, volume 2380 of *Lecture Notes in Computer Science*, 597–610. Springer.

Clarke, E.; Grumberg, O.; and Peled, D. 1999. *Model Checking*. The MIT Press.

Corradini, A.; Montanari, U.; Rossi, F.; Ehrig, H.; Heckel, R.; and Löwe, M. 1997. *Algebraic approaches to graph transformation*, volume 1. World Scientific. chapter Basic concepts and double push-out approach.

Courcelle, B. 1997. *Handbook of graph grammars and computing by graph transformations*, volume 1 : Foundations. World Scientific. chapter 5: The expression of graph properties and graph transformations in monadic second-order logic, 313–400.

Culberson, J. C., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(4):318–334.

Demmer, M. J., and Herlihy, M. 1998. The arrow distributed directory protocol. In *International Symposium on Distributed Computing (DISC 98)*, 119 –133.

Edelkamp, S., and Hoffmann, J. 2004. PDDL2.2: The language for the classical part of the 4th international planning competition. Technical Report 195, University of Freiburg.

Edelkamp, S., and Lluch-Lafuente, A. 2004. Abstraction databases in theory and model checking practice. In *ICAPS Workshop on Connecting Planning Theory with Practice*.

Edelkamp, S.; Jabbar, S.; and Lluch-Lafuente, A. 2005a. Cost-algebraic heuristic search. In *Proceedings of Nineteenth National Conference on Artificial Intelligence (AAAI'05)*. To appear.

Edelkamp, S.; Jabbar, S.; and Lluch-Lafuente, A. 2005b. Heuristic search for the analysis of graph transition systems. draft.

Edelkamp, S.; Leue, S.; and Lafuente, A. L. 2003. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer (STTT)* 5(2-3):247–267.

Edelkamp, S. 2001. Planning with pattern databases. In *European Conference on Planning (ECP)*, Lecture Notes in Computer Science. Springer. 13-24.

Edelkamp, S. 2003a. Promela planning. In *Model Checking Software (SPIN)*, 197–212.

Edelkamp, S. 2003b. Taming numbers and durations in the model checking integrated planning system. *Journal of Artificial Research (JAIR)* 20:195–238.

Edelkamp, S. 2004. Generalizing the relaxed planning heuristic to non-linear tasks. In *German Conference on Artificial Intelligence (KI)*. 198–212.

Fikes, R., and Nilsson, N. 1971. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Research*. Special issue on the 3rd International Planning Competition.

Golkov, E. 2005. Graphtransformation als Planungsproblem. Master's thesis, University of Dortmnund. draft.

Hoffmann, J., and Nebel, B. 2001. Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Hoffmann, J.; Englert, R.; Liporace, F.; Thiebaux, S.; and Trüg, S. 2005. Towards realistic benchmarks for planning: the domains used in the classical part of IPC-4. Submitted.

Hoffmann, J. 2003. The Metric FF planning system: Translating "Ignoring the delete list" to numerical state variables. *Journal of Artificial Intelligence Research* 20:291–341.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97–109.

Korf, R. E. 1997. Finding optimal solutions to Rubik's Cube using pattern databases. In *National Conference on Artificial Intelligence (AAAI)*, 700–705.

Milner, R. 1989. *Communication and Concurrency*. Prentice Hall.

Pednault, E. P. D. 1989. ADL: Exploring the middleground between STRIPS and situation calculus. In *Knowledge Representation and Reasoning (KR)*, 324–332. Morgan Kaufman.

Rensink, A. 2003. Towards model checking graph grammars. In *3rd Workshop on Automated Verification of Critical Systems*, Tech. Report DSSE-TR-2003, 150–160.

Rozenberg, G., ed. 1997. *Handbook of graph grammars and computing by graph transformations*. World Scientific.

Sobrinho, J. 2002. Algebra and algorithms for QoS path computation and hop-by-hop routing in the internet. *IEEE/ACM Trans. Netw.* 10(4):541–550.

Varrò, D. 2003. Automated formal verification of visual modeling languages by model checking. *Journal on Software and Systems Modeling*.

Wegener, I. 2003. *Komplexitätstheorie*. Springer. (in German).

Appendix

PDDL model of the arrow distributed directory protocol as described in *The Arrow Distributed Directory Protocol* by M. J. Demmer and M. P. Herlihy.

Domain Description

```
(define (domain arrow-domain)
(:requirements :typing :strips)
(:types node - object)
(:predicates
  (link ?n1 ?n2 - node)
  (queue ?n1 ?n2 - node)
  (owner ?n1 - node)
  (not-request-send ?n1 - node)
  (find-pending ?n1 ?n2 ?n3 - node)
  (move-pending ?n1 ?n2 ?n3 - node))

(:action request-object
:parameters (?u - node)
:precondition
  (and (not-request-send ?u))
:effect
  (and
     (not (not-request-send ?u))
     (find-pending ?u ?u ?u)))

(:action accept-request
:parameters (?u ?v ?w ?z - node)
:precondition
   (and
     (link ?u ?z)
     (not (= ?u ?z))
     (find-pending ?u ?w ?v))
:effect
   (and
     (not (find-pending ?u ?w ?v))
     (find-pending ?z ?w ?u)
     (link ?u ?v)
     (not (link ?u ?z))))

(:action accept-request
:parameters (?u ?v ?w - node)
:precondition
   (and
     (link ?u ?u)
     (find-pending ?u ?w ?v))
:effect
   (and
     (not (find-pending ?u ?w ?v))
     (link ?u ?v)
     (not (link ?u ?u))
     (queue ?u ?w)))

(:action satisfy-request
:parameters (?u ?x - node)
:precondition
  (and
    (owner ?u)
    (queue ?u ?x))
:effect
  (and
    (move-pending ?x ?x ?u)
    (not (owner ?u))
    (not (queue ?u ?x))))

(:action receive-object
:parameters (?u ?w ?v - node)
:precondition
  (and (move-pending ?u ?w ?v))
:effect
  (and
    (not (move-pending ?u ?w ?v))
    (owner ?u)))
```

Problem Instance

```
(define (problem tree)
(:domain arrow-domain)
(:objects v0 v1 v2 v3 v4 v5 - node)
(:init
  (link v0 v0)  (link v1 v0)
  (link v2 v0)  (link v3 v0)
  (link v4 v0)  (link v5 v0)
  (owner v0))

(:goal (owner v1)))
```