Acta Cybernetica **15** (2002) 601–620.

Resource-Conscious AI Planning with Conjunctions and Disjunctions*

Peep Küngas†

Abstract

The aim of this work is to develop a resource-conscious Artificial Intelligence (AI) planning system, which allows for nondeterminism in the environment. Such planner has a potential in applications where actions in "real world" are considered.

The planning process is based on proof search for a fragment of Linear Logic (LL) [10] sequents using a subset of LL rules. As LL is resource-conscious and has additive disjunction connective (represents nondeterminism), LL sequents are used to describe an application domain, whereby every LL sequent represents pre- and postconditions of a particular action execution.

We present an idea to use Petri net reachability tree analysis for finding proofs for propositional LL sequents. Game playing is used to solve LL additive disjunctions. From LL proofs plans are extracted which because of underlying LL properties keep track of resources and handle both deterministic and nondeterministic actions.

Keywords: Linear Logic Theorem Proving, AI Planning, Petri Nets, Game Playing.

1 Introduction

In mission critical situations the response of a system must be reactive. For a system relying on a symbolic representation it means that a backup plan, describing alternative actions to be carried out if the main plan is not applicable anymore, should be available immediately. Therefore nondeterminism in results of actions should be taken into account already in the planning phase and action representation must thus support describing nondeterministic actions.

The aim of this work is to develop a resource-conscious AI planning system, which is able to manage nondeterminism in an environment. Such planner has a potential in applications where actions in the "real world" are considered.

*This work was partially supported by the Estonian Science Foundation under grant no. 4155.

†Software Department, Institute of Cybernetics at Tallinn Technical University, Akadeemia tee 21, 12618 Tallinn, Estonia, e-mail: peep@cs.ioc.ee.

There exists a framework called Linear Logic [10] (LL) which allows handling uncertainties while being also resource-conscious.

As LL is resource-conscious, we are using LL formulae to describe application domain and LL theorem proving to construct a plan achieving a goal. We propose a subset of LL connectives and operators sufficient for describing both deterministic and nondeterministic actions in a resource-sensitive world.

For proving propositional LL sequents, we present a new approach by composing Petri net reachability tree analysis and game playing. Plans are then extracted from these proofs. Some extensions and improvements to our algorithm are described in [19], where also the algorithm is compared to other systems and algorithms in the AI planning field.

2 Linear Logic

LL is a refinement of classical logic introduced by J.-Y. Girard to provide a means for keeping track of "resources"—two assumptions of propositional constant A are distinguished from a single assumption of A. Although LL is not the first attempt to develop resource-oriented logics (well-known examples are relevance logic and Lambek calculus), it is by now the most investigated one.

Since its introduction LL has enjoyed increasing attention both from proof theorists and computer scientists. Therefore, because of its maturity, LL is useful as formal representation of planning system kernel. Good tutorials to LL are [32] and [21]. One of the first overviews of LL applications is presented in [1]. There exist several efficient formal method tools for proving LL sequents [31].

From the complete set of LL connectives and operators we are using multiplicative conjunction ($\otimes$), additive disjunction ($\oplus$) and "of-course" (!). Whilst the connectives $\otimes$ and $\oplus$ are needed to describe pre- and postconditions of actions, the operator ! gives us control over resources.

In terms of resource acquisition the logical expression $A \otimes B \vdash C \otimes D$ means that the resources C and D are obtainable only if both A and B are obtainable. Thus the connective $\otimes$ defines deterministic relations between resources and actions.

The expression $A \vdash B \oplus C$ on the contrary means that if we have a resource A, we can obtain either a B or a C, but we do not know which one of those. It is definitely clear that $\oplus$ is suitable to represent nondeterminism in results of actions.

The operator ! means that we can use or generate particular resource as much as we want—the resource is somehow unlimited for us.

To illustrate the above let us consider the following LL formula, adapted to our set of LL connectives and operators, from [21]—$(D \otimes D \otimes D \otimes D \otimes D) \vdash (H \otimes C \otimes !F \otimes (P \oplus I))$, which encodes a fixed price menu in a fastfood restaurant: for 5 dollars (D) you can get an hamburger (H), a coke (C), all the french fries (F) you can eat plus a pie (P) or an ice cream (I) depending on availability.

To increase the expressiveness of formulae, we are using the $a^n = \underbrace{a \otimes \ldots \otimes a}_{n}$, for $n \geq 0$, with the degenerate case $a^0 = 1$.

Since we do not use all LL connectives and operators for planning, only a subset of LL rules listed below is needed for proof search.

Logical axiom and Cut rule:

$$A \vdash A \quad (Axiom) \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma^{'}, A \vdash \Delta^{'}}{\Gamma, \Gamma^{'} \vdash \Delta, \Delta^{'}} \ (Cut)$$

Rules for the propositional constants:

$$\vdash 1 \qquad \frac{\Gamma \vdash A}{\Gamma, 1 \vdash A}$$

$$\frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \oplus B \vdash \Delta} \ (L\oplus) \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \oplus B, \Delta} \ (R\oplus)(a) \qquad \frac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \oplus B, \Delta} \ (R\oplus)(b)$$

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \otimes B \vdash \Delta} \ (L\otimes) \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma^{'} \vdash B, \Delta^{'}}{\Gamma, \Gamma^{'} \vdash A \otimes B, \Delta, \Delta^{'}} \ (R\otimes)$$

Rules for the exponential !:

$$\frac{\Gamma \vdash \Delta}{\Gamma, !A \vdash \Delta} \ (W!) \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma, !A \vdash \Delta} \ (L!) \qquad \frac{!\Gamma \vdash A, ?\Delta}{!\Gamma, \vdash !A, ?\Delta} \ (R!) \qquad \frac{\Gamma, !A, !A \vdash \Delta}{\Gamma, !A \vdash \Delta} \ (C!)$$

3 Planning and LL

One merit of LL deductive planning is said [11] to consist in its ability to solve the technical frame problem [24] without the need to state frame axioms explicitly and is therefore especially good for representing causal relations between actions and resources.

The multiplicative conjunction connective ($\otimes$) and additive disjunction ($\oplus$) have been used in [23], where a demonstration of robot planning system has been given. The usage of ? and !, whose importance to AI planning is emphasised [4], is discussed there, but not demonstrated.

Influenced by [23], LL theorem proving has been used by Jacopin [14] as an AI planning kernel. As only the multiplicative conjunction $\otimes$ is used in formulae there, the problem representation is equivalent to presentation in STRIPS [9]-like planners—the left side of a LL sequent represents STRIPS *delete*-list and the right side accordingly *add*-list. Multiplicative conjunctions just separate propositions.

Unfortunately, the algorithm Jacopin proposes for proof search is very inefficient and belongs to the class of brute force methods.

In [6] a formalism has been given for deductively generating recursive plans in Linear Logic. This advancement is a step further to more general plans, which are capable of solving instead of a single problem a class of problems.

To illustrate the usefulness of LL in resource-aware planning we give proofs for the two following tasks. After that, according to the LL rules and axioms used, plans from the proofs are extracted. Application domains are represented as sets of extralogical axioms.

It should be mentioned that as the LL planning idea consists finding a proof for a certain LL sequent, the goal for the planner is specified with a LL sequent: *initial conditions* $\vdash$ *final conditions*. In order to implement plan reuse, every proved sequent (theorem) can be added to the set of extralogical axioms and other previously proved sequents, describing a particular application domain. Thus the next time it can be used to prove other sequents.

The LL sequents we are going to use for describing application domains and goals are given in form $D \vdash D$ whereas $D ::= K \mid K \oplus D$, $K ::= 1 \mid A \mid K \otimes A$ and A is an atomic formula.

The first task for a robot is to mine a ton of gold and the second is to fill a box with balls.

3.1 Gold-mining problem

Let us assume that in some particular case the application domain consists of three actions: Excavate, Refine and Convert. We have also two predicates—KG and T for inspecting whether we have a kilogram or a ton of gold respectively. At last two constants—*Sand* and *Gold*—are used to indicate the resources we are mining.

The STRIPS-like *add*- and *delete*-effects for the just mentioned actions are specified with LL sequents as follows:
Excavate: $\vdash KG(Sand)$
Refine: $!KG(Sand) \vdash KG(Gold)$
Convert: $KG(Gold)^{1000} \vdash T(Gold)$

It should be reminded that the *delete*-effect of an action is defined before the $\vdash$ symbol and *add*-effect after that symbol.

The proof for the sequent $\vdash T(Gold)$ follows here[1]:

$$\dfrac{\dfrac{\dfrac{\vdash KG(Sand)}{\vdash !KG(Sand)}\,(R!) \qquad !KG(Sand) \vdash KG(Gold)}{\vdash KG(Gold)}\,(Cut)}{\vdash KG(Gold)^{1000}}\,999 \times (R\otimes) \qquad KG(Gold)^{1000} \vdash T(Gold)}{\vdash T(Gold)}\,(Cut)$$

The plan $\mathcal{P} = \{1000 * \{n * \mathbf{Excavate}, \mathbf{Refine}\}, \mathbf{Convert}\}$ extracted from the previous proof could be stated in the natural language as: excavate sand by one kilogram and extract gold from it until you have a kilogram of gold. Repeat in such a way thousand times.

[1] Literally speaking—how to get a ton of gold from nothing.

3.2 Box filling problem

In the second example task a robot has to fill a box with balls. The robot can move around and pick up balls it encounters on its way. Every ball is then stored in a box. The box is said to be full if it holds seven balls. The robot starts moving with empty hands (propositional constant $EMPTY$). The goal in LL sequent form is $EMPTY \vdash EMPTY \otimes FULL$.

The expression system consists of five propositions. The specification of actions available for the second task is listed here:

Take: $EMPTY \otimes NEAR \vdash HOLD$
Store: $HOLD \vdash EMPTY \otimes IN$
Move: $\vdash NEAR$
Fill: $IN^7 \vdash FULL$

First we prove that the sequent $EMPTY \vdash EMPTY \otimes IN$ is derivable:

$$\dfrac{\dfrac{\dfrac{EMPTY \vdash EMPTY \quad \vdash NEAR}{EMPTY \vdash EMPTY \otimes NEAR}\,(R\otimes) \qquad EMPTY \otimes NEAR \vdash HOLD}{EMPTY \vdash HOLD}\,(Cut) \qquad HOLD \vdash EMPTY \otimes IN}{EMPTY \vdash EMPTY \otimes IN}\,(Cut)$$

The plan extracted from the proof is $\mathcal{P}_1 = \{\mathbf{Move}, \mathbf{Take}, \mathbf{Store}\}$. The preceding proof was used to construct a plan for finding and storing a ball. The following generates a plan to fill a box with balls found:

$$\dfrac{\dfrac{EMPTY \vdash EMPTY \otimes IN \qquad \dfrac{\dfrac{\dfrac{EMPTY \vdash EMPTY \quad IN \vdash IN}{\vdots}}{EMPTY \vdash EMPTY \otimes IN^6 \quad IN \vdash IN}\,R\otimes}{EMPTY, IN \vdash EMPTY \otimes IN^7}}{EMPTY \otimes IN \vdash EMPTY \otimes IN^7}\,(L\otimes)}{EMPTY \vdash EMPTY \otimes IN^7}\,(Cut) \qquad \dfrac{\dfrac{\dfrac{EMPTY \vdash EMPTY \quad IN^7 \vdash FULL}{EMPTY, IN^7 \vdash EMPTY \otimes FULL}\,(R\otimes)}{EMPTY \otimes IN^7 \vdash EMPTY \otimes FULL}\,(L\otimes)}{}$$

$$EMPTY \vdash EMPTY \otimes FULL\,(Cut)$$

The plan $\mathcal{P}_2 = \{7 * \mathcal{P}_1, \mathbf{Fill}\}$ presents another plan, which uses a previously canned plan $\mathcal{P}_1$. In this way a modular plan representation is achieved and plan reuse is implemented.

In plans using deterministic actions it is always clear which actions must be executed to achieve a goal. The situation is more complex, if we include nondeterministic actions to an application domain specification. In that case the plan must cover all cases possibly occurring because of nondeterministic actions. A plan is valid if execution of every action in its sequence leads to a goal.

Although LL has been demonstrated to be useful for AI planning [23, 14, 11, 4, 6], there has been little discussion about which algorithms to use for proving LL sequents.

It is intuitively clear that for a smaller set of logic connectives, operators and rules simpler proving techniques are applicable. Thus it makes sense to look for new proving methods.

4 Petri nets and LL

It has been shown [7, 3] that Petri nets can be presented in the form of LL sequents. Thus at least a part of a set of LL sequents can be translated into Petri nets. One of the first surveys on Petri nets is [28], where an overview of basic concepts and extensions and subclasses of Petri nets may be found.

4.1 Petri nets

A Petri net is a formal tool which is particularly well suited for representing true parallelism, concurrency and causal relations in discrete event dynamic systems. In this section we define the concept of Petri net and give the main notation and definitions to be used in the sequel.

A Petri net is a 5-tuple $N = (P, T, Pre, Post, M_0)$, where $P = \{p_1, p_2, \ldots, p_n\}$ is a finite set of places, $T = \{t_1, t_2, \ldots, t_m\}$ is a finite set of transitions, $Pre : P \times T \to \mathcal{N}$ is the input incidence function, $Post : T \times P \to \mathcal{N}$ is the output incidence function and $M_0 : P \to \mathcal{N}$ is the initial marking. A Petri net with a given initial marking is denoted by (N, M_0).

In the graphical representation, circles denote places and vertical bars denote transitions, tokens are represented as dots inside places. The *Pre* incidence function describes the oriented arcs connecting places to transitions. It represents for each transition t the fragment of the state in which the system has to be before the state change corresponding to t occurs. $Pre(p, t)$ is the weight of the arc (p, t), $Pre(p, t) = 0$ denotes that the place p is not connected to transition t.

The *Post* incidence function describes arcs from transitions to places. Analogously to *Pre*, $Post(t, p)$ is the weight of the arc (t, p).

The vectors $Pre(., t)$ and $Post(t, .)$ denote all input and output arcs respectively of transition t with their weights.

The Petri net dynamics is given by firing enabled transitions, whose occurrence corresponds to a state change of the system modeled by the net. A transition t is enabled for a marking M, if $M \geq Pre(., t)$. This enabling condition is equivalent to $\forall p \in P, M(p) \geq Pre(p, t)$. Only enabled transitions can be fired.

If M is a marking of N enabling a transition t, and M' is the marking derived by the firing of a transition t from M, then $M' = (M - Pre(., t)) + Post(t, .)$. The firing is denoted as $M \xrightarrow{t} M'$.

In a Petri net N it is said that a marking M_g is reachable from a marking M iff there exists a sequence of transitions s such that $M \xrightarrow{s} M_g$. We call the *reachability problem* for Petri nets the problem of finding a firing sequence s to reach a given marking M_g from M_0.

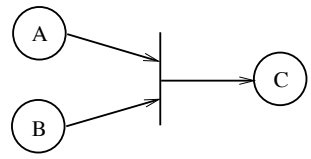

Figure 1: The Petri net representation of LL sequent $A \otimes B \vdash C$.

The *coverability problem* (sometimes also called the submarking reachability problem), given a marking M_g, is defined as the problem of finding a firing sequence s to reach a marking M_s from M_0 such that $M_g \subseteq M_s$.

4.2 Mapping LL sequents to Petri nets

LL sequents involving only $\otimes$, like $A \otimes B \vdash C \otimes D$, can be presented directly in form of Petri nets. Then the set of places P of a Petri net N is augmented with places A, B, C and D. The set of transitions T is augmented with a new transition t_i. *Pre* and *Post* are augmented respectively with $Pre(A, t_i)$, $Pre(B, t_i)$ and $Post(t_i, C)$ plus $Post(t_i, D)$.

Using LL rule $L\oplus$ the sequent $A \oplus B \vdash C$ is splitted into sequents $A \vdash C$ and $B \vdash C$. Constructions like $!A$ may be used only in the left hand side of an sequent, which has to be proved (a goal sequent).

The semantics of the connective $\oplus$ on the right hand side of a sequent should be implemented explicitly using other techniques. The Petri net representation of a LL sequents containing multiplicative conjunctions is demonstrated in Figure 1.

It must be noted that the left hand side of a goal sequent forms the initial state M_0 (marking) of a Petri net and the right hand side forms the final Petri net state M_g (goal in AI planning terminology) which must be achieved as a result of proof search. Thus a proof is found if there exists a way to fire Petri net transitions so that from the initial state the final state is reached.

4.3 Rewriting "of-course" in formulae

To fit into the Petri net framework, formulae containing the "of-course" operator must be rewritten. The following rules should be kept in mind, when doing that:

1. there may be no ! in extralogical axioms
2. there may be no ! in the right hand side of a sequent for which a proof has to be generated
3. for every $!X$ in the left hand side of that sequent generate a new extralogical axiom $\vdash X$ and remove $!X$ from left hand side of the initial sequent
4. if there are several instances of $!X$, then only one axiom $\vdash X$ is generated and all instances are removed from the left side of the sequent

For instance the LL sequent $!A \otimes C \vdash B$ to be proved is translated to sequents $\vdash A$ and $C \vdash B$, whereas $\vdash A$ is a new extralogical axiom and $C \vdash B$ the new sequent to be proved.

5 Solving LL multiplicative conjunctions with Petri nets

Petri nets have been used in AI planning for example in [26, 5, 30, 25] thanks to their clear and well-defined semantics, as well the formal analysis techniques and tools available.

There exist several other works considering AI planning with graphs—quite similar by ideology to Petri nets. For example in [12] colouring of bipartite graphs is used in goal search.

Another AI planner using graph as a planning structure is Graphplan [2], where the application domain, initial conditions and goals are presented as nodes and arcs between nodes. Graphplan uses breadth-first search for finding a solution for achieving a goal. A plan in Graphplan is represented in partial order.

Kanovich [16, 17] proved that the derivability problem of the LL subset consisting only of tensor $\otimes$, modal storage operator !, and linear implication $\multimap$ is directly equivalent to Petri net reachability problem and thus is decidable.

While using Petri net reachability tree analysis for theorem proving, many irrelevant choices in proof search are eliminated, making proof search tractable by:

- avoiding useless loops, which are generated for instance by applications of the *Cut* rule
- reducing the set of permutations of inference as applications of some inference rules like $L\otimes$ and $R\otimes$ are ignored

Useless loops in a proof are characterised by the following situation, where one sequent σ inside a proof is identical to the root of the proof:

$$\begin{array}{c} \vdots \\ \hline \sigma \\ \vdots \\ \hline \sigma \end{array}$$

Hence, the sub-proof starting from that internal sequent could replace the overall proof.

5.1 Petri net reachability tree analysis

For analysing properties of Petri nets, the basic technique used involves finding a finite representation for the reachability set of a Petri net. The representation used is known as the *reachability tree*, which consists of a tree whose nodes represent

markings (states) of a Petri net and whose arcs represent the possible changes in Petri net state resulting the firing of transitions.

Thus a reachability tree represents all Petri net states reachable from an initial Petri net state using all of its transitions.

However, the reachability set of a marked Petri net is often infinite. Thus to form a finite representation of an infinite set we must map many markings into the same node of the tree. This mapping is accomplished by *collapsing* a set of states into a state by ignoring the number of tokens in a place of the net when this number becomes "too large". This is represented by using special symbol ω. The symbol ω represents a value which can be arbitrarily large (infinite), whereby $\omega + a = \omega$, $\omega - a = \omega$ and $a < \omega$, where a is an arbitrary positive integer.

Each node in the reachability tree is labelled with a marking, arcs are labelled with transitions. The initial (root) node is labelled with the initial marking. Given a node x in the tree, additional nodes are added to the tree for all markings that are directly reachable from the marking of the node x. For each transition t_j which is enabled in the marking for node x, a new node with marking[2] $\delta(x, t_j)$ is created and an arc labelled t_j is directed from the node x to this new node. This process is repeated for all new nodes.

Continuing this process will obviously create the entire reachable state space. A path from the initial marking to a node in the tree corresponds to an execution sequence. Since the state space may be infinite, two special steps [18] are taken to define a finite reachability tree.

First, if a new marking is generated, which is equal to an existing marking on the path from the root node to the new marking, the new (duplicate) marking becomes a terminal node. Since the new marking is equal to the previous marking, all markings reachable from it have already been added to the reachability tree by the earlier identical marking. Detecting a new duplicate marking is used later in this article also under the term *cycle detection*.

Second, if any new marking x is generated, which is greater than a marking y on the path from the root node to the marking x, then these components of marking x, which are greater than the corresponding components of marking y are replaced by the symbol ω (this action is further called *collapsing*). Since marking x is greater than marking y, any sequence of transition firings which is possible from marking y, is also possible from marking x. In particular, the sequence that transformed marking y into marking x can be repeated indefinitely, each time increasing the number of tokens in those places, which have a ω. Thus the number of tokens in these places can be made arbitrarily large. A sequence of labels of arcs from the node y to the node x, would be referred later with term *subplan*.

As an example of this construction, consider the marked Petri net in Figure 4. The Petri net consists of 6 places—($EMPTY$, $NEAR$, $HOLD$, IN, $MOVE_OK$, $FULL$) and 4 transitions—(**Take**, **Store**, **Move**, **Fill**). Initially we assume that places $EMPTY$ and $MOVE_OK$ both hold one token.

[2] δ is a transition function from one Petri net state to another.

Thus, the initial state of that Petri net is coded as $\{1,0,0,0,1,0\}$, where the number at the first position is the number of tokens at place $EMPTY$, the second position corresponds to the number of tokens at place $NEAR$, the third at $HOLD$, the fourth at IN, the fifth at $MOVE_OK$ and the sixth at $FULL$.

We begin with $\{1,0,0,0,1,0\}$ as the root node of the tree. In this marking we have only one enabled transition—**Move**. Thus we have a new node corresponding to firing **Move**, $\{1,1,0,0,0,0\}$ and an arc from $\{1,0,0,0,1,0\}$ to $\{1,1,0,0,0,0\}$. From this marking we can fire **Take** and from that newly created node then **Store** resulting in $\{1,0,0,1,1,0\}$. Now, since $\{1,0,0,1,1,0\} > \{1,0,0,0,1,0\}$, we replace the forth component by ω. This reflects the fact that we can fire sequence $\{\mathbf{Move}, \mathbf{Take}, \mathbf{Store}\}$ arbitrary number of times and make the number of tokens in the place IN as large as desired.

From the marking $\{1,0,0,\omega,1,0\}$ we can fire transitions **Fill** and **Move**. After firing **Fill** again collapsing takes place because $\{1,0,0,\omega,1,0\} < \{1,0,0,\omega,1,1\}$. It can be seen that after firing sequentially **Move**, **Take** and **Store** from marking $\{1,0,0,\omega,1,0\}$ we reach again marking $\{1,0,0,\omega,1,0\}$, which is a duplicate and therefore is set to terminal node. The partial Petri net reachability tree is as shown in Figure 5.

Although the Karp-Miller algorithm is useful for the analysis of Petri net reachability tree, due to the loss of information caused by ω, it cannot detect all possible firing sequences needed. A workaround for that problem and several interesting examples about Karp-Miller algorithm may be found at [33].

5.2 Representation of application domain, valid plans and goals

An application domain specified with LL sequents is translated to a Petri net using previously defined transformations. For disjunctions special nodes are added, where splitting to different Petri net places is done.

A valid plan and a subplan is represented by a sequence of the Petri net transitions to be fired for achieving a goal. Every transition may refer to a subplan, which has to be applied after that transition zero or more times. The number of repetitions is computed while checking the correctness of the plan. Note that subplan in our case is not a plan for achieving subgoals—it is just a reusable sequence of transitions.

Every transition in the plan and subplan is enriched with its precondition, which is presented by a Petri net state where that transition was fired. None of transitions in subplan can refer to another subplan. Every subplan is enriched with its precondition. Goal is presented by a Petri net state.

5.3 The PNSolver algorithm

The depth-first algorithm for using Petri net reachability tree analysis for generating plans, where only deterministic actions are considered, is presented in Figure 2.

Naturally other search methods can be easily adapted to that algorithm.

Initially the list of passed states consists of just the initial Petri net state. Initial plan *plan* is empty, *goal* is the state which must be achieved while firing transitions ("executing" actions).

The main idea of the algorithm is to test all transitions in the set T whether they are fireable from a certain Petri net state or not. If a transition is fireable, a new state is computed and it is checked whether the state already exists in the

```
Algorithm PNSolver(init, proof, H, goal)
inputs:  init //initial state of a Petri net
         proof //an initial proof
         H //a set of states visited during the proof search so far
         goal //a final state of a Petri net
output:  P //a set of valid proofs

begin
for ∀t ∈ T
    state ← init
    if fireable(t, state) then
       state ← fire(t, state)
       if state ∈ H then //that state has been visited
          continue //do not proceed (Karp-Miller)
       end if
       proof.push(t)
       state ← collapse(state, H) //collapse state space (Karp-Miller)
       H.push(state)
       if state ≡ goal then //a proof is found
          announce(proof)
          proof.pop()
          H.pop()
          continue
       end if
       proof.pop()
       H.pop()
    end if
end for
for ∀p ∈ announcedProofs()
    p ← CheckCorrectness(init, p, goal)
    P ← P ∪ p
end for
return P
end PNSolver
```

Figure 2: A pseudocode for finding sequences of transitions from the initial state of a Petri net to the final state.

sequence of visited states H. If the state happens to exist in H, a cycle is detected and search from that Petri net reachability tree node is terminated. As the cycle was detected, it is clear that if the goal state was not found in previous round of that cycle, it would not be found on the next round either.

Else, if the *state* was not discovered in H, *state* is added to H, *plan* is augmented with a transition, and Petri net possibly infinite state space is collapsed, if possible. Collapsing generates a subplan, which is added to a list of subplans and a reference from the last transition in the plan to that subplan is inserted. Also information about how many resources that subplan generated, consumed and its precondition is remembered. For more information about Petri net reachability tree analysis see [28].

The usage of subplans reduces dramatically the time needed to construct a plan if used wisely [29]. In our case subplans are generated as a side-effect using Petri net state space collapsing.

If the achieved state is equivalent to *goal*, search is terminated at that particular reachability tree node, plan is added to a list of plans P, and the inspection of next transitions begins.

In the case goal is not achieved after firing particular transition, search from the new state is recursively proceeded until whole available search space is explored. The possibly infinite search-space is reduced by cycle detection and collapsing.

Checking the correctness of a plan (see algorithm in Figure 3) in the end of algorithm is started to solve ambiguities generated through collapsing. Correctness checking computes how many times certain subplans must be executed sequentially to achieve needed amount of resources. During the correctness checking it may turn out that some goals are not valid at all and the exact number of some resources cannot be achieved. It may turn out for example that only even number of units of a resource may be generated instead of needed odd number defined by the goal.

Correctness checking starts from the goal state and moves towards the initial state, while undoing effects of fired transitions. If a transition referring to a subplan is detected, the number of subplan execution cycles is computed according to needed resources. If finally Petri net state *init* is achieved, the plan is considered to be valid and is returned.

To illustrate this algorithm, let us take a look again at the box filling problem (see Sect. 3.2) we solved previously and modify[3] the application domain specification to be more "real":

Take: $EMPTY \otimes NEAR \vdash HOLD$
Store: $HOLD \vdash EMPTY \otimes IN \otimes MOVE_OK$
Move: $MOVE_OK \vdash NEAR$
Fill: $IN^7 \vdash FULL$

The Petri net representing the same specification consists of 6 places ($EMPTY$, $NEAR$, $HOLD$, IN, $MOVE_OK$, $FULL$) and 4 transitions (**Take**, **Store**, **Move**, **Fill**). Initially we assume that places $EMPTY$ and $MOVE_OK$ both have one

[3] In the previous version of the specification a robot was able to run from one place to another and then pick up balls it encountered from one final place.

```
Algorithm CheckCorrectness(init, proof, goal)

inputs:  init //initial state of a Petri net
         proof //an initial proof
         goal

output:  p //a complete proof

begin
state ← goal
for ∀t ∈ p
    if referenceToSubsolution(t) then
       t ← addSubsolutionRepetitions(t, state, init)
       state ← undoSubsolution(t, state)
    else
       state ← undo(t, state)
    end if
end for
if state ≡ init then
    return p
else
    return nil
end CheckCorrectness
```

Figure 3: A pseudocode for checking the validity of a proof.

token and the goal would be to get one token to each of $EMPTY$, $MOVE_OK$ and $FULL$. In LL terms it means finding a proof for a sequent $MOVE_OK \otimes EMPTY \vdash FULL \otimes MOVE_OK \otimes EMPTY$. See also Figure 4 for graphical representation of that Petri net and its initial state.

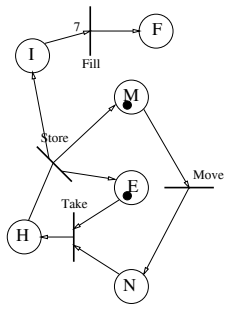

Figure 4: The Petri net with an initial marking for the box filling example.

So, the initial state of that Petri net is coded as $\{1,0,0,0,1,0\}$, where the number at the first position is number of tokens at place $EMPTY$, the second position corresponds to the number of tokens at place $NEAR$, the third at $HOLD$, the fourth at IN, the fifth at $MOVE_OK$ and the sixth at $FULL$. The goal state is accordingly $\{1,0,0,0,1,1\}$. Petri net reachability tree for our application domain and goal specification is in Figure 5.

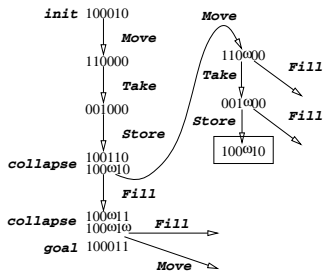

Figure 5: A fragment of reachability tree of Petri net in Figure 4. Arcs without ending node show that the tree follows.

Petri net transitions can be presented with STRIPS-like *add*- and *delete*-lists, see Table 1. The precondition in STRIPS sense for every transition is that there is enough of tokens to fire particular transition.

Transition	*delete*-list	*add*-list
Move	000010	010000
Take	110000	001000
Store	001000	100110
Fill	000700	000001

Table 1: Petri net transitions as *add*- and *delete*-lists.

It can be seen from the reachability tree that after firing transitions **Move**, **Take** and **Store**, we reach a state where compared to the initial state the number of tokens at place IN has increased: $\{1,0,0,1,1,0\} > \{1,0,0,0,1,0\}$. Therefore this component is replaced by ω, meaning that using that sequence again we can generate infinite number of resources IN—this is called Petri net state space collapsing. After collapsing we apply **Fill** and reach the state which is a possible goal state and resulting plan is $\mathcal{P} = \{n * \{\mathbf{Move}, \mathbf{Take}, \mathbf{Store}\}, m * \mathbf{Fill}\}$.

It is evident that through Petri net state-space collapsing we reduce the search space, but we lose information about how many times a subplan must be executed. Therefore we have to start with correctness checking to compute the number of times we have to execute generated subplans. In that particular case the final plan is $\mathcal{P} = \{7 * \{\mathbf{Move}, \mathbf{Take}, \mathbf{Store}\}, 1 * \mathbf{Fill}\}$.

Empirically estimating, a valid plan with our algorithm (even when using breadth-first search) may be found with smaller number of steps than Graphplan planner could, because in our case subplans are used and cycles are detected while planning.

Unfortunately finding the *shortest* plan requires searching the whole available search space—all valid plans are computed, subplans are unfolded and then the shortest plan is selected. It is due to the fact that we do not know how many times a subplan must be executed—it may be 0 as well as 1000 times. As the previous example illustrated, a plan with length 22 was found from the Petri net reachability tree at depth 4. Thus there is no strong connection between a search depth and a plan length if the Karp-Miller algorithm is used.

6 Using game playing for solving LL additive disjunctions

In respect to planning, having a LL sequent $A \vdash B \oplus C$, we do not know which one of the resources B or C would become the result of execution of particular action. Therefore a way to handle sequents like $A \vdash B \oplus C$ within Petri nets we may choose between stochastic Petri nets [27] and coloured Petri nets [15], sometimes misleadingly called high-level nets. In such a way the model can be specified within one net. The colour of tokens depends on which disjunct was selected as a result of applying particular transition.

As stochastic Petri nets alter in some way the firing rule of Petri nets and make it so the main part of net theory no longer applicable [27], they should be avoided as long as possible.

Instead of using previously mentioned Petri net derivations we start game playing on a Petri net reachability tree. The advantage of game playing on a tree is pruning of search space by using AND nodes. Thus transition selection represents OR and disjunct selection represents AND level of a game.

An advanced PNSolver algorithm, PNGameSolver, for solving disjunctions on the right hand side of LL sequents is presented in Figure 6. The only difference with algorithm in Figure 2 is that here reachability of the goal state from all disjuncts is considered. If at least from one disjunct the goal is unachievable, search with particular transition is terminated, backtracking is performed and search at other OR node proceeds.

If the goal at least from one disjunct is not achievable, plans for other disjuncts at the same AND node are discarded. In addition, it must be noted that at the current moment the effect of subplans including nondeterministic actions is not quite clear and therefore Petri net state space collapsing is not applied if at least one action in resulting subplan would represent a nondeterministic action. Therefore generating plans including disjunctions is quite exhaustive.

```
Algorithm PNGameSolver(init, proof, H, goal)
output:  P //a set of valid proofs

begin
upper_cycle:  for ∀t ∈ T //OR node
    for ∀disj ∈ t //AND node
        ...
        if notAchievable(disj) then
           continue upper_cycle
        end if
    end for
end for
...
return P
end PNGameSolver
```

Figure 6: A pseudocode for handling nondeterminism within Petri nets.

7 Computational complexity of the PNSolver and the PNGameSolver algorithm

Lipton proved [22] an exponential space lower bound for Petri net reachability problem, while the known algorithms require nonprimitive recursive space. As PNSolver algorithm uses Petri net reachability tree analysis for finding solutions, its minimal complexity is EXPSPACE-hard.

PNGameSolver uses additionally games in Petri net reachability tree analysis, which makes its complexity comparable to reachability problem of nondeterministic Petri nets, whose complexity is proved [16] to be undecidable.

However, tight complexity bounds of the reachability problem are known for many Petri net classes [8]. For example, If we would limit expressive power of used LL sequents so that sinkless or normal Petri nets may be used, we would achieve NP-complete complexity [13] for reachability analysis.

As we are using Petri net reachability tree analysis *on the fly*, meaning that we build Petri net and analyse it simultaneously, it is possible to bypass difficulties arised from complexities of algoritms by using heuristics at search. If powerful heuristics is used only a fragment of reachability tree would be generated before a solution is found.

Another constraint, we may set, is to fix a bound on the number of tokens at places—algorithms for bounded Petri nets are less expensive in complexity.

In [20] an abstraction technique for LL theorem proving is proposed which in the best case reduces the exponential problem solving complexity of PNSolver algorithm to linear.

8 Conclusions

In this article we proposed a new way of resource-conscious AI planning using propositional LL sequents as a knowledge representation form. These sequents are translated into Petri nets and then a plan achieving a certain goal is computed.

A fusion of Petri net reachability tree analysis and game playing is used to solve problems described with LL sequents. Whilst game playing allows handling LL additive disjunctions, Petri net analysis handles LL multiplicative conjunctions and preserves resource-consciousness.

By Petri net state space collapsing subplans are generated and thereby plan generation time is reduced.

Experimental results with PNSolver algorithm are presented in [19], where a comparison between depth-first and breadth-first search algorithms with and without certain extensions is given.

Acknowledgements

I would like to express my gratitude to Tarmo Uustalu from Software Department, Institute of Cybernetics at Tallinn Technical University for his assistance in LL.

Also I would like to thank Keijo Heljanko from Theoretical Computer Science Laboratory, Helsinki University of Technology for introducing me to the computational complexity of Petri net problems and pointing to some weaknesses and faults in my algorithms.

References

[1] V. Alexiev. Applications of Linear Logic to Computation: An Overview. Bulletin of the IGPL, Vol. 2, No. 1, March 1994.

[2] A. L. Blum, M. L. Furst. Fast Planning Through Planning Graph Analysis. Artificial Intelligence, Vol. 90, pp. 281–300, 1997.

[3] C. T. Brown. Linear Logic and Petri Nets: Categories, Algebra and Proof. PhD thesis, Department of Computer Science, University of Edinburgh, Scotland, 1991.

[4] S. Brüning, S. Hölldobler, J. Schneeberger, U. Sigmund, M. Thielscher. Disjunction in Resource-Oriented Deductive Planning. Technical Report AIDA-93-03, Technische Hochschule Darmstadt, Germany, 1994.

[5] S. Caselli, F. Zanichelli. On assembly sequence planning using Petri nets. Proceedings of IEEE International Symposium on Assembly and Task Planning, Pittsburgh, Pennsylvania, August 1995, pp. 239–244, 1995.

[6] S. Cresswell, A. Smaill, J. Richardson. Deductive Synthesis of Recursive Plans in Linear Logic. In Proceedings of the Fifth European Conference on Planning, pp. 252–264, 1999.

[7] U. Engberg, G. Winskel. Petri nets as models of Linear Logic. In: A. Arnold (ed). Proceedings of Colloquium on Trees in Algebra and Programming (CAAP'90), Copenhagen, Denmark, May 15–18, 1990, Lecture Notes in Computer Science, Vol. 431, Springer-Verlag, pp. 147–161, 1990.

[8] J. Esparza, M. Nielsen. Decidability Issues for Petri Nets—a Survey. Journal of Information Processing and Cybernetics, Vol. 30, No. 3, pp. 143–160, 1995.

[9] R. Fikes, N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence, Vol. 2, pp. 189–208, 1971.

[10] J.-Y. Girard. Linear Logic. Theoretical Computer Science, Vol. 50, pp. 1–102, 1987.

[11] G. Grosse, S. Hölldobler, J. Schneeberger. Linear Deductive Planning. Journal of Logic and Computation, Vol. 6, pp. 232–262, 1996.

[12] M. Harf, E. Tyugu. Algorithms of structured synthesis of programs. Programming and Computer Software, Vol. 6, pp. 165–175, 1980.

[13] R. Howell, L. Rosier, H. Yen. Normal and Sinkless Petri Nets. Journal of Computer and System Sciences, Vol. 46, pp. 1–26, 1993.

[14] E. Jacopin. Classical AI planning as theorem proving: The case of a fragment of Linear Logic. In: AAAI Fall Symposium on Automated Deduction in Nonstandard Logics, Palo Alto, California, AAAI Press, pp. 62–66, 1993.

[15] K. Jensen. Coloured Petri Nets. In: W. Brauer, W. Reisig, G. Rozenberg (eds). Petri Nets: Central Models and Their Properties. Proceedings of Advances in Petri Nets 1986, Part I, Bad Honnef, September 8–19, 1986, Lecture Notes in Computer Science, Vol. 254, Springer-Verlag, pp. 248–299, 1987.

[16] M. I. Kanovich. Petri Nets, Horn Programs, Linear Logic, and Vector Games. In: M. Hagiya, J. C. Mitchell (eds). Theoretical Aspects of Computer Software, International Symposium TACS'94, Sendai, Japan, Lecture Notes in Computer Science, Vol. 789, Springer-Verlag, pp. 642–666, 1994.

[17] M. I. Kanovich. The complexity of Horn fragments of Linear Logic. Annals of Pure and Applied Logic, Vol. 69, No. 2–3, pp. 195–241, 1994.

[18] R. M. Karp, R. E. Miller. Parallel program schemata. Journal of Computer and Systems Sciences, Vol. 3, No. 2, pp 147–195, May 1969.

[19] P. Küngas. Linear Logic Programming for AI Planning. Master thesis, Tallinn Technical University, Estonia, Research Report CS 103/02, Institute of Cybernetics at Tallinn Technical University, May 2002.

[20] P. Küngas. Linear Logic Theorem Proving with Abstraction. To appear in Proceedings of 14th European Summer School in Logic, Language and Information, ESSLLI'2002, Trento, Italy, 5-16 August, 2002.

[21] P. Lincoln. Linear Logic. ACM SIGACT Notices, Vol. 23, No. 2, pp. 29–37, Spring 1992.

[22] R. J. Lipton. The Reachability Problem Requires Exponential Space. Department of Computer Science, Research Report 62, Yale University, 1976.

[23] M. Masseron, C. Tollu, J. Vauzeilles. Generating plans in Linear Logic I–II. Theoretical Computer Science, Vol. 113, pp. 349–375, 1993.

[24] J. McCarthy, P. J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer, D. Michie, M. Swann (eds). Machine Intelligence, Vol. 4, Edinburgh University Press, pp. 463–502, 1969.

[25] Y. Meiller, P. Fabiani. Planning with Petri nets. Proceedings of RJCIA'2000, 2000.

[26] K. E. Moore, A. Gungor, S. M. Gupta. Disassembly process planning using Petri nets. Proceedings of IEEE International Symposium on Electronics and the Environment, Oak Brook, IL, pp. 88–93, 1998.

[27] A. Pagnoni. Stochastic Nets and Performance Evaluation. In: W. Brauer, W. Reisig, G. Rozenberg (eds). Petri Nets: Central Models and Their Properties. Proceedings of Advances in Petri Nets 1986, Part I, Bad Honnef, September 8–19, 1986. Lecture Notes in Computer Science, Vol. 254, Springer-Verlag, pp. 460–478, 1987.

[28] J. L. Peterson. Petri nets. ACM Computing Surveys, Vol. 9, pp. 223–252, 1977.

[29] D. Ruby, D. Kibler. Learning Subgoal Sequences for Planning. Proceedings of IJCAI'89, Detroit, Michigan USA, 20–25 August 1989, Vol. 1, pp. 609–614, 1989.

[30] F. Silva, M. Castilho, L. A. Künzle. Petriplan: a new algorithm for plan generation (Preliminary report). In M. C. Monard, J. S. Sichman (eds). Advances in Artificial Intelligence. Proceedings of International Joint Conference 7th Ibero-American Conference on AI 15th Brazilian Symposium on AI, IBERAMIA-SBIA 2000, Atibaia, SP, Brazil, November 19-22, 2000, Lecture Notes in Computer Science, Vol. 1952, Springer-Verlag, 2000.

[31] T. Tammet. Proof Strategies in Linear Logic. Journal of Automated Reasoning, Vol. 12, pp. 273–304, 1994.

[32] A. S. Troelstra. Tutorial on Linear Logic. In: P. Schroeder-Heister, K. Dosen (eds). Substructural Logics, Oxford University Press, pp. 327–355, 1993.

[33] F.-Y. Wang. A Modified Reachability Tree for Petri Nets. Proceedings of 1991 IEEE International Conference on Systems, Man, and Cybernetics, Blackburg, VA, pp. 329–334, 1991.