Factored Planning: From Automata to Petri Nets

Loïg Jezequel, Université de Nantes, IRCCyN, UMR CNRS 6597, Nantes, France
Eric Fabre, INRIA Rennes Bretagne Atlantique, Rennes, France
Victor Khomenko, Newcastle University, Newcastle, United Kingdom

Factored planning mitigates the state explosion problem by avoiding the construction of the state space of the whole system and instead working with the system's components. Traditionally, finite automata have been used to represent the components, with the overall system being represented as their product. In this paper we change the representation of components to safe Petri nets. This allows one to use cheap structural operations like transition contractions to reduce the size of the Petri net, before its state space is generated, which often leads to substantial savings compared with automata. The proposed approach has been implemented and proven efficient on several factored planning benchmarks.

This paper is an extended version of our ACSD 2013 paper [Jezequel et al. 2013], with the addition of the proofs and the experimental results of Sections 6 and 7.

Categories and Subject Descriptors: [**Computing methodologies**]: Planning and scheduling; [**Software and its engineering**]: Petri nets

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: Factored planning, Weighted Petri nets

ACM Reference Format:
Loïg Jezequel, Eric Fabre, and Victor Khomenko. Factored Planning: From Automata to Petri Nets. *ACM Trans. Embedd. Comput. Syst.* 0, 0, Article 00 (0000), 25 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Planning consists in organising a set of actions in order to reach some predefined (set of) goal state(s), where each action modifies some of the state variables of the considered system. In that sense, planning is very similar to reachability analysis in model checking, or to path search in a graph, viz. the state graph of the system. A solution to these problems is either an action plan reaching the goal, or an example of a run proving the reachability, or a successful path in a graph. These problems have benefited much from the introduction of true concurrency semantics [Esparza and Heljanko 2008] to describe plans or runs. Concurrency represents the possibility to execute simultaneously several actions that involve different subsets of resources. With such semantics, a plan or a trajectory becomes a partial order of actions rather than a sequence, which can drastically reduce the number of trajectories to explore.

In the planning community, it was soon observed that concurrency could be turned into an ally, as one can avoid the exploration of meaningless interleavings of actions. The first attempt in that direction was GRAPHPLAN [Blum and Furst 1995], which lays plans on a data structure representing explicitly the parallelism of actions. This data structure has connections with merged processes [Khomenko et al. 2006] and

Author's addresses: loig.jezequel@irccyn.ec-nantes.fr, eric.fabre@inria.fr, victor.khomenko@ncl.ac.uk

trellis processes [Fabre 2007b], where the conflict relation is non-binary and can not be checked locally. GRAPHPLAN did not notice specifically these facts, and chose to connect actions with a loose and local check of the conflicts. Hence the validity of the extracted plans had to be checked, and numerous backtrackings were necessary. A more rigourous approach to concurrent planning was later proposed by [Hickmott et al. 2007] and improved in [Bonet et al. 2008]. The idea was to represent a planning problem as an accessibility problem for a safe Petri net (possibly with read arcs). One can then represent concurrent runs of the net using unfoldings, and the famous A* search algorithm was adapted to Petri net unfoldings.

An alternative and indirect way to take advantage of concurrency in planning problems is the so-called *factored planning* approach. It was first proposed in [Amir and Engelhardt 2003], and variations on this idea were described in [Brafman and Domshlak 2008; Fabre et al. 2010]. Factored planning consists in splitting a planning problem into simpler subproblems, involving fewer state variables. If these subproblems are loosely coupled, they can be solved almost independently, provided one properly manages the actions shared by several subproblems. In [Fabre et al. 2010], the problem was expressed under the form of a network (actually a product) of automata, that must be driven optimally to a target state. A plan in this setting is a tuple of executions (one per component) synchronised on shared actions. In this representation, a plan is again a partial order on events, and the concurrency between components is maximally exploited. This is what we call the *global concurrency*, the concurrency of actions belonging to different components. However, this approach fails to take advantage of a *local concurrency*, that would be internal to each component or each subproblem. This is specifically the point addressed by this paper: we replace the automaton encoding a planning sub-problem (which we call a component) by a Petri net, which allows for a natural representation of internal concurrency in this component. We therefore encode a planning problem as a product of Petri nets, and extend the distributed planning techniques to this setting.

The contributions of this paper can be summarised in the following points. First, it reconciles local and global concurrency in the factored planning approach. This means taking advantage both of the concurrency between components of a large planning problem, or equivalently of the loose coupling of planning subproblems, and of the concurrency that is internal to each component. In other words, this paper demonstrates that the planning approach proposed in [Hickmott et al. 2007] can be coupled with distributed/factored planning ideas as developed in [Fabre et al. 2010]. The main move consists in replacing modular calculations performed on automata by calculations performed on Petri nets, which are often significantly faster.

Secondly, these ideas are experimentally evaluated on standard benchmarks from [Corbett 1996], in order to demonstrate the gains obtained by exploiting the local concurrency within each component. In particular, we compare the runtimes of the distributed computations described in [Fabre et al. 2010] with those obtained when automata are replaced by Petri nets.

Finally, we extend the results of [Vogler and Kangsah 2007] on the projection of Petri nets to the case of Petri nets with costs attached to transitions, which allows one to apply the developed techniques to the cost-optimal planning. We also experimentally evaluate the efficiency of this projection.

2. PETRI NETS AND FACTORED PLANNING

This section recalls the standard STRIPS propositional formalism that is commonly used to describe planning problems. It then explains how factored planning problems can be recast into an accessibility problem (or more precisely a fireability problem for a specific labelled transition) for a product of Petri nets.

2.1. Planning problems

A *planning problem* is a tuple (A, O, i, G) where A is a set of *atoms*, $O \subseteq 2^A \times 2^A \times 2^A$ is a set of *operators* or *actions*. A *state* of the planning problem is an element of 2^A, or equivalently a subset of atoms. $i \subseteq A$ is the *initial* state, and $G \subseteq A$ defines a set of *goal* states as follows: $s \subseteq A$ is a goal state iff $G \subseteq s$. An operator $o \in O$ is defined as a triple $o = (pre, del, add)$ where pre is called the *precondition* of o, del is called its *negative effect*, and add is called its *positive effect*. The operator $o = (pre, del, add)$ is *enabled* from a state $s \subseteq A$ as soon as $pre \subseteq s$. In this case o can *fire*, which leads to the new state $o(s) = (s \setminus del) \cup add$. The objective of a planning problem is to find a sequence $p = o_1 \dots o_n$ of operators such that i enables o_1, for any $k \in [2..n], o_{k-1}(\dots(o_1(i)))$ enables o_k, and $o_n(\dots(o_1(i))) \supseteq G$.

One can directly translate a planning problem into a directed graph, where the nodes of the graph represent the states and the arcs are derived from the operators. Solutions to the planning problem are then paths leading from i to goal states. Traditional planners thus take the form of path search algorithms in graphs: most of them derive from the well-known A* algorithm [Hart et al. 1968] and provide plans as sequences of operator firings. A more recent set of works tried to take advantage of the locality of operators: they involve limited sets of atoms, which means that some operators can fire concurrently. This leads to the idea of providing plans as partial orders of operator firings rather than sequences. These approaches rely on the translation of planning problems into safe Petri nets [Hickmott et al. 2007], and look for plans using unfolding techniques [Esparza et al. 1996] in combination with an adapted version of A* [Bonet et al. 2008].

2.2. Petri nets and planning problems

A *net* is a tuple (P, T, F) where P is a set of *places*, T is a set of *transitions*, $P \cap T = \emptyset$, and $F : (P \times T) \cup (T \times P) \to \mathbb{N}$ is a *flow function*. For any *node* $x \in P \cup T$, we denote by ${}^\bullet x$ the set $\{y : F(y, x) > 0\}$ of *predecessors* of x, and by $x^\bullet$ the set $\{y : F(x, y) > 0\}$ of *successors* of x. In a net, a *marking* is a function $M : P \to \mathbb{N}$ associating a natural number to each place. A marking M *enables* a transition $t \in T$ if $\forall p \in {}^\bullet t, M(p) \geq F(p, t)$. In such a case, the *firing* of t from M leads to the new marking M_t such that $\forall p \in P, M_t(p) = M(p) - F(p, t) + F(t, p)$. In the sequential semantics, an *execution* from marking M is a sequence of transitions $t_1 \dots t_n$ such that M enables t_1, and for any $k \in [2..n], M_{t_1 \dots t_{k-1}}$ enables t_k (where $M_{t_1 \dots t_i}$ is defined recursively as $M_{t_1 \dots t_i} = (M_{t_1 \dots t_{i-1}})_{t_i}$). We denote by $\langle M \rangle$ the set of executions from a marking M.

A *Petri net* is a tuple (P, T, F, M^0) where (P, T, F) is a net and M^0 is the *initial* marking. In a Petri net, a marking M is said to be *reachable* if there exists an execution $t_1 \dots t_n$ from M^0 such that $M = M^0_{t_1 \dots t_n}$. A Petri net is said to be *k-bounded* if any reachable marking M is such that $\forall p \in P, M(p) \leq k$. It is said to be *safe* if it is 1-bounded.

A *labelled Petri net* is a tuple $(P, T, F, M^0, \Lambda, \lambda)$ where (P, T, F, M^0) is a Petri net, Λ is an alphabet, and $\lambda : T \to \Lambda \cup \{\varepsilon\}$ is a labelling function associating a label from $\Lambda \cup \{\varepsilon\}$ to each transition. The special label ε is never an element of Λ and the transitions with label ε are called *silent transitions*. In such a labelled Petri net, the *word* associated to an execution $o = t_1 \dots t_n$ is $\lambda(o) = (\lambda(t_1) \dots \lambda(t_n))_{|\Lambda}$ (so silent transitions are ignored). The language of a labelled Petri net $N = (P, T, F, M^0, \Lambda, \lambda)$ is the set $\mathcal{L}(N)$ of all the words corresponding to executions from M^0 in N:

$$\mathcal{L}(N) = \{\lambda(o) : o \in \langle M^0 \rangle\}.$$

[Hickmott et al. 2007] proposed a representation of planning problems as safe labelled Petri nets. Each atom was represented by a place, and for technical reasons –

namely to guarantee the safeness of the Petri net – [Hickmott et al. 2007] also introduced complementary places representing the negation of each atom. The initial state i naturally gives rise to the initial marking M^0. An operator o is then instantiated as several transitions labelled by o, one per possible enabling of this operator. This duplication is due to the fact that an operator can delete an atom that it does not request as an input, and thus that could either be present or not. The goal states, corresponding to goal markings of the Petri net, are then captured using an additional transition that consumes the tokens in the places representing the atoms G, which is thus enabled if and only if a marking corresponding to some goal state has been reached.

From now on we consider that a planning problem is a pair (N, g) where $N = (P, T, F, M^0, \Lambda, \lambda)$ is a safe labelled Petri net and $g \in \Lambda$ is a particular ***goal label***. In such a problem one wants to find an execution $o = t_1 \dots t_n$ from M^0 such that for any $k \in [1..n-1], \lambda(t_k) \neq g$ and $\lambda(t_n) = g$. We denote the set of all such executions by $\mathcal{L}^g(N) = \mathcal{L}(N) \cap (\Lambda \setminus \{g\})^* g$.

2.3. Petri nets and factored planning problems

A factored planning problem is defined by a set of interacting planning subproblems [Amir and Engelhardt 2003; Brafman and Domshlak 2008; Fabre et al. 2010]. These interactions can take the form of shared atoms or of shared actions, but one can simply turn one model into its dual. So we choose here the synchronisation on shared actions, which naturally fits with the notion of synchronous product. In the context of Petri nets, a factored planning problem takes the form of a set of Petri nets synchronised on shared transition labels: if two nets share a transition label σ, the transitions of these nets labelled by σ have to be fired simultaneously. This synchronisation on shared labels – which corresponds to the synchronisation on shared actions for planning problems – can be formalised as the product of labelled Petri nets.

The *product* of two labelled Petri nets N_1 and N_2 (with alphabets Λ_1 and Λ_2) is a labelled Petri net $N = N_1 || N_2$ (with alphabet $\Lambda_1 \cup \Lambda_2$) representing the parallel executions of N_1 and N_2 with synchronisations on common transition labels from $\Lambda_1 \cap \Lambda_2$ (notice that ε is never a common label as, by definition, it never belongs to Λ_1 nor Λ_2). It is obtained from the disjoint union of N_1 and N_2 by fusing each σ-labelled transition of N_1 with each σ-labelled transition of N_2, for each common action σ, and then deleting the original transitions that participated in such fusions. An example is given in Figure 1.

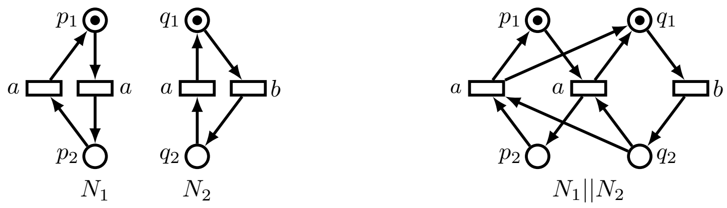


Fig. 1. Two Petri nets and their product, here $\Lambda_1 \cap \Lambda_2 = \{a\}$.

In the following definition, $\star \notin P_1 \cup T_1 \cup P_2 \cup T_2$ is an artificial element used for convenience. Let $N_1 = (P_1, T_1, F_1, M_1^0, \Lambda_1, \lambda_1)$ and $N_2 = (P_2, T_2, F_2, M_2^0, \Lambda_2, \lambda_2)$, then $N = (P, T, F, M^0, \Lambda, \lambda)$ with: $P = P_1 \cup P_2$, $T = \{(t_1, t_2) : t_1 \in T_1, t_2 \in T_2, \lambda_1(t_1) = \lambda_2(t_2) \neq \varepsilon\} \cup \{(t_1, \star) : t_1 \in T_1, \lambda_1(t_1) \notin \Lambda_2\} \cup \{(\star, t_2) : t_2 \in T_2, \lambda_2(t_2) \notin \Lambda_1\}$, $F(p, (t_1, t_2))$ equals $F_1(p, t_1)$ if $p \in P_1$ and else equals $F_2(p, t_2)$, $F((t_1, t_2), p)$ equals $F_1(t_1, p)$ if $p \in P_1$ and else equals $F_2(t_2, p)$, $M^0(p)$ equals $M_1^0(p)$ for $p \in P_1$ and else equals $M_2^0(p)$, $\Lambda =$

$\Lambda_1 \cup \Lambda_2$, and finally $\lambda((t_1, t_2))$ equals $\lambda_1(t_1)$ if $t_1 \neq \star$ and else equals $\lambda_2(t_2)$). Note that if N_1 and N_2 are safe then their product $N_1||N_2$ is safe as well.

From that, a *factored planning problem* is defined as a tuple $N = (N_1, \ldots, N_n)$ of planning problems $N_i = ((P_i, T_i, F_i, M_i^0, \Lambda_i, \lambda_i), g)$ (all having the same goal label g). The N_is are the *components* of N. Given such a tuple, one has to find a solution to the planning problem $(N_1|| \ldots ||N_n, g)$ without computing the full product of the components (as the number of transitions in this product can be exponential in the number of components). In other words, one would like to find this solution doing only local computations for each component N_i (that is computations involving only N_i and its neighbours, i.e. the components sharing labels with N_i).

3. MESSAGE PASSING ALGORITHMS

[Fabre and Jezequel 2009; Fabre et al. 2010] solved factored planning problems using a particular instance of the message passing algorithms described in [Fabre 2007a]. This section recalls this algorithm in the context of languages and shows that it can be instantiated for solving factored planning problems represented by synchronised Petri nets.

3.1. A message passing algorithm for languages

The message passing algorithm that we present here is based on the notions of product and projection of languages.

The *projection* of a language $\mathcal{L}$ over an alphabet Λ to an alphabet Λ' is the language:

$$\Pi_{\Lambda'}(\mathcal{L}) = \{w_{|\Lambda'} \ : \ w \in \mathcal{L}\},$$

where $w_{|\Lambda'}$ is the word obtained from w by removing all letters not from Λ'. The *product* of two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ over alphabets Λ_1 and Λ_2, respectively, is

$$\mathcal{L}_1||\mathcal{L}_2 = \Pi^{-1}_{\Lambda_1 \cup \Lambda_2}(\mathcal{L}_1) \cap \Pi^{-1}_{\Lambda_1 \cup \Lambda_2}(\mathcal{L}_2),$$

where the inverse projection $\Pi^{-1}_{\Lambda'}(\mathcal{L})$ of a language $\mathcal{L}$ over an alphabet $\Lambda \subseteq \Lambda'$ is

$$\Pi^{-1}_{\Lambda'}(\mathcal{L}) = \{w \in {\Lambda'}^* \ : \ w_{|\Lambda} \in \mathcal{L}\}.$$

Suppose a language (the global system) is specified as a product of languages $\mathcal{L}_1, \ldots, \mathcal{L}_n$ (the components) defined respectively over the alphabets $\Lambda_1, \ldots, \Lambda_n$. The *interaction graph* between components $(\mathcal{L}_i)_{i \leq n}$ is defined as the (non-directed) graph $\mathcal{G} = (V, E)$ whose vertices V are these languages and such that there is an edge $(\mathcal{L}_i, \mathcal{L}_j)$ in E if and only if $i \neq j$ and $\Lambda_i \cap \Lambda_j \neq \emptyset$. In such a graph an edge $(\mathcal{L}_i, \mathcal{L}_j)$ is said to be *redundant* if and only if there exists a path $\mathcal{L}_i\mathcal{L}_{k_1} \ldots \mathcal{L}_{k_\ell}\mathcal{L}_j$ between $\mathcal{L}_i$ and $\mathcal{L}_j$ such that: for any $m \in [1..\ell]$ one has $k_m \neq i$, $k_m \neq j$, and $\Lambda_{k_m} \supseteq \Lambda_i \cap \Lambda_j$. By iteratively removing redundant edges from the interaction graph until reaching stability (i.e. until no more edge can be removed) one obtains a *communication graph* $\mathcal{G}$ between components $\mathcal{L}_1, \ldots, \mathcal{L}_n$. By $\mathcal{N}(i)$ we denote the set of indices of neighbours of $\mathcal{L}_i$ in $\mathcal{G}$, i.e. the set of all j such that there is an edge between $\mathcal{L}_i$ and $\mathcal{L}_j$ in $\mathcal{G}$. Note that if any communication graph for these languages is a tree, then all their communication graphs are trees [Fabre 2007a]. In this case the system is said to *live on a tree*.

PROPOSITION 3.1 ([FABRE AND JEZEQUEL 2009]). *If the system lives on a tree, Algorithm 1 computes for each $\mathcal{L}_i$ some $\mathcal{L}'_i$, such that $\mathcal{L}'_i = \Pi_{\Lambda_i}(\mathcal{L}_1|| \ldots ||\mathcal{L}_n) \subseteq \mathcal{L}_i$.*

These reduced $\mathcal{L}'_i$ still satisfy $\mathcal{L} = \mathcal{L}'_1|| \ldots ||\mathcal{L}'_n$, so they still form a valid (factored) representation of the global system $\mathcal{L}$. Moreover, they are the smallest sub-languages of the $\mathcal{L}_i$ that preserve this equality. Algorithm 1 runs on a communication graph $\mathcal{G}$.

Algorithm 1 Message passing algorithm for languages

```
1: $M \leftarrow \{(i,j),(j,i) \mid (\mathcal{L}_i,\mathcal{L}_j)$ is an edge of $\mathcal{G}\}$
2: while $M \neq \emptyset$ do
3:     extract $(i,j) \in M$ such that $\forall k \neq j, (k,i) \notin M$
4:     set $\mathcal{M}_{i,j} = \Pi_{\Lambda_j}(\mathcal{L}_i||(||_{k\in\mathcal{N}(i)\setminus\{j\}}\mathcal{M}_{k,i}))$
5: end while
6: for all $\mathcal{L}_i$ in $\mathcal{G}$ do
7:     set $\mathcal{L}'_i = \mathcal{L}_i||(||_{k\in\mathcal{N}(i)}\mathcal{M}_{k,i})$
8: end for
```

It first computes languages $\mathcal{M}_{i,j}$ (line 4, where $||_{k\in\emptyset}\mathcal{M}_{k,i}$ is the neutral element of $||$: a language containing only an empty word and defined over the empty alphabet), called *messages,* from each $\mathcal{L}_i$ to each of its neighbours $\mathcal{L}_j$ in $\mathcal{G}$. Intuitively, $\mathcal{M}_{i,j}$ represents the knowledge that $\mathcal{L}_i$ has of how $\mathcal{L}$ constraints $\mathcal{L}_j$. These messages start propagating from the leaves of $\mathcal{G}$ (recall that $\mathcal{G}$ is a tree) towards its internal nodes, and then back to the leaves as soon as all edges have received a first message. Observe that, by starting at the leaves, the messages necessary to computing $\mathcal{M}_{i,j}$ have always been computed before. Once all messages have been computed, that is two messages per edge, one in each direction, then each component $\mathcal{L}_i$ is combined with all its incoming messages to yield its updated (or reduced) version $\mathcal{L}'_i$ (line 7). This combination of $\mathcal{L}_i$ with the messages received can be seen as a refinement of $\mathcal{L}_i$, removing those words that some of its neighbours knows to be not in $\mathcal{L}$.

Intuitively, $\mathcal{L}'_i = \Pi_{\Lambda_i}(\mathcal{L})$ exactly describes the words of $\mathcal{L}_i$ that are still possible when $\mathcal{L}_i$ is restricted by the environment given by the other languages in the product. The fundamental properties of these updated languages $\mathcal{L}'_i$ are: (1) any word w in $\mathcal{L}$ is such that $w_{|\Lambda_i} \in \mathcal{L}'_i$, and (2) for any word w_i in $\mathcal{L}'_i$ there exists $w \in \mathcal{L}$ such that $w_{|\Lambda_i} = w_i$. Thus, one can then find a w in $\mathcal{L}$ from the $\mathcal{L}'_i$s (if they are non-empty, else it means that $\mathcal{L}$ is empty) using Algorithm 2.

Algorithm 2 Construction of a word of $\mathcal{L} = \mathcal{L}_1||\ldots||\mathcal{L}_n$ from its updated components $\mathcal{L}'_1,\ldots,\mathcal{L}'_n$ obtained by Algorithm 1

```
1: $nexts \leftarrow \{1\}$
2: $W \leftarrow \emptyset$
3: while $nexts \neq \emptyset$ do
4:     extract $i \in nexts$
5:     choose $w_i \in \mathcal{L}'_i||\Pi_{\Lambda_i}(||_{j\in W\cap\mathcal{N}(i)}w_j)$
6:     add $i$ to $W$
7:     for all $j \in \mathcal{N}(i) \setminus W$ do
8:         add $j$ to $nexts$
9:     end for
10: end while
11: return any word $w$ from $||_{i\in W}w_i$
```

In this algorithm, for w_i a word in $\mathcal{L}'_i$ and w_j a word in $\mathcal{L}'_j$, we denote by $w_i||w_j$ the product of the languages $\{w_i\}$ and $\{w_j\}$ respectively defined over Λ_i and Λ_j. This algorithm is in fact close to Algorithm 1: the w_i propagate from an arbitrary root (here $\mathcal{L}_1$) to the leaves of the communication graph considered. The tricky parts are to notice that choosing w_i in line 5 is always possible and that w always exists at line 11. Both these facts are due to the fundamental properties of $\mathcal{L}'_i$ explained above [Fabre and Jezequel 2009].

3.2. An example of message passing for languages

We do not provide a proof of Proposition 3.1 as it would require to introduce a more general notion of message passing algorithms, not relevant to this paper. This proof can be found in [Fabre and Jezequel 2009]. Instead, in order to give some intuition on how and why Algorithms 1 and 2 actually work, we provide sample executions of them.

For that, consider three languages: $\mathcal{L}_1 = \{aa\alpha aa, \alpha aa\alpha, a\alpha a\}$, $\mathcal{L}_2 = \{b\alpha\beta b, bb\alpha\beta\beta, \alpha b\beta b\alpha\alpha, \alpha\alpha\beta bb\beta\}$, and $\mathcal{L}_3 = \{\beta\beta ccc\beta, \beta cc\beta, \beta\beta c\beta, cc\beta\}$. As $\Lambda_1 \cap \Lambda_2 = \{\alpha\}$, $\Lambda_2 \cap \Lambda_3 = \{\beta\}$, and $\Lambda_3 \cap \Lambda_1 = \emptyset$, their interaction graph (thus their unique communication graph) is a line with $\mathcal{L}_2$ in the middle. So Algorithm 1 will converge.

The initialisation step of Algorithm 1 consists in setting $M = \{(1,2),(2,1),(2,3),(3,2)\}$. Four messages have to be computed: $\mathcal{M}_{1,2}$, $\mathcal{M}_{2,1}$, $\mathcal{M}_{2,3}$, and $\mathcal{M}_{3,2}$. $\mathcal{M}_{2,1}$ cannot be computed immediately because it requires $\mathcal{M}_{3,2}$ to be computed first. Similarly $\mathcal{M}_{2,3}$ requires $\mathcal{M}_{1,2}$ to be computed first. Assume one computes $\mathcal{M}_{1,2}$ first: $\mathcal{M}_{1,2} = \Pi_{\Lambda_2}(\mathcal{L}_1) = \Pi_{\{\alpha,b,\beta\}}(\mathcal{L}_1) = \{\alpha, \alpha\alpha\}$.

After that, $M = \{(2,1),(2,3),(3,2)\}$. So, either $\mathcal{M}_{2,3}$ or $\mathcal{M}_{3,2}$ can be computed. Assume one chooses $\mathcal{M}_{2,3}$ first: $\mathcal{M}_{2,3} = \Pi_{\Lambda_3}(\mathcal{M}_{1,2}||\mathcal{L}_2)$. Applying the definition of the product of languages $\mathcal{M}_{1,2}||\mathcal{L}_2 = \{b\alpha\beta b, bb\alpha\beta\beta, \alpha\alpha\beta bb\beta\}$, so $\mathcal{L}_2$ is refined to remove all words containing no α or more than two α (the words that cannot possibly be compatible with the ones in $\mathcal{L}_1$). From that $\mathcal{M}_{2,3} = \{\beta, \beta\beta\}$ by projection on $\Lambda_3 = \{\beta, c\}$.

Now, $M = \{(2,1),(3,2)\}$. So, the next message computed can only be $M_{3,2} = \Pi_{\Lambda_2}(\mathcal{L}_3) = \Pi_{\{\alpha,b,\beta\}}(\mathcal{L}_3) = \{\beta\beta\beta, \beta\beta, \beta\}$. And then, $M_{2,1}$ is finally computed: $M_{2,1} = \Pi_{\Lambda_1}(M_{3,2}||\mathcal{L}_2) = \{\alpha, \alpha\alpha\alpha, \alpha\alpha\}$.

At that point all the messages were computed, and one can now compute $\mathcal{L}'_1$, $\mathcal{L}'_2$ and $\mathcal{L}'_3$: $\mathcal{L}'_1 = \mathcal{L}_1||\mathcal{M}_{2,1} = \mathcal{L}_1$ (nothing can be filtered out of $\mathcal{L}_1$ from the information that $\mathcal{L}_2$ has about the full system); similarly, $\mathcal{L}'_3 = \mathcal{L}_3||\mathcal{M}_{2,3} = \{\beta cc\beta, cc\beta\}$ (the two words with three occurrences of β are filtered out); and finally, $\mathcal{L}'_2 = \mathcal{L}_2||\mathcal{M}_{1,2}||\mathcal{M}_{3,2} = \{b\alpha\beta b, bb\alpha\beta\beta, \alpha\alpha\beta bb\beta\}$ (the word with three occurrences of α is filtered out, no word had no occurrences of β or more than three occurrences of β so $M_{3,2}$ filters no word out). One can now check that Proposition 3.1 holds for these languages.

One can then apply Algorithm 2 to $\mathcal{L}'_1$, $\mathcal{L}'_2$, and $\mathcal{L}'_3$ in order to get a word of $\mathcal{L}_1||\mathcal{L}_2||\mathcal{L}_3$. For that one just selects a word from $\mathcal{L}'_1$, for example $w_1 = a\alpha a$. As the only neighbour of $\mathcal{L}_1$ is $\mathcal{L}_2$ one then selects a word in $\mathcal{L}'_2||\Pi_{\Lambda_2}(w_1) = \{b\alpha\beta b, bb\alpha\beta\beta\}$, for example $w_2 = b\alpha\beta b$. And finally one takes a word in $\mathcal{L}'_3||\Pi_{\Lambda_3}(w_2) = \{cc\beta\}$, the only possibility is $w_3 = cc\beta$. It is clear that these three words can be interleaved in the last step of Algorithm 2, for example as $w = bac\alpha ca\beta b$. Checking that $w \in \mathcal{L}_1||\mathcal{L}_2||\mathcal{L}_3$ concludes this example.

In the rest of this section we explain how Petri nets can be used as an efficient representation of languages in Algorithm 1, and make a link to factored planning.

3.3. Message passing algorithm for Petri nets

In our previous work on factored planning we represented languages by automata. In this paper we use safe Petri nets instead, which are potentially exponentially more compact. For that we use the well-known facts that: (i) the product of labelled Petri nets implements the product of languages (see Proposition 3.2 below); and (ii) it is straightforward to define a projection operation for Petri nets which implements the projection of languages (Proposition 3.4). Notice that in practice only the fact that (i) and (ii) hold for those words ending by the goal label g is used in planning (Corollaries 3.3 and 3.5).

PROPOSITION 3.2 ([CHU 1987]). *For any two labelled Petri nets* $N_1 = (P_1, T_1, F_1, M_1^0, \Lambda_1, \lambda_1)$ *and* $N_2 = (P_2, T_2, F_2, M_2^0, \Lambda_2, \lambda_2)$ *one has* $\mathcal{L}(N_1||N_2) = \mathcal{L}(N_1)||\mathcal{L}(N_2)$.

COROLLARY 3.3. *In particular it holds that* $\mathcal{L}^g(N_1||N_2) = \mathcal{L}^g(N_1)||\mathcal{L}^g(N_2)$.

PROOF. Notice that $\mathcal{L}^g(N_1) = \mathcal{L}(N_1) \cap (\Lambda_1 \setminus \{g\})^*g = \mathcal{L}(N_1) \cap ((\Lambda_1 \cup \Lambda_2) \setminus \{g\})^*g$ and similarly $\mathcal{L}^g(N_2) = \mathcal{L}(N_2) \cap ((\Lambda_1 \cup \Lambda_2) \setminus \{g\})^*g$. From that $\mathcal{L}^g(N_1)||\mathcal{L}^g(N_2) = \mathcal{L}(N_1)||\mathcal{L}(N_2) \cap ((\Lambda_1 \cup \Lambda_2) \setminus \{g\})^*g$. Thus, by definition, $\mathcal{L}^g(N_1||N_2) = \mathcal{L}^g(N_1)||\mathcal{L}^g(N_2)$. □

The projection operation for labelled Petri nets can be defined simply by relabelling some of the transitions by ε (i.e. making them silent). Formally, the *projection* of a labelled Petri net $N = (P, T, F, M^0, \Lambda, \lambda)$ to an alphabet Λ' is the labelled Petri net $\Pi_{\Lambda'}(N) = (P, T, F, M^0, \Lambda', \lambda')$ such that $\lambda'(t) = \lambda(t)$ if $\lambda(t) \in \Lambda'$ and $\lambda'(t) = \varepsilon$ otherwise. Notice that the projection operation preserves safeness. An example is given in Figure 2.

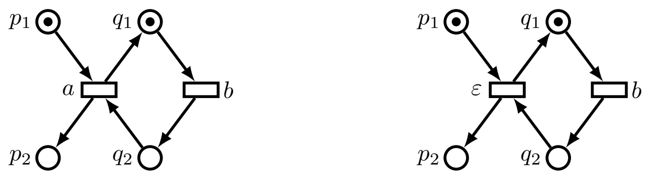


Fig. 2. A Petri net (left) and its projection on the alphabet $\{b\}$ (right).

PROPOSITION 3.4. *For any Petri net* $N = (P, T, F, M^0, \Lambda, \lambda)$, $\mathcal{L}(\Pi_{\Lambda'}(N)) = \Pi_{\Lambda'}(\mathcal{L}(N))$.

COROLLARY 3.5. *In particular, when* $g \in \Lambda'$ *it holds that* $\mathcal{L}^g(\Pi_{\Lambda'}(N)) = \Pi_{\Lambda'}(\mathcal{L}^g(N))$.

PROOF. Notice that $\mathcal{L}^g(\Pi_{\Lambda'}(N)) = \mathcal{L}(\Pi_{\Lambda'}(N)) \cap (\Lambda' \setminus \{g\})^*g$. As the alphabet of $\mathcal{L}(\Pi_{\Lambda'}(N))$ is included in Λ' it follows that: $\mathcal{L}^g(\Pi_{\Lambda'}(N)) = \mathcal{L}(\Pi_{\Lambda'}(N)) \cap (\Lambda' \cup \Lambda \setminus \{g\})^*g$. Then, by Proposition 3.4, $\mathcal{L}^g(\Pi_{\Lambda'}(N)) = \Pi_{\Lambda'}(\mathcal{L}(N)) \cap (\Lambda' \cup \Lambda \setminus \{g\})^*g = \Pi_{\Lambda'}(\mathcal{L}(N) \cap (\Lambda' \cup \Lambda \setminus \{g\})^*g)$. Since the alphabet of $\mathcal{L}(N)$ is included in Λ, $\mathcal{L}^g(\Pi_{\Lambda'}(N)) = \Pi_{\Lambda'}(\mathcal{L}(N) \cap (\Lambda \setminus \{g\})^*g)$. Finally, by definition of $\mathcal{L}^g(N)$, $\mathcal{L}^g(\Pi_{\Lambda'}(N)) = \mathcal{L}^g(N)$. □

Propositions 3.2 and 3.4 allow one to directly apply Algorithm 1 using safe labelled Petri nets to represent languages. That is, from a compound Petri net $N = N_1|| \dots ||N_n$ such that $N_1, \dots, N_n$ lives on a tree one obtains – with local computations only – an updated version N_i' of each component N_i of N with the following property:

$$\begin{aligned}\mathcal{L}(N_i') &= \Pi_{\Lambda_i}(\mathcal{L}(N_1)|| \dots ||\mathcal{L}(N_n)) \\ &= \Pi_{\Lambda_i}(\mathcal{L}(N_1|| \dots ||N_n)) \\ &= \mathcal{L}(\Pi_{\Lambda_i}(N)).\end{aligned}$$

So, as for languages in the previous section, this allows one to compute a word in N by doing only local computations, i.e. computations that only involve some component N_i and its neighbours in the considered communication graph of N. For that, one just computes the N_i's using Algorithm 1, and then applies Algorithm 2 with Petri nets as the representation of languages. This shows that an instance of Algorithm 1 can be used for solving factored planning problems represented as products of Petri nets.

One may question the possibility of the communication graphs of factored planning problems represented by Petri nets to be trees, especially because all the nets share a common goal label g. In fact a label shared by all the components of a problem does not affect its communication graphs: if the interaction graph of $N_1, \ldots, N_n$ is connected then $\mathcal{G} = (\{N_1, \ldots, N_n\}, E)$ is a communication graph of $N_1, \ldots, N_n$ if and only if $\mathcal{G}^g = (\{N_1^g, \ldots, N_n^g\}, E^g)$ with $E^g = \{(N_i^g, N_j^g) \; : \; (N_i, N_j) \in E\}$ is a communication graph of any $N_1^g, \ldots, N_n^g$ where $\forall i, \Lambda_i^g = \Lambda_i \dot{\cup} \{g\}$. This is due to the definition of redundant edges: all the edges (N_i^g, N_j^g) such that $\Lambda_i^g \cap \Lambda_j^g = \{g\}$ can be first removed and then any edge (N_i^g, N_j^g) is redundant if and only if (N_i, N_j) is redundant. Non-connected interaction graphs are not an issue as in this case the considered problem can be split into several completely independent problems (one for each connected component of the interaction graph) and should never be solved as a single problem.

3.4. An example of message passing for Petri nets

In order to show what exactly are components and messages in this context, we now give an example of a part of a run of Algorithm 1 with languages directly represented as Petri nets. For avoiding technicalities in the example we consider prefix-closed languages (so we have no goal transition in our nets). Figure 3 presents the problem we consider. It is in fact the standard dining philosophers model with three philosophers. The corresponding interaction graph is not a tree, but merging some of its nodes (i.e. taking their product) solves this problem. Figure 4 shows the two first messages computed, using the same order as in the previous example (with languages).

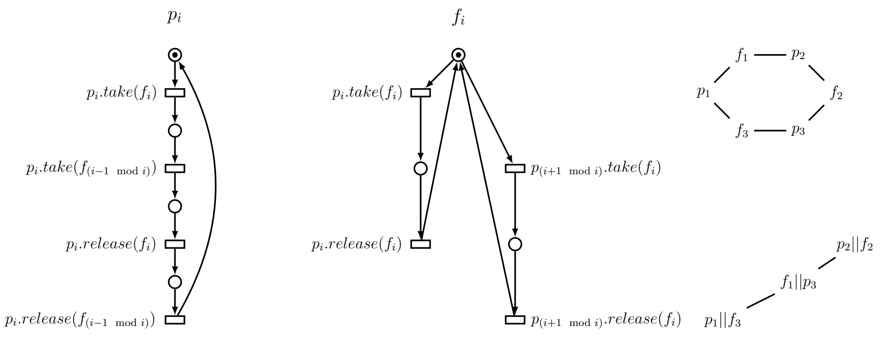


Fig. 3. A philosopher (left), a fork (middle), the interaction graph for three philosophers (top right), and a merging of nodes making it a tree (bottom right).

3.5. Efficiency of the projection

The method presented above for solving factored planning problems exploits both the internal concurrency in each component (to represent local languages like $\mathcal{L}_i, \mathcal{L}_i'$ and the $\mathcal{M}_{i,j}$ by Petri nets) and the global concurrency between components (to represent global plans w as the interleaving of compatible local plans $(w_1, ..., w_n)$). However, due to the rather basic definition of the projection operation for Petri nets, the size of the updated component N_i' is the same as the size of the full factored planning problem $N = N_1 || \ldots || N_n$. And similarly, the messages grow in size along the computations performed by Algorithm 1. This problem can be mitigated by applying language-preserving structural reductions [Vogler and Kangsah 2007; Khomenko et al. 2008] to the intermediate Petri nets computed by the algorithm. This subsection briefly recalls one such method.

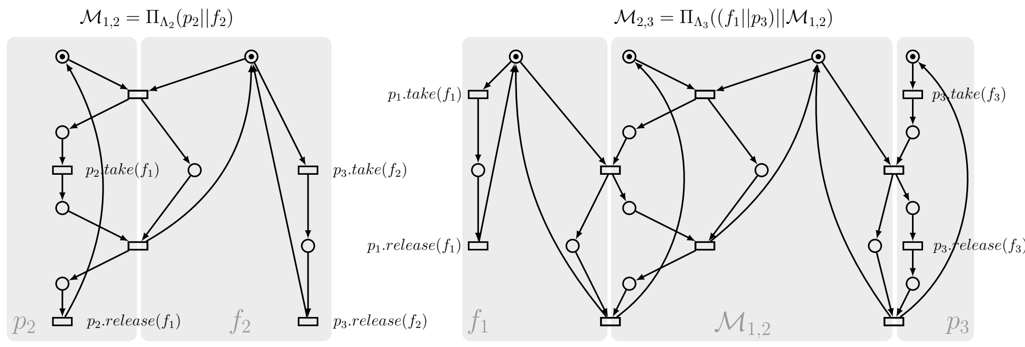


Fig. 4. Message from $N_1 = p_2||f_2$ to $N_2 = f_1||p_3$ (left), and message from N_2 to $N_3 = p_1||f_3$ (right). Transitions with no label are silent. In gray are indications about the origin of each part of the nets.

3.5.1. Contraction of silent transitions. The most important structural reduction we use is *transition contraction* originally proposed in [André 1982] and further developed in [Vogler and Kangsah 2007; Khomenko et al. 2008]. (Indeed, as can be seen in Figure 4, there are many silent transitions in the messages we compute.) We now recall its definition (again, $\star \notin P \cup T$ is an artificial element used for convenience). For a labelled Petri net with silent transitions $N = (P, T, F, M^0, \Lambda, \lambda)$, consider a transition $t \in T$ such that $\lambda(t) = \varepsilon$ and ${}^\bullet t \cap t^\bullet = \emptyset$. The *t-contraction* $N' = (P', T', F', M^{0'}, \Lambda, \lambda')$ of N is defined by:

$$\begin{aligned} P' &= \{(p, \star) \ : \ p \in P \setminus ({}^\bullet t \cup t^\bullet)\} \\ &\quad \cup \{(p, p') \ : \ p \in {}^\bullet t, p' \in t^\bullet\}, \\ T' &= T \setminus \{t\}, \\ F'((p, p'), t') &= F(p, t') + F(p', t') \\ F'(t', (p, p')) &= F(t', p) + F(t', p'), \\ M^{0'}((p, p')) &= M^0(p) + M^0(p'), \\ \lambda' &= \lambda_{|T'}, \end{aligned}$$

where $F(\star, t') = F(t', \star) = 0$ for any p and t', and $M^0(\star) = 0$. It is clear that this contraction operation does not necessarily preserve the language. For this reason it cannot be used directly to build an efficient projection operation. There exist however conditions ensuring language preservation: A t-contraction is said to be *type-1 secure* if $({}^\bullet t)^\bullet \subseteq \{t\}$ and it is said to be *type-2 secure* if ${}^\bullet(t^\bullet) = \{t\}$ and $M^0(p) = 0$ for some $p \in t^\bullet$.

PROPOSITION 3.6 ([VOGLER AND KANGSAH 2007]). *Secure contractions preserve the language of the Petri nets.*

Figure 5 gives an example of a type-1 secure contraction. Notice that this contraction is not type-2 secure because $M^0(q_1) \neq 0$.

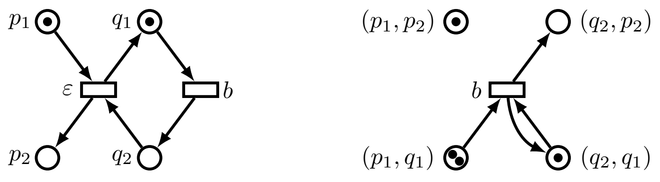


Fig. 5. A Petri net with a silent transition (left) and the same net after contraction of this transition (right).

If N is a safe labelled Petri net then its t-contraction is 2-bounded, but not necessarily safe. As we need to work with safe Petri nets (essentially because solutions to planning problems are found using unfolding techniques [Hickmott et al. 2007]) we are interested only in secure t-contractions preserving safeness. The proposition below gives a cheap sufficiency test for a contraction to be secure and safeness-preserving (it is obtained by combination of the definition of secure given above with the sufficient conditions for safeness-preserving given in [Khomenko et al. 2009]).

PROPOSITION 3.7. *A contraction of a transition t in a net N is secure and safeness-preserving if either*

(*1*) $|t^\bullet| = 1$, ${}^\bullet(t^\bullet) = \{t\}$ *and* $M^0(p) = 0$ *with* $t^\bullet = \{p\}$
(*2*) $|{}^\bullet t| = 1$, ${}^\bullet(t^\bullet) = \{t\}$ *and* $\forall p \in t^\bullet, M^0(p) = 0$*; or*
(*3*) $|{}^\bullet t| = 1$ *and* $({}^\bullet t)^\bullet = \{t\}$*;*

There exists a full characterisation of safeness-preserving contractions as a model checking problem [Khomenko et al. 2009]. However, testing it is much more expensive, so we do not consider it.

3.5.2. Redundant transitions and places. It may be the case (in particular after performing some silent transition contractions) that the Petri net contains *redundant* transitions and/or places. Removing them reduces the size of the net, while preserving its language and safeness [Khomenko et al. 2009].

A transition t in a labelled Petri net $N = (P, T, F, M^0, \Lambda, \lambda)$ is redundant if either

— it is a *loop-only transition*: an ε-transition such that $F(p,t) = F(t,p)$ for each $p \in P$; or
— it is a *duplicate transition*: there is another transition t' such that $\lambda(t) = \lambda(t')$, and $F(p,t) = F(p,t')$ and $F(t,p) = F(t',p)$ for each $p \in P$.

Examples of such transitions are given in Figure 6.

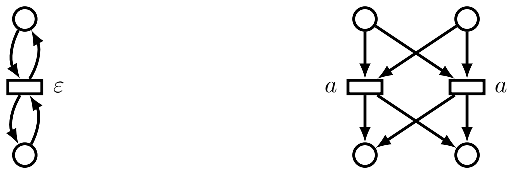


Fig. 6. A loop-only transition (left) and two duplicate transitions (right).

Redundant places of a Petri net can be characterised by a set of linear equations [Vogler and Kangsah 2007] that we do not describe here. Examples of redundant places are:

— *loop-only places*: p is a loop-only place if $F(t,p) = F(p,t)$ for all t and $M^0(p) > 0$;
— *duplicate places*: p is a duplicate of q if $M^0(p) = M^0(q)$, $F(t,p) = F(t,q)$ and $F(p,t) = F(q,t)$ for all t.

Examples of such places are given in Figure 7.

3.5.3. Algorithmic description of the suggested projection operation. Using the reductions described above we implement the projection operation as a re-labelling of the corresponding transitions by ε as described in Section 3.3, followed by (secure, safeness preserving) transition contractions and redundant places/transitions removing while it is possible.

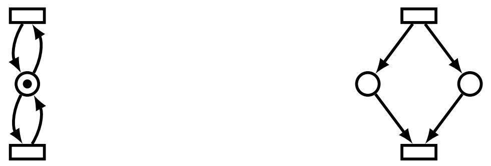


Fig. 7. A loop-only place (left) and two duplicate places (right).

Note that there is no guarantee that all the silent transitions are removed from a Petri net. Moreover, depending on the order of the transition contractions and on the order of the redundant places and transitions removals, the resulting nets may be different. Some guidelines about which silent transitions should be removed first are given in [Khomenko et al. 2009].

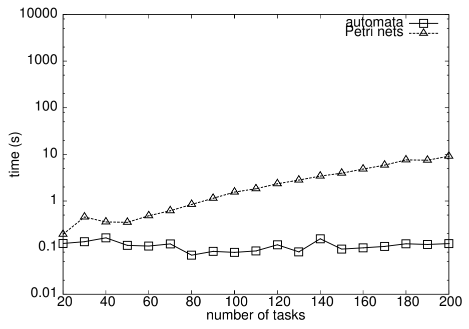
4. EXPERIMENTAL EVALUATION

In order to show the practical interest of replacing automata by Petri nets in the message passing algorithms we compared these two approaches on benchmarks. We focused on the analysis of Algorithm 1 because it has been previously shown to be, by far, the bottleneck of the approach [Jezequel 2012]. For that we used Corbett's benchmarks [Corbett 1996] (they are not actual planning problems and we did not add particular goals to them, assuming that we were interested in finding words belonging to their prefix-closed languages). Among these we selected the ones suitable to our algorithm, that is the ones such that increasing the size of the problem increases the number of components rather than their size. This gave us five problems. They are not all living on trees so we had to merge some components (i.e. replace them by their product) in order to come up with trees and be able to run our algorithm (we did it by hand, but it could be automated using tree-decomposition techniques [Bodlaender 1993]). Notice that the necessity of merging some components is an argument in favour of the use of Petri nets as there is usually local concurrency inside the new components obtained after merging (consider e.g. $p_3||f_1$ in the example of Figure 3: there is no interaction at all between p_3 and f_1, so any action of p_3 is concurrent with any action of f_1). We first describe the five problems we considered and explain how we made each of them live on a tree. After that we present and comment our experimental results.

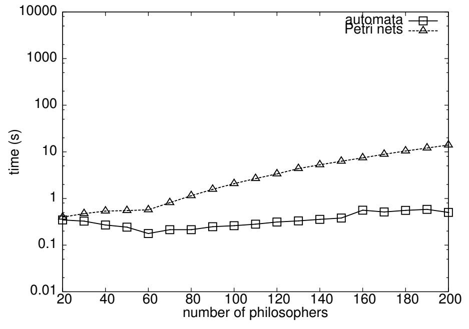
4.1. Presentation of the problems

Milner's cyclic scheduler. A set of n schedulers (numbered from 0 to $n-1$) are organised in a circle. They have to activate tasks on a set of n customers (one for each scheduler) in the cyclic order: customer i's task must have started for the k^{th} time before customer $i+1$'s task starts for the k^{th} time. Each customer is a component, as well as each scheduler. Customer i interacts only with scheduler i while a scheduler interacts with its two neighbour schedulers. The interaction graph of this system is thus not a tree. We first make it a circle by merging each customer with its scheduler. After that we make it a tree (in fact a line) by merging the component i (customer i and scheduler i) with component $n-i-1$.

Divide and conquer computation. A divide and conquer computation using a fork/join principle. A bounded number n of possible tasks is assumed. Each task, when activated, chooses (nondeterministically) to ”divide” the problem by forking (i.e. by activating the next task) and then doing a small computation, or to ”conquer” it by doing a bigger computation. Initially the first task is activated. The last task cannot fork. Each task is a component and their interaction graph is a line: each task interacts (by forking) with the next task and (by joining) with the previous task.



(a) Divide and conquer



(b) Dining philosophers

Fig. 8. Runtimes of Algorithm 1.

Dining philosophers. The classical dining philosophers problem where n philosophers are around a table with one fork between each two philosophers. Each philosopher can perform four actions in a predetermined order: take the fork at its left, take the fork at its right, release the left fork, release the right fork. The components are the forks and the philosophers. Each philosopher shares actions with the two forks he can take, so the interaction graph of this problem is a circle. To make a tree from it we simply merge each philosopher with a fork as follows: philosopher 1 with fork n, philosopher 2 with fork $n - 1$, and so on.

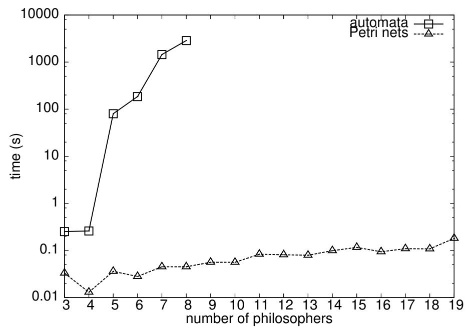
Dining philosophers with dictionary. The same problem as dining philosophers except that the philosophers also pass a dictionary around the table, preventing the philosopher holding it from taking forks. This changes the interaction graph as each philosopher now interacts with his two neighbour philosophers. To make it a tree we merge each philosopher with the corresponding fork (philosopher i with fork i) and then merge these new components as in the case of Milner's scheduler.

Mutual exclusion protocol (Token ring mutex). A standard mutual exclusion protocol in which n users (each one associated with a different server) access a shared resource without conflict by passing a token around a circle formed by the servers (the server possessing the token enables access to the resource for its customer). Each user as well as each server is a component. User i interacts with server i and each server interacts with the server before it and the server after it on the circle (by passing the token). The interaction graph of this system is not a tree. We merge each user with the corresponding server (user i with server i), making the interaction graph a circle. Then we use the same construction as in the case of Milner's scheduler in order to make it a tree.

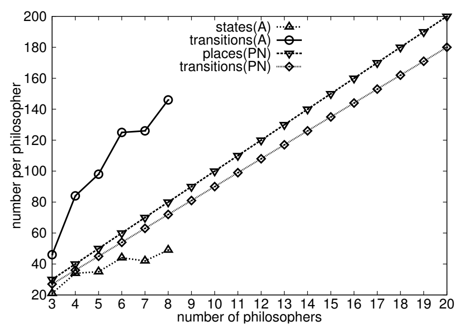
4.2. Experimental results

We ran the message passing algorithm using a representation of the components as automata and as Petri nets. All our experiments were run on the same computer (Intel Core i5 processor, 8GB of memory) with a time limit of 50 minutes. Our objective was to compare the runtimes of both versions of Algorithm 1, and in particular to see if they scale up well as problem sizes increase (a comparison of the automata based version with other planning tools can be found in [Jezequel 2012]).

The results obtained for the divide and conquer computation and for the dining philosophers problem are presented in Figures 8a and 8b respectively. For these two problems, the approach using automata scales up better than the approach using Petri nets. In order to explain this difference we looked at the size of the automata and of

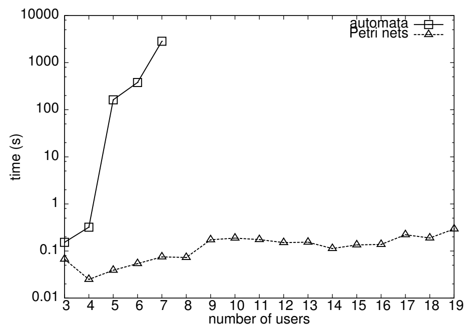


(a) Runtimes

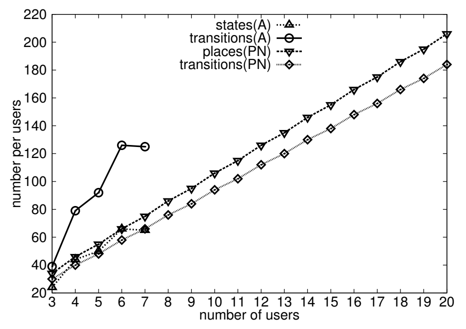


(b) Size of outputs

Fig. 9. Philosophers with dictionary



(a) Runtimes



(b) Size of outputs

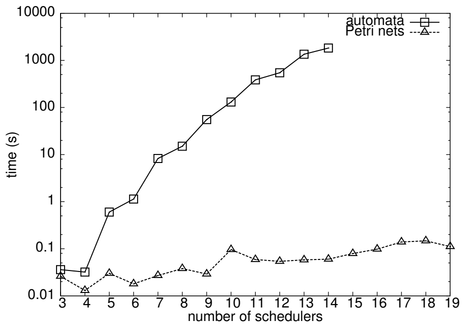
Fig. 10. Token ring mutex

the Petri nets involved in the computations. It appeared that the size of the automata was not depending on the number of components. However, the size of the Petri nets was growing with the size of the problems. Looking more closely to these Petri nets we noticed that they were containing mostly silent transitions. Implementing more size reduction operations, in particular the ones based on unfolding techniques, may solve this issue.

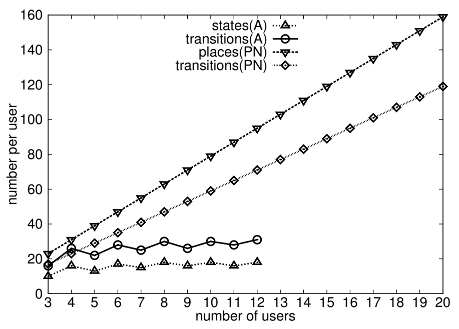
The experimental results for the dining philosophers with a dictionary, the mutual exclusion protocol on a ring, and Milner's cyclic scheduler are shown in Figures 9a, 10a and 11a, respectively; see also Figures 16c, 16d and 16e, respectively, for larger instances of these benchmarks. On these three problems the Petri nets approach scales up far better than the automata approach. In fact, only very small instances of these three problems can be solved using automata.

To explain the reasons behind the efficiency of the Petri nets approach (compared with the automata approach) on these last three problems we plotted the sizes of the final objects (automata or Petri nets) computed by Algorithm 1 in Figures 9b, 10b, and 11b. For the first two benchmarks one can observe that the sizes of the automata are growing much faster than those of Petri nets. This can be explained by the necessity to represent explicitly a large number of action interleavings in the automata, whereas Petri nets represent interleavings implicitly.

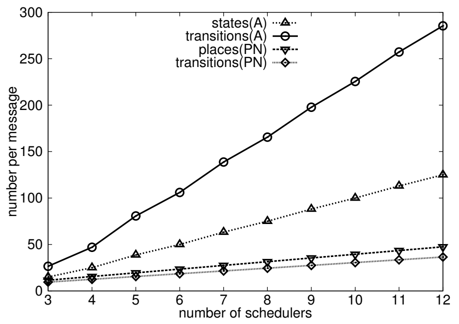
The case of Milner's cyclic scheduler cannot be explained directly from the sizes of the final objects computed. In fact it appears that with the automata approach these sizes are almost stable. For this particular case we thus looked at the average sizes of the intermediate objects (messages) computed during a run of Algorithm 1, which



(a) Runtimes



(b) Size of outputs



(c) Average size of messages

Fig. 11. Milner's cyclic scheduler

are shown in Figure 11c. They are growing much slower with the proposed Petri nets approach.

5. TOWARD COST-OPTIMAL PLANNING

This section shows how the previous factored planning approach can be adapted to cost-optimal planning. It first defines formally the cost-optimal factored planning problem in terms of weighted Petri nets. Next section shows that the central notions of transition contraction and of redundant transition/place removal can be both extended to the setting of weighted Petri nets in some particular cases.

5.1. Cost-optimal planning and weighted Petri nets

In cost-optimal planning the objective is not only to find an execution leading to the goal, but to find a cheapest one. This notion of a cheapest sequence can be defined by the means of costs associated with the transitions of a Petri net.

A *weighted* labelled Petri net is a tuple $(P, T, F, M^0, \Lambda, \lambda, c)$ where $(P, T, F, M^0, \Lambda, \lambda)$ is a labelled Petri net and $c : T \to \mathbb{R}_{\geq 0}$ is a cost-function on transitions. In such a Petri net, each execution $o = t_1 \dots t_n$ has a cost $c(o) = c(t_1) + \cdots + c(t_n)$. The (weighted) language of a weighted Petri net is then:

$$\mathcal{L}(N) = \{(\lambda(o), c) \mid o \in \langle M^0 \rangle, c = \min_{o' \in \langle M^0 \rangle, \lambda(o') = \lambda(o)} c(o')\}.$$

A cost-optimal planning problem is then defined as a pair (N, g) where N is a weighted labelled Petri net and g is a goal label. One has to find an execution $o = t_1 \dots t_n$ from the initial marking of N, such that g appears exactly once at the end of the labelling of o and such that the cost of o is minimal among all similar executions in N (i.e. the ones ending by a transition labelled by g).

5.2. Product of weighted Petri nets

As for factored planning problems, cost-optimal planning problems are defined using a notion of product of weighted labelled Petri nets.

The *product* $N_1||N_2 = (P, T, F, M^0, \Lambda, \lambda, c)$ of two weighted labelled Petri nets is defined as the product of the underlying labelled Petri nets, by assigning to the transitions resulting from a fusion the sum of costs of the original transitions (the transitions that did not participate in a fusion retain their original cost.) An example is given in Figure 12.

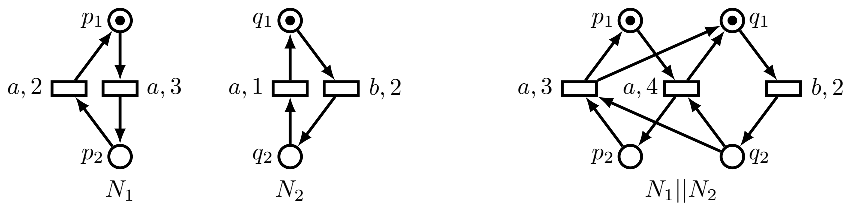


Fig. 12. Two weighted Petri nets (left) and their product (right).

A *cost-optimal factored planning problem* is then defined as a tuple $N = (N_1, \dots, N_n)$ of cost-optimal planning problems $((P_i, T_i, F_i, M_i^0, \Lambda_i, \lambda_i, c_i), g)$. One has to find a cost-optimal solution to the problem $(N_1|| \dots ||N_n, g)$ without computing the full product.

5.3. Message passing for cost-optimal factored planning

The message passing algorithms can be used on weighted languages [Fabre and Jezequel 2009]. For that, the projection of a weighted language $\mathcal{L}$ (defined over Λ) to a sub-alphabet Λ' is:

$$\Pi_{\Lambda'}(\mathcal{L}) = \{(w_{|\Lambda'}, c) \ : \ (w, c) \in \mathcal{L}, c = \min_{\substack{(w',c') \in \mathcal{L} \\ w'_{|\Lambda'} = w_{|\Lambda'}}} c'\},$$

and the product of $\mathcal{L}_1$ and $\mathcal{L}_2$ (defined over Λ_1 and Λ_2 respectively) is:

$$\mathcal{L}_1||\mathcal{L}_2 = \{(w, c) \ : \ w \in \Pi^{-1}_{\Lambda_1 \cup \Lambda_2}(\bar{\mathcal{L}}_1) \cap \Pi^{-1}_{\Lambda_1 \cup \Lambda_2}(\bar{\mathcal{L}}_2),$$
$$c = c_1 + c_2 \textit{ with } (w_{|\Lambda_1}, c_1) \in \mathcal{L}_1, (w_{|\Lambda_2}, c_2) \in \mathcal{L}_2\},$$

where $\bar{\mathcal{L}}$ is the *support* of the weighted language $\mathcal{L}$:

$$\bar{\mathcal{L}} = \{w \ : \ \exists (w, c) \in \mathcal{L}\}.$$

Exactly as in the case of languages without weights one gets a method to find a cost-optimal word into a compound weighted language $\mathcal{L} = \mathcal{L}_1|| \dots ||\mathcal{L}_n$ without computing $\mathcal{L}$ as soon as $\mathcal{L}_1, \dots, \mathcal{L}_n$ lives on a tree. From the updated components $\mathcal{L}'_i$ obtained by Algorithm 1 one can extract this cost-optimal word of $\mathcal{L}$ using Algorithm 2, just replacing each selection of a word by the selection of a locally cost-optimal word. This is due to the fact that:

(1) any cost-optimal word w with cost c in $\mathcal{L}$ is such that $w_{|\Lambda_i}$ is a cost-optimal word with the same cost c in $\mathcal{L}'_i = \Pi_{\Lambda_i}(\mathcal{L})$; and
(2) for any cost-optimal word w_i in $\mathcal{L}'_i = \Pi_{\Lambda_i}(\mathcal{L})$ with cost c there exists a word w in $\mathcal{L}$ with the same cost c, which is also cost-optimal and satisfies $w_i = w_{|\Lambda_i}$.

Exactly as before we can show that the product of weighted Petri nets implements the product of weighted languages.

PROPOSITION 5.1. *For any two weighted labelled Petri nets N_1 and N_2 one has $\mathcal{L}(N_1||N_2) = \mathcal{L}(N_1)||\mathcal{L}(N_2)$.*

PROOF. From Proposition 3.2 one directly gets that $\mathcal{L}(N_1||N_2)$ and $\mathcal{L}(N_1)||\mathcal{L}(N_2)$ have the same support. It remains to prove that for any $(w, c) \in \mathcal{L}(N_1||N_2)$ the corresponding $(w, c') \in \mathcal{L}(N_1)||\mathcal{L}(N_2)$ is such that $c = c'$. For that assume $c < c'$ (resp. $c > c'$) the construction of the proof of $\subseteq$ (resp. $\supseteq$) for Proposition 3.2 can be applied to construct an execution o in $\mathcal{L}(N_1)||\mathcal{L}(N_2)$ (resp. $\mathcal{L}(N_1||N_2)$) such that the labelling of o is w and its cost is c (resp. c'), which is a contradiction with the fact that $(w, c') \in \mathcal{L}(N_1)||\mathcal{L}(N_2)$ (resp. $(w, c) \in \mathcal{L}(N_1||N_2)$) because $c' > c$ (resp. $c > c'$) is not the minimal cost for w in this net. □

Similarly as for non-weighted Petri nets, the projection of a weighted labelled Petri net $N = (P, T, F, M^0, \Lambda, \lambda, c)$ on an alphabet Λ' is simply the weighted labelled Petri net $\Pi_{\Lambda'}(N) = (P, T, F, M^0, \Lambda', \lambda', c)$ where

$$\begin{aligned} \lambda'(t) &= \lambda(t) \text{ if } \lambda(t) \in \Lambda' \\ &= \varepsilon \text{ else.} \end{aligned}$$

PROPOSITION 5.2. *For any weighted labelled Petri net N and any alphabet Λ', one has $\mathcal{L}(\Pi_{\Lambda'}(N)) = \Pi_{\Lambda'}(\mathcal{L}(N))$.*

PROOF. The fact that $\mathcal{L}(\Pi_{\Lambda'}(N))$ and $\Pi_{\Lambda'}(\mathcal{L}(N))$ have the same support comes from Proposition 3.4. Observe that only the label of a transition may change during the projection, while its cost remains the same. Hence for any $(w, c) \in \mathcal{L}(\Pi_{\Lambda'}(N))$ the corresponding $(w, c') \in \Pi_{\Lambda'}(\mathcal{L}(N))$ is such that $c = c'$, which concludes the proof. □

This allows us to use weighted Petri nets in our message passing algorithm instead of weighted languages.

6. EFFICIENT PROJECTION OF WEIGHTED PETRI NETS

In order to make the above message passing algorithm of practical interest in presence of weights, we examine when the size reduction operations (transition contraction, redundant places and transitions removal) can be applied to weighted Petri nets while preserving their weighted languages.

6.1. Removing redundant transitions and places

We first show that redundant transitions and places can be safely removed. One only has to take care to remove the right transitions among redundant ones.

PROPOSITION 6.1. *Removing a loop-only transition from a weighted Petri net does not change its weighted language.*

PROOF. Consider any execution $o = t_1 \dots t_n$ of a Petri net N such that for some $i \leq n$ the transition t_i is loop-only. It is straightforward that $o' = t_1 \dots t_{i-1} t_{i+1} \dots t_n$ is also an execution of N. As t_i is a silent transition one gets $\lambda(o) = \lambda(o')$. So the word $\lambda(o)$ still belongs to the language of P after the removal of t_i.

Suppose that the cost of $\lambda(o)$ in P is the cost of o, that is, there exists no execution o'' such that $\lambda(o'') = \lambda(o)$ and $c(o'') < c(o)$. Two cases are possible. If $c(t_i) > 0$ one can take $o'' = o'$ and contradict the above supposition. One can thus remove t_i from P without changing the cost of $\lambda(o)$ in the weighted language of P. If $c(t_i) = 0$ one has $c(o') = c(o)$ and so t_i can also be removed from P without changing the cost of $\lambda(o)$ (it is achieved by o').

Thus, removing t_i from P does not change the words belonging to its language, nor their cost. This concludes the proof. □

PROPOSITION 6.2. *Let t and t' be duplicate transitions of a weighted Petri net P such that $c(t') \leq c(t)$. Removing t from P does not change its weighted language.*

PROOF. Consider any execution $o = t_1 \dots t_n$ of a Petri net N such that for some $i \leq n$ the transition $t_i = t$ is a duplicate of some transition t' with a smaller cost. It is straightforward that $o' = t_1 \dots t_{i-1} t' t_{i+1} \dots t_n$ is also an occurrence sequence of N. One has $\lambda(t) = \lambda(t')$, so the word $\lambda(o) = \lambda(o')$ still belongs to the language of P after removal of t.

Suppose that the cost of $\lambda(o)$ in P is the cost of o, that is, there exists no execution o'' such that $\lambda(o'') = \lambda(o)$ and $c(o'') < c(o)$. Two cases are possible. If $c(t) > c(t')$ one can take $o'' = o'$ and contradict the above supposition. One can thus remove t from P without changing the cost of $\lambda(o)$ in the weighted language of P. If $c(t) = c(t')$ one has $c(o') = c(o)$ and so t can also be removed from P without changing the cost of $\lambda(o)$ (it is achieved by o').

Thus, removing t from P does not change the words belonging to its language, nor their cost. This concludes the proof. □

PROPOSITION 6.3. *Any redundant place can be removed from a weighted Petri net without changing its weighted language.*

PROOF. By definition, a redundant place of a Petri net P is a place that can be removed without changing the possible occurrence sequences of P. So any sequence of transition that can be fired in P can also be fired in the net P' obtained by removing a redundant place from P, and conversely. So, for a given word w, the sets of occurrence sequences o such that $\lambda(o) = w$ are the same in P and P'. In a weighted Petri net, the cost of w is the minimal cost among these o such that $\lambda(o) = w$. Removing any redundant place thus preserves this cost. □

6.2. Contraction of silent transitions

We now consider the possibility to contract silent transitions, as in the non-weighted case. In general not all transitions that could be safely contracted in a non-weighted Petri net can still be safely contracted in a weighted Petri net. This is in particular the case as we want to keep the contraction operation local: the contraction of a silent transition should require only to look at and modify neighbour transitions and places. In the following we first define the transition contraction operation for weighted Petri nets. Then, we remark that transitions with cost 0 can be contracted as in the weighted case. And finally we show how transition contraction can be performed in each case of Proposition 3.7 (i.e. when transitions can be safely contracted in the non-weighted case), this sometimes involve to strengthen the conditions for contraction.

For a weighted labelled Petri net with silent transitions $N = (P, T, F, M^0, \Lambda, \lambda, c)$, consider a transition $t \in T$ such that $\lambda(t) = \varepsilon$ and ${}^\bullet t \cap t^\bullet = \emptyset$, consider also a set of transitions $T(t) \subseteq T \setminus \{t\}$. The (weighted) $t, T(t)$-*contraction* $N' = (P', T', F', M^{0\prime}, \Lambda, \lambda', c')$ of N is such that $(P', T', F', M^{0\prime}, \Lambda, \lambda')$ is the t-contraction of $(P, T, F, M^0, \Lambda, \lambda)$ and:

$$\forall t' \in T', c'(t') = \begin{cases} c(t') + c(t) & \text{if } t' \in T(t) \\ c(t') & \text{else} \end{cases}$$

PROPOSITION 6.4. *For a weighted Petri net $N = (P, T, F, M^0, \Lambda, \lambda, c)$ and a silent transition $t \in T$ such that $c(t) = 0$, if the t-contraction of $(P, T, F, M^0, \Lambda, \lambda)$ has the same language as $(P, T, F, M^0, \Lambda, \lambda)$, then the weighted $t, T(t)$-contraction of N has the same weighted language as N for any $T(t)$.*

PROOF. As $c(t) = 0$ one can consider without loss of generality that $T(t)$ is empty. Assume, also without loss of generality, that t is the only silent-transition and each other transition t' has for label $(t', c(t'))$. This way, any occurrence sequence corresponding to a word w in N has for cost the cost of w. And the cost of $w = (t_1, c(t_1)) \dots (t_k, c(t_k))$ in N is obtained directly from w as $c(t_1) + \dots + c(t_k)$. As $(P, T, F, M^0, \Lambda, \lambda)$ and its t-contraction have the same language it comes directly that N and its $t, T(t)$-contraction have the same weighted language. □

From now on we consider only silent transitions t such that $c(t) > 0$. For an execution o and a set T' of transitions, denote by $|o|_{T'}$ the number of transitions from T' in o. The idea behind the use of the set $T(t)$ in the definition of the $t, T(t)$-contraction of a weighted Petri net is that choosing $T(t)$ such that $|o|_{T(t)} = |o|_{\{t\}}$ for any execution o allows to preserve weighted languages, as stated by the following proposition.

PROPOSITION 6.5. *For a weighted Petri net $N = (P, T, F, M^0, \Lambda, \lambda, c)$, a silent transition $t \in T$, and a set $T(t) \in T \setminus \{t\}$ such that for any execution o of N one has $|o|_{T(t)} = |o|_{\{t\}}$, if the t-contraction of $(P, T, F, M^0, \Lambda, \lambda)$ has the same language as $(P, T, F, M^0, \Lambda, \lambda)$, then the weighted $t, T(t)$-contraction of N has the same weighted language as N.*

However, such a $T(t)$ does not exist in general, as one can notice in Figure 13: the execution t_1 does not contain the silent transition t_2, so $t_1 \notin T(t_2)$, and then necessarily $|t_1 t_2|_{T(t)} \neq |t_1 t_2|_{\{t\}}$.

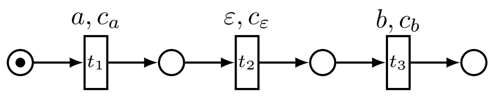


Fig. 13. A Petri net with weights on transitions.

It is possible to relax the constraints on the choice of $T(t)$. Indeed, in a weighted language, only the best costs for words are considered, so it is sufficient to ensure that for any word w the executions o achieving the best cost for this word are such that $|o|_{T(t)} = |o|_{\{t\}}$. Considering the net of Figure 13 one can then take $T(t_2) = \{t_3\}$. Indeed, the word a is obtained with best cost from the execution t_1 and the word ab from the execution $t_1 t_2 t_3$, so the execution $t_1 t_2$ never has to be considered. This is formalized by the following proposition.

PROPOSITION 6.6. *For a weighted Petri net $N = (P, T, F, M^0, \Lambda, \lambda, c)$, a silent transition $t \in T$, and a set $T(t) \in T \setminus \{t\}$ such that for any execution $o = \arg\min_{\lambda(o')=w} c(o')$ with w a word of N one has $|o|_{T(t)} = |o|_{\{t\}}$, if the t-contraction of $(P, T, F, M^0, \Lambda, \lambda)$ has the same language as $(P, T, F, M^0, \Lambda, \lambda)$, then the weighted $t, T(t)$-contraction of N has the same weighted language as N.*

PROOF. As before consider, without loss of generality, that each non-silent transition t_i has for label $(t_i, c(t_i))$ and that t is the only silent transition in N.

Let $o = t_1 \dots t_k$ be an execution of N such that $o = \arg\min_{\lambda(o')=w} c(o')$ with w a word of N. One has $w = (t_{i_1}, c(t_{i_1})) \dots (t_{i_\ell}, c(t_{i_\ell}))$ where $t_{i_1} \dots t_{i_\ell}$ is the longest subsequence of o not containing t. As the t-contraction of $(P, T, F, M^0, \Lambda, \lambda)$ has the same language as $(P, T, F, M^0, \Lambda, \lambda)$, w is a word of this t-contraction and so $o' = t_{i_1} \dots t_{i_\ell}$ is an execution of it (by construction of the labelling of transitions). Moreover, by construction of $T(t)$, one has $|o'|_{T(t)} = |o|_{T(t)} = |o|_{\{t\}}$. So the cost of o' in the $t, T(t)$-contraction of N is the same as the cost of o in N: it is obtained as $c'(t_{i_1}) + \dots + c'(t_{i_\ell}) = c(t_{i_1}) + \dots + c(t_{i_\ell}) + |o|_{T(t)} * c(t)$.

The $t, T(t)$-contraction of N contains no silent transition, so the cost of w in it is the cost of o' in it (o' is the only execution such that $\lambda'(o') = w$). We have shown that any weighted word of N is also a weighted word of its $t, T(t)$-contraction.

Let w be a word of the $t, T(t)$-contraction of N. Let $o = t_1 \dots t_k$ be an execution of the $t, T(t)$-contraction of N such that $o = \arg\min_{\lambda'(o')=w} c'(o')$. As t-contraction preserves language, w is also a word in N. Let o' be an execution of N such that $o' = \arg\min_{\lambda(o'')=w} c(o'')$. Due to how we defined the labelling of the transitions, the only difference between o and o' is that o' may contain occurrences of the silent transition t. However, by definition of $T(t)$ one has $|o'|_{T(t)} = |o'|_{\{t\}}$ and by construction of o' one has $|o'|_{T(t)} = |o|_{T(t)}$. And so, $c(o') = c(t_1) + \cdots + c(t_k) + |o'|_{\{t\}} * c(t) = c(t_1) + \cdots + c(t_k) + |o|_{T(t)} * c(t) = c'(t_1) + \cdots + c'(t_k) = c'(o)$. This proves that any weighted word of the $t, T(t)$-contraction of N is also a weighted word of N.

This concludes the proof, and also proves Proposition 6.5. Indeed, the set of all executions o of N such that $|o|_{T(t)} = |o|\{t\}$ contains the set of all executions $o = \arg\min_{\lambda(o')=w} c(o')$ with w a word of N such that $|o|_{T(t)} = |o|_{\{t\}}$. $\square$

In the particular case of Petri nets representing planning problems the constraints to choose $T(t)$ can be relaxed a bit more. Indeed, one is not interested in all the words in the weighted language of such weighted Petri nets but only in the ones which are plans, that is the ones ending by the special letter g. This is formalized by the following proposition, and it will be of interest in one of the cases of transition contraction described below.

PROPOSITION 6.7. *For a weighted Petri net $N = (P, T, F, M^0, \Lambda, \lambda, c)$, a silent transition $t \in T$, and a set $T(t) \in T \setminus \{t\}$ such that for any execution $o = \arg\min_{\lambda(o')=w} c(o')$ with w a word of N ending by g one has $|o|_{T(t)} = |o|_{\{t\}}$, if the t-contraction of $(P, T, F, M^0, \Lambda, \lambda)$ has the same language as $(P, T, F, M^0, \Lambda, \lambda)$, then the weighted $t, T(t)$-contraction of N has the same set of weighted plans as N.*

The proof of this proposition is the same as the proof of Proposition 6.6, only considering plans instead of words.

With that in mind we look at the different cases of secure and safeness preserving transition contractions given in Proposition 3.7. In each case we propose a way to compute a valid $T(t)$. This sometimes implies to make the conditions for contraction more restrictive.

PROPOSITION 6.8. *If t is a silent transition of a weighted Petri net N satisfying case 1 of Proposition 3.7: $|t^\bullet| = 1$, ${}^\bullet(t^\bullet) = \{t\}$ and $M^0(p) = 0$ with $t^\bullet = \{p\}$, then $T(t) = (t^\bullet)^\bullet$ is such that $|o|_{T(t)} = |o|_{\{t\}}$ for any execution $o = \arg\min_{\lambda(o')=w} c(o')$ with w a word of N.*

PROOF. In this case, $t^\bullet = \{p\}$ with $M^0(p) = 0$ and ${}^\bullet(t^\bullet) = \{t\}$ so the transitions in $(t^\bullet)^\bullet$ can only be fired after a firing of t (they are not initially enabled and they can only be enabled by t) and at most one of them can be fired after each firing of t (because $|t^\bullet| = 1$ and ${}^\bullet(t^\bullet) = \{t\}$). Thus for any execution o one has $|o|_{T(t)} \leq |o|_{\{t\}}$. Moreover, for any execution o achieving the minimal cost for the word $\lambda(o)$ one has $|o|_{T(t)} \geq |o|_{\{t\}}$ (else one occurrence of the silent transition t can be removed from o without changing the obtained word, and thus o did not achieve the minimal cost for $\lambda(o)$). $\square$

The case of a transition t satisfying case 2 in Proposition 3.7 is close to the previous case. Indeed, for the same reasons as above the transitions in $(t^\bullet)^\bullet$ can only be fired after firing t and for any execution o achieving the minimal cost for a word one has $|o|_{(t^\bullet)^\bullet} \geq |o|_{\{t\}}$. However, in general, it can be the case that $|o|_{(t^\bullet)^\bullet} > |o|_{\{t\}}$, because $|t^\bullet| > 1$ and so some transitions from $(t^\bullet)^\bullet$ may be fired concurrently (as an example consider t_3 and the dashed transition in Figure 14) or sequentially without having to fire t another time.

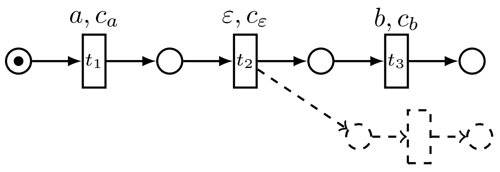


Fig. 14. Another Petri net with weights on transitions.

To solve this issue we limit these contractions to a simple case where only one transition in $(t^\bullet)^\bullet$ can be fired after each occurrence of t. This is formalized by the following proposition.

PROPOSITION 6.9. *If t is a silent transition of a weighted Petri net N satisfying case 2 of Proposition 3.7: $|{}^\bullet t| = 1$, ${}^\bullet(t^\bullet) = \{t\}$ and $\forall p \in t^\bullet, M^0(p) = 0$, if moreover $\forall t', t'' \in (t^\bullet)^\bullet, {}^\bullet t' \cap {}^\bullet t'' \cap t^\bullet \neq \emptyset$, then $T(t) = (t^\bullet)^\bullet$ is such that $|o|_{T(t)} = |o|_{\{t\}}$ for any execution $o = \arg\min_{\lambda(o')=w} c(o')$ with w a word of N.*

PROOF. The proof that for any execution o achieving the minimal cost for a word one has $|o|_{(t^\bullet)^\bullet} \geq |o|_{\{t\}}$ is the same as for Proposition 6.8. It remains to prove that for any execution o achieving the minimal cost for a word one has $|o|_{(t^\bullet)^\bullet} \leq |o|_{\{t\}}$. First remark that because $\forall p \in t^\bullet, M^0(p) = 0$ and ${}^\bullet(t^\bullet) = \{t\}$, it is not possible to fire a transition from $(t^\bullet)^\bullet$ before having fired t at least once. Then remark that after each firing of t no more than one transition from $(t^\bullet)^\bullet$ can be fired. This is because $\forall t', t'' \in (t^\bullet)^\bullet, {}^\bullet t' \cap {}^\bullet t'' \cap t^\bullet \neq \emptyset$, so firing any t' from $(t^\bullet)^\bullet$ removes a token from a predecessor of each of the $t'' \in (t^\bullet)^\bullet$ which is also a successor of t (this token can only be put back by t as ${}^\bullet(t^\bullet) = \{t\}$). As we consider only safe Petri nets, removing a token from a predecessor of a transition always disables this transition. So by firing a transition in $(t^\bullet)^\bullet$ one disables all transitions in $(t^\bullet)^\bullet$ until a new firing of t. This ensures that $|o|_{(t^\bullet)^\bullet} \leq |o|_{\{t\}}$ and concludes the proof. □

In the last case (contraction of t satisfying case 3 in Proposition 3.7) one cannot take $T(t) = (t^\bullet)^\bullet$, as it is possible that ${}^\bullet(t^\bullet) \supset \{t\}$, and so the transitions from $(t^\bullet)^\bullet$ may be enabled without firing t before. Denote by p the only place in ${}^\bullet t$. Assume it is such that $M^0(p) = 0$. Then, $T(t) = {}^\bullet p$ is a reasonable candidate. Indeed, t can only be enabled by the firing of some transition in ${}^\bullet p$. However, there is no guarantee that the firing of a transition $t' \in {}^\bullet p$ enforces to fire t afterwards (as an example t_1 is useful for firing the dotted transition in Figure 15), which would be necessary if $T(t) = {}^\bullet p$.

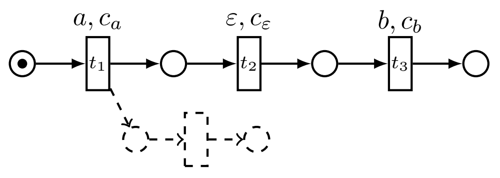


Fig. 15. Another Petri net with weights on transitions.

In the particular case of planning, however, one is not interested in the full language of a Petri net, but only in those words finishing by the special label g. In this context it is possible to ensure that t will always be fired after such t' in an execution achieving the minimal cost for a word. This is formalized by the following proposition.

PROPOSITION 6.10. *If t is a silent transition of a weighted Petri net N satisfying case 3 of Proposition 3.7: $|{}^\bullet t| = 1$ and $({}^\bullet t)^\bullet = \{t\}$, if moreover $M^0(p) = 0$, and $\forall t' \in {}^\bullet p, \lambda(t') \neq g$, $c(t') > 0$ and $t'^\bullet = \{p\}$ for p the only place in ${}^\bullet t$, then $T(t) = {}^\bullet p$ is such that $|o|_{T(t)} = |o|_{\{t\}}$ for any execution $o = \arg\min_{\lambda(o')=w} c(o')$ with w a word of N finishing by g.*

PROOF. First remark that t can only be fired if some $t' \in {}^\bullet p$ has been fired before (because $M^0(p) = 0$ and $p \in {}^\bullet t$), and for each firing of such a t', t can be fired at most once (because we consider safe nets and so p cannot contain more than one token). So $|o|_{{}^\bullet p} \leq |o|_{\{t\}}$ for any execution $o = \arg\min_{\lambda(o')=w} c(o')$ with w a word of N. Then remark that in any occurrence sequence o achieving the cost of a minimal cost plan (the last transition of such sequence is labelled by g), any occurrence of a $t' \in {}^\bullet p$ is followed by an occurrence of t and there is no occurrence of any $t'' \in {}^\bullet p$ between the occurrences of t' and t. Indeed if t' and t'' are fired without firing t in between, then p would contain two tokens at some time (which is not possible as we consider safe Petri nets only). And if there is some occurrence of t' without occurrence of t afterwards, then the sequence o' obtained by removing the said occurrence of t' is an occurrence sequence (because as $t'^\bullet = \{p\}$ and ${}^\bullet t = \{p\}$ firing t' only enables t) such that $c(o') < c(o)$ (because $c(t') > 0$) and $\lambda(o')$ is a plan (because $\lambda(t') \neq g$), so $\lambda(o)$ is not a minimal cost plan, which is in contradiction with the definition of o. Hence, $|o|_{{}^\bullet p} \geq |o|_{\{t\}}$ for any execution $o = \arg\min_{\lambda(o')=w} c(o')$ with w a word of N ending by g. □

The fact that we require $c(t') > 0$ in the above definition is not a strong limitation as all actions of a planning problem will generally have a positive cost. However it is always possible to avoid this limitation by preferring the shortest among two plans having the same cost.

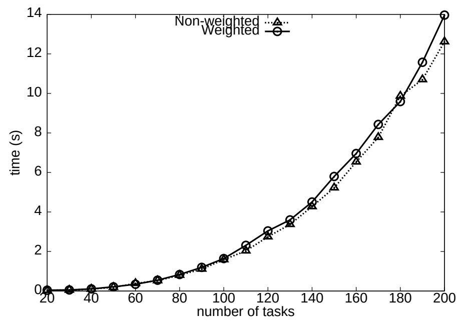
In summary, for each case where a transition can be removed while preserving safeness, one is able to give a sufficient condition for preserving also the costs of words that matter in the resolution of an optimal planning problem, i.e. those finishing by the special goal label g. In two cases however these conditions are more restrictive than in the non-weighted case, when transition contraction has to preserve language and safeness only. The next section of this paper proposes an experimental evaluation of at which extent these new conditions are more restrictive.

7. EXPERIMENTAL EVALUATION IN PRESENCE OF COSTS

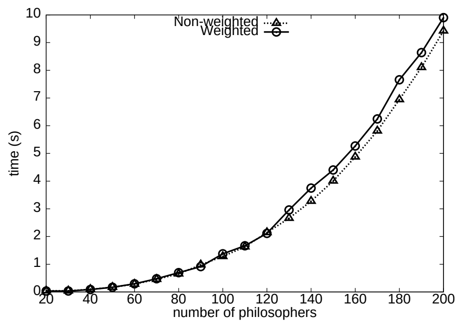
In order to evaluate the impact of the new restrictions (Propositions 6.9 and 6.10) on the efficiency of the silent transitions removal, we ran the message passing algorithm on the same benchmarks as previously (Milner's cyclic scheduler, divide and conquer computation, dining philosophers, dining philosophers with dictionary, mutual exclusion protocol on a ring) with and without weights on transitions. Notice that, in practice, the actual weight of a transition (as soon as it is not 0) has no impact on the contraction, so we do not really add weights to transitions in these examples, but we consider that each transition has a non-zero weight (this corresponds to a "worst" case where no transition can be contracted as if it was without weight).

As before, we ran all our experiments on the same computer (Intel Core i5 processor, 8GB of memory). We first compared the runtimes of the weighted and the non-weighted cases. The results of this comparison are presented in Figures 16a, 16b, 16c, 16d, and 16e.

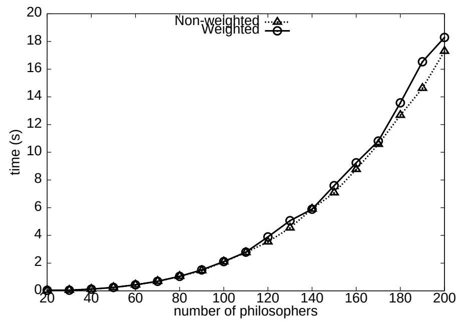
We also compared the numbers of silent transitions remaining in the whole system with and without weights. It turned out that in all the considered benchmark families



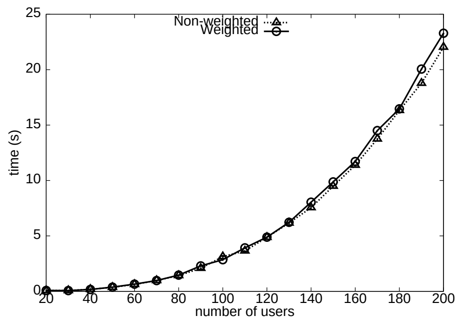
(a) Divide and conquer



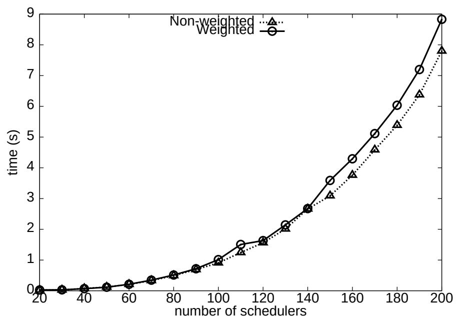
(b) Dining philosophers



(c) Dining philosophers with dictionary



(d) Token ring mutex



(e) Milner's cyclic scheduler

Fig. 16. Comparison of runtimes in the weighted and the non-weighted cases.

the ratio between these numbers was the same irrespective of the size of the instance: 1.0 for the philosophers (with and without dictionary) and for Milner's scheduler, 1.11 for the mutual exclusion protocol, and 1.12 for the divide and conquer computation.

From our experiments it appears that, at least on the particular benchmarks considered, it is reasonable to treat the weighted case by doing contraction according to the conditions given in Propositions 6.8, 6.9, and 6.10. The small runtime difference between the weighted and the non-weighted case appearing even in the cases where the same number of silent transitions can be contracted (dining philosophers with and without dictionary and Milner's cyclic scheduler) can be explained by the additional computational cost of testing the supplementary conditions in presence of weights.

8. CONCLUSION

This paper described an extension of a distributed planning approach proposed in [Fabre et al. 2010], where the local plans of each component are represented as Petri nets rather than automata. This technical extension has three main advantages. First, Petri nets can represent and exploit the internal concurrency of each component. This frequently appears in practice, as factored planning must operate on interaction graphs that are trees, and getting back to this situation is naturally performed by grouping components (through a product operation), which creates internal concurrency. Secondly, the size reduction operations for Petri nets (transition contraction, removal of redundant places and transitions) are local operations: they do not modify the full net but only parts of it. By contrast, the size reduction operation for automata (minimisation) is expensive (while necessary, as only removing silent transitions does not reduce the size of the automata enough and make messages grow too quickly [Jezequel 2012]). As the performance of the proposed factored planning algorithm heavily depends on the ability to master the size of the objects that are handled, Petri nets allow us to deal with much larger components. Thirdly, the product of Petri nets is also less expensive than the product of automata. This again contributes to keeping the complexity of factored planning under control, and to reaching larger components.

This new approach was compared to its preceding version, based on automata computations (which has been previously successfully compared to a version of A* in [Fabre et al. 2010]), on five benchmarks from [Corbett 1996]. For two of these benchmarks, the automata approach is the best. Still, the Petri nets approach scales up decently on these problems, allowing one to address large instances. On the remaining three benchmarks, the automata approach could hardly deal with small instances, while the Petri nets version scaled up very well. We believe that these experimental results show the practical interest of our new approach to factored planning.

We examined how far these ideas could be pushed to perform cost-optimal factored planning. Surprisingly, the extension is rather natural: the central operation of size reduction for Petri nets can be adapted to weighted Petri nets, with almost no increase in theoretical complexity. By contrast, when working with automata, the extension was much more demanding: the minimisation of weighted automata is not even always possible.

Finally, we implemented and evaluated the interest of cost-optimal factored planning based on weighted Petri nets, with a specific focus on the contraction of silent weighted transitions. On the same benchmarks as before, this has shown that the conditions for contraction are not too restrictive and that a size reduction of weighted Petri nets as effective as for non-weighted Petri nets is possible.

Acknowledgments. We would like to thank the anonymous reviewers for their helpful comments. This research was supported by the EPSRC grant EP/K001698/1 (UNCOVER).

REFERENCES

Eyal Amir and Barbara Engelhardt. 2003. Factored Planning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*. 929–935.

Charles André. 1982. Structural Transformations Giving B-Equivalent PT-nets. In *Proceedings of the 3rd European Workshop on Applications and Theory of Petri Nets*. 14–28.

Avrim Blum and Merrick Furst. 1995. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence* 90, 1-2 (1995), 281–300.

Hans Bodlaender. 1993. A Linear Time Algorithm for Finding Tree-Decompositions of Small Treewidth. In *Proceedings of the 25th annual ACM Symposium on Theory of Computing*. 226–234.

Blai Bonet, Patrik Haslum, Sarah Hickmott, and Sylvie Thiébaux. 2008. Directed Unfolding of Petri Nets. *Transactions on Petri Nets and other Models of Concurrency* 1, 1 (2008), 172–198.

Ronen Brafman and Carmel Domshlak. 2008. From One to Many: Planning for Loosely Coupled Multi-Agent Systems. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling*. 28–35.

Tam-Anh Chu. 1987. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis. Massachusetts Institute of Technology.

James C. Corbett. 1996. Evaluating Deadlock Detection Methods for Concurrent Software. *IEEE Transactions on Software Engineering* 22 (1996), 161–180.

Javier Esparza and Keijo Heljanko. 2008. *Unfoldings – A Partial-Order Approach to Model Checking*. Springer.

Javier Esparza, Stefan Romer, and Walter Vogler. 1996. An Improvement of McMillan's Unfolding Algorithm. *Formal Methods in System Design* 20, 3 (1996), 285–310.

Eric Fabre. 2007a. *Bayesian Networks of Dynamic Systems*. Habilitation à Diriger des Recherches. Université de Rennes1.

Eric Fabre. 2007b. Trellis Processes: A Compact Representation for Runs of Concurent Systems. *Discrete Event Dynamic Systems* 17, 3 (2007), 267–306.

Eric Fabre and Loïg Jezequel. 2009. Distributed Optimal Planning: an Approach by Weighted Automata Calculus. In *Proceedings of the 48th IEEE Conference on Decision and Control*. 211–216.

Eric Fabre, Loïg Jezequel, Patrik Haslum, and Sylvie Thiébaux. 2010. Cost-Optimal Factored Planning: Promises and Pitfalls. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling*. 65–72.

Peter Hart, Nils Nilsson, and Bertram Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

Sarah Hickmott, Jussi Rintanen, Sylvie Thiébaux, and Lang White. 2007. Planning via Petri Net Unfolding. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*. 1904–1911.

Loïg Jezequel. 2012. *Distributed Cost-Optimal Planning*. Ph.D. Dissertation. ENS Cachan.

Loïg Jezequel, Eric Fabre, and Victor Khomenko. 2013. Factored Planning: From Automata to Petri Nets. In *Proceedings of the 13th International Conference on Application of Concurrency to System Design*. 137–146.

Victor Khomenko, Alex Kondratyev, Maciej Koutny, and Walter Vogler. 2006. Merged Processes: A New Condensed Representation of Petri net Behaviour. *Acta Informatica* 43, 5 (2006), 307–330.

Victor Khomenko, Mark Schaefer, and Walter Vogler. 2008. Output-Determinacy and Asynchronous Circuit Synthesis. *Fundamenta Informaticae* 88, 4 (2008), 541–579.

Victor Khomenko, Mark Schaefer, Walter Vogler, and Ralf Wollowski. 2009. STG Decomposition Strategies in Combination with Unfolding. *Acta Informatica* 46, 6 (2009), 433–474.

Walter Vogler and Ben Kangsah. 2007. Improved Decomposition of Signal Transition Graphs. *Fundamenta Informaticae* 78, 1 (2007), 161–197.

Received October 2013; revised March 2014; accepted September 2014