Statecharting Petri nets

Statecharting Petri Nets

Rik Eshuis*

Abstract

A polynomial algorithm that translates a Petri net into an equivalent (bisimilar) statechart is presented. The translation preserves the structure of the Petri net. Key property of the algorithm is that it is structural and does not use any Petri net analysis technique. The algorithm is formally proven correct. Though not every statechart equivalent to a net is constructible by the algorithm, the set of constructible statecharts is fairly large.

1 Introduction

Petri nets and statecharts are two popular visual formalisms for modelling systems that exhibit concurrency. Petri nets were introduced in 1962 by C.A. Petri [24]. Since the 1980's Petri nets found their way in practical applications like manufacturing, workflow modelling and performance analysis [28]. Statecharts were introduced in 1987 by D. Harel [9], for use in the structured analysis method STATEMATE [15]. Soon after their appearance, statecharts were adopted in several object-oriented methods, including OMT [29], ROOM [32], and their successor Unified Modeling Language [34, 35]. Thus, both Petri nets and statecharts have found widespread use in both academia and industry.

Both formalisms are supported by various tools, like CPN tools [26] and GreatSPN [2] for Petri nets, and Statemate [12, 15] and Stateflow [20] for statecharts. UML [35] tools are a special case, since they support not only statecharts but also activity diagrams, a Petri-net like notation. Tools supporting Petri nets are strongly focused on analysis of both functional and stochastic properties, while tools supporting statecharts are usually more focused on the software design process, for example offering the facility to generate code from a statechart.

Given this wide tool support, it is interesting to have well-defined translations between Petri nets and statecharts. Such translations could facilitate the exchange of models in different notations between tools, thus allowing designers to use the best of both worlds. Ideally, such translations are correct and structure preserving. A translation is correct if the original and translated model are equivalent. Correctness guarantees that operations done with a tool on a particular model are meaningful, even if the original model was expressed in another formalism. Preservation of structure ensures that the syntax of the original and translated model are alike, making it easier for designers to understand the translated model.

*Address: Eindhoven University of Technology, Department of Technology Management, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, h.eshuis@tm.tue.nl.

concept	Petri net	Statechart
state	place	node (state)
transition	transition	hyperedge (full compound transition)
active state	place containing a token	active node (active state)
global state	marking	configuration

Table 1: Relating Petri net and statechart terminology

The goal of this paper is to define a translation from Petri nets to statecharts, and vice versa, that is both correct and structure preserving. Before giving any details on these two translations, it is useful to discuss the similarities and differences between statecharts and Petri nets. Both formalisms are generalisations of finite state machines. Finite state machines model sequential processes as states connected by transitions. Mathematically, finite state machines are represented as directed graphs, consisting of nodes (representing states) and directed edges (representing transitions). The global state of such a machine at a certain point in time is always one single state, i.e., only one node of the graph is active at a time.

Both Petri nets and statecharts generalise finite state machines by using transitions that can enter and leave multiple states. Sources and targets of such transitions are concurrent: a transition can only be taken if all its sources are active in parallel, and if a transition is taken, all its targets become active in parallel. Mathematically, a transition is represented as a hyperedge. Consequently, the underlying mathematical model is a hypergraph rather than a graph[1]. The global state of a Petri net or statechart is distributed and typically consists of multiple states, i.e. multiple nodes of the hypergraph can be simultaneously active. Even though Petri nets and statecharts share these concepts, the terminology is somewhat different (see Table 1; for definitions see Section 3). To clearly distinguish between the two formalisms, for statecharts we use the terms 'node' and 'hyperedge' rather than the more common 'state' and 'transition'.

Though both Petri nets and statecharts are based on hypergraphs, statecharts have as additional feature an AND/OR hierarchy on nodes to explicitly model concurrency [2]. This hierarchy is visualised by node containment, where nodes that contain other nodes are called composite. There are two kinds of composite nodes: AND and OR, each imposing its own constraint on the global state of the statechart. An AND node denotes concurrency: If an AND node is active, all its immediate subnodes must be active as well. An OR node denotes exclusiveness: If an OR node is active, one of its immediate subnodes must be active as well. (So OR is actually XOR.)

The constraints imposed by the statechart hierarchy imply that leaf nodes of the hierarchy, BASIC nodes, are not active more than once in the same global state. This seems to suggest that statecharts correspond to safe Petri

[1] Usually Petri nets are represented as bipartite graphs. However, equivalently they can be represented as hypergraphs [23].

[2] Even though there do exist extensions of Petri nets with hierarchy, the execution semantics of such hierarchical Petri nets does not depend on the hierarchy. That is, a hierarchical Petri net can always be transformed into an equivalent non-hierarchical Petri net by substituting for each parent node its decomposition. Such a syntactic elimination of hierarchy is not possible in statecharts, essentially because in statecharts concurrency can only be expressed in a hierarchical way using AND nodes.

nets (nets in which in each marking every place has at most one token; see Section 3). However, there is no known structure-preserving translation from safe Petri nets to statecharts. The difficulty in defining such a translation lies in constructing an appropriate AND/OR hierarchy. On the other hand, to translate a statechart into a Petri net, simply dropping composite nodes from the statechart seems to suffice.

In this paper, we present a structure-preserving algorithm that translates a Petri net into an equivalent (bisimilar) statechart. The algorithm is structure-preserving in the sense that it maps each place to a BASIC node and each transition to a hyperedge. Thus, loosely speaking, the algorithm imposes an AND/OR hierarchy of nodes on the Petri net structure. If the algorithm cannot find an equivalent statechart, it fails. The algorithm has been implemented in a software tool, which is available for public download at
`http://is.tm.tue.nl/staff/heshuis/pn2sc`.

Key property of the algorithm is that it is structural and does not use any Petri net analysis techniques, like place invariants or reachability graphs. This leads to the surprising result that the algorithm is polynomial in the size of the net, even though the algorithm involves the construction of an additional structure on top of a Petri net.

The correctness of the algorithm is formally proven. However, it is not complete: it fails for some nets that do have a statechart equivalent. But statecharts which it fails to construct are not likely to be drawn by statechart designers, as we argue in Section 4.2 (p. 31), so this non-completeness does not appear to be a severe limitation in practice.

For the reverse direction, we give a characterisation of the statecharts that can be mapped by a structure-preserving translation into an equivalent Petri net. This shows that not every statechart can be translated into an equivalent Petri net, so omitting composite nodes from a statechart does not always yield an equivalent net.

We focus on unweighted, low level Petri nets and statecharts without any transition labels. For statecharts, weighted transitions are not very useful, as each node is active at most once. Defining translations that take into account data, by considering for example coloured Petri nets [17] and statecharts with local variables and guard and action labels, we defer to future work. Moreover, we do not consider statecharts with event labels. In previous work, we already studied to what respect event triggering can be supported in Petri nets [7, 6, 4].

Since we consider structure-preserving translations, we do not use the statechart in-predicate as guard condition on hyperedges. Given a node x, predicate $in(x)$ is true if and only if the system is currently in x. With the in-predicate as guard condition, a hyperedge can test a node for (in)activeness without leaving it. In Petri nets, this can be modelled using read and inhibitor arcs.

Related work. There is but little work on the topic of translating Petri nets to statecharts. In fact, the only published work with considerable amount of details appears to be Schnabel et al. [31] (in German). They outline an interactive method to translate a safe Petri net into a statechart. The method uses place invariants. Roughly speaking, each invariant maps to a parallel node of a statechart. Like this work, their aim is to translate every place into one BASIC node. However, since a place can occur in several place invariants, in most cases

a place translates into several BASIC nodes. Schnabel et al. outline some ways to prevent such duplications, but sometimes duplications cannot be avoided. In contrast, our algorithm is not interactive and always translates a place into a single BASIC node.

However, since the translation of Schabel et al. is not structure preserving, it sometimes succeeds where our translation fails. Their translation uses guard conditions, in particular the in-predicate, and local variables in combination with action expressions, to simulate cross-synchronisations between parallel statechart nodes. For example, their method could translate the net in Figure 6, for which no structure-preserving equivalent statechart exists, into the statechart shown in Figure 8 (Section 2 explains these examples in detail). As stated above, in this paper we are interested in structure-preserving translations only, hence we do not consider guard conditions and action expression on statechart hyperedges.

Considering the reverse direction, translations from statecharts to Petri nets appear quite frequently in the literature (e.g. [16, 21, 31, 30, 33]). In these approaches, statecharts are considered as having an informal semantics that is formalised by the mapping to Petri nets. However, this ignores that statecharts have their own semantics. Thus, in these approaches no formal proof is given that the translation is actually correct. In Section 3 we define a translation that is proven correct if the statechart satisfies certain conditions. Thus, not every statechart can be mapped in to an equivalent Petri net.

Kishinevsky et al. [19] define a Petri net variant that incorporates some statechart features. The variant, called place chart net, uses hierarchy on places and preemptive transitions: a transition does not only empty its input places put also all descendant places of the input places. However, the authors do not relate place chart nets to standard Petri nets.

Finally, Drusinsky and Harel [3] show that a class of concurrency models that includes both statecharts and Petri nets is more succinct than finite state machines. However, they do not explicitly make a distinction between statecharts and Petri nets, i.e., they fall in the same class.

Structure of this paper. Section 2 introduces some motivating examples of Petri nets and statecharts that do or do not have equivalent behaviour. This sets the stage for the translation in Section 4. Section 3 recalls definitions of Petri nets and statecharts. We formally prove when a statechart has an equivalent Petri net. The definition of statecharts is nonstandard, since we do not consider default nodes. Section 4 presents the transformation algorithm from Petri nets into statecharts. Section 5 winds up with conclusions and further work.

2 Motivating the translation

Some motivating examples of Petri nets that can or cannot be translated into equivalent (bisimilar) statecharts are presented. The translations impose AND/OR hierarchies on the net structures. (Formal definitions can be found in the next section.) In Section 4 these examples are used to motivate the algorithm. The equivalent statecharts shown here are also returned by the algorithm in Section 4, but the inequivalent ones are not, since the algorithm fails if it cannot find a structure-preserving, equivalent statechart for the input Petri net. In

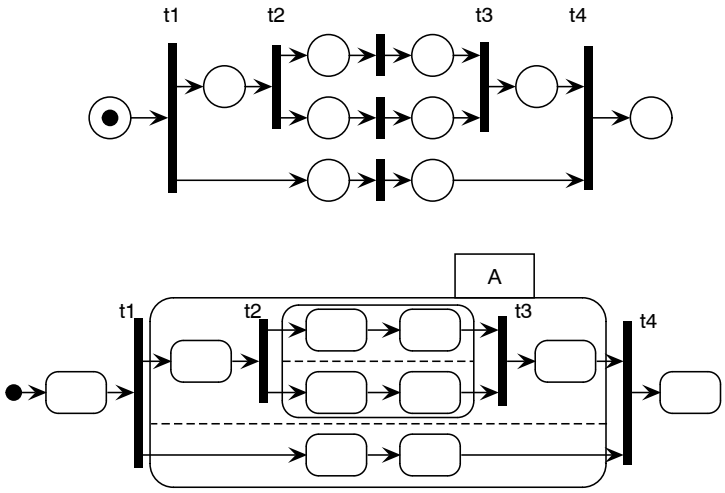

Figure 1: Petri net with balanced forks and joins and equivalent statechart

the examples, statechart nodes and edges are labelled with the names of the corresponding Petri net constructs.

The intuitive relation between a Petri net and a statechart is that every Petri net with balanced forks and joins corresponds to a statechart. A *fork* is a transition with more than one output place and a *join* is a transition with more than one input place. A net has balanced forks and joins if and only if the paths leaving a fork are eventually merged by entering a subsequent join. Every fork-join pair directly corresponds to an AND node. For example, the balanced net in Figure 1 has two fork-join pairs: {t1, t4} and {t2,t3}. Consequently, the equivalent statechart in Figure 1 has two AND nodes. This balancing constraint has been adopted in UML 1.x activity diagrams [34] (a notation with a Petri net-like syntax whose semantics is defined in terms of UML statecharts) in order to ensure that each activity diagram can mapped into a statechart.

However, even Petri nets with unbalanced forks and joins may have structure-preserving, equivalent statecharts. For example, the join in the Petri net in Figure 2 corresponds to two forks, rather than one. Thus, the net is not balanced. Still, an equivalent statechart exists, also shown in Figure 2. As another example, in the net in Figure 3 the join t2 only partially matches the fork t1. But also for this net, an equivalent statechart exists, as shown in Figure 3.

Nevertheless, there are also unbalanced Petri nets that have no structure-preserving equivalent statechart. For example, the net shown in Figure 4 is unbounded; places p2 and p4 can contain an unbounded number of tokens. An equivalent statechart does not exist, because of transition t2. A somewhat similar statechart is shown in the same figure. However, in the statechart, if t2 is taken, node A and all its descendants are left, including nodes p2 and p4. Thus, the statechart is not unbounded.

Unsafe nets also have no structure-preserving statechart equivalent. For example, the net in Figure 5 is unsafe. Place p10 is unsafe, because t3 and t4 can both be taken. Although the statechart in Figure 5 appears to be equivalent, it does not have the same behaviour as the net. In the statechart, if t3 fires,

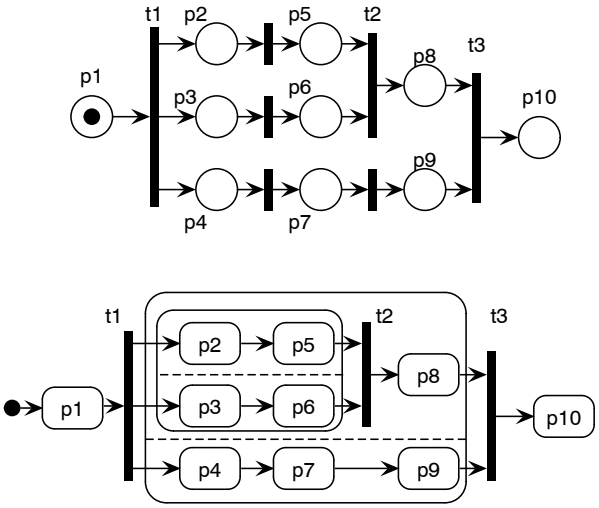

Figure 2: Petri net with unbalanced forks and joins and equivalent statechart

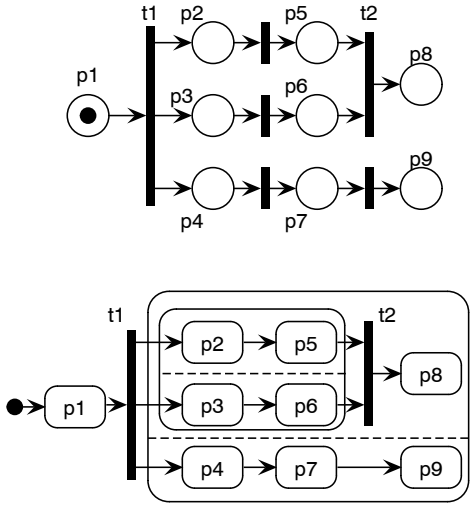

Figure 3: Another Petri net with unbalanced forks and joins and equivalent statechart

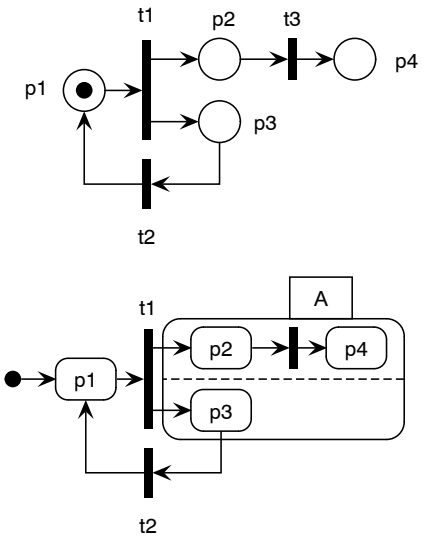

Figure 4: Unbounded Petri net and non-equivalent statechart

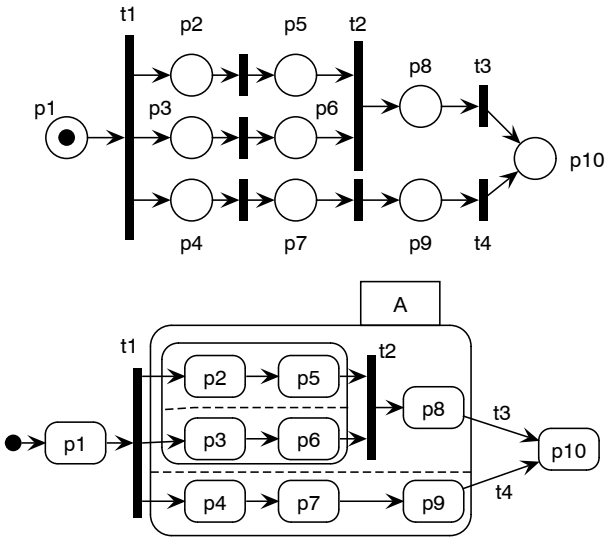

Figure 5: Unsafe Petri net and inequivalent statechart

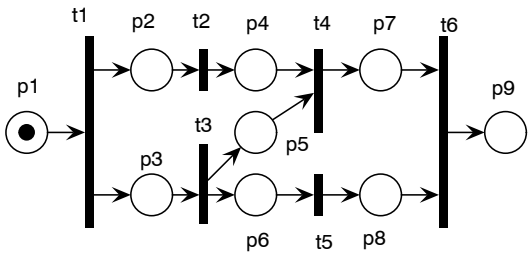

Figure 6: Safe Petri net with cross-synchronisation for which no structure-preserving equivalent statechart exists

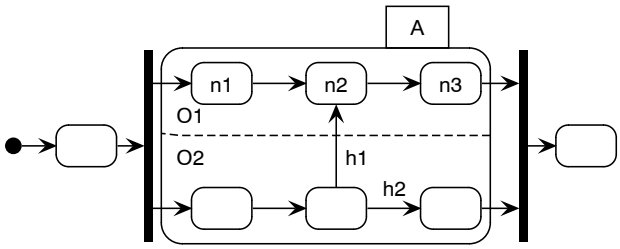

Figure 7: Statechart with a hyperedge connecting parallel nodes

the source of t3, node p8, as well as AND node A is left. According to the statechart semantics, therefore all descendants of A are left as well, including the source of t4, node p9. So in the statechart, taking t3 disables t4, and vice versa. Therefore, the statechart cannot be unsafe.

As we pointed out in the introduction, a Petri net has to be safe to be translatable into a structure-preserving, equivalent statechart, since according the statechart semantics nodes cannot be active more than once at the same time. However, not every safe Petri net can be translated into a statechart. For instance, Figure 6 shows a Petri net in which two parallel branches have a cross-synchronisation between them. The cross-synchronisation prevents that an equivalent statechart exists.

The behaviour of a seemingly similar statechart, shown in Figure 7, is quite complex. Hyperedge h1 connects parallel nodes. If h1 is taken, state A and all its descendants are left, including nodes O1, n1, n2 and n3. (In fact, the statechart is not valid, since it is not specified which node of O2 has to be entered.) Clearly, a statechart with a hyperedge connecting parallel nodes cannot have a Petri net equivalent.

The Petri net in Figure 6 also illustrates the implications of only considering structure-preserving translations. If we allowed a non-structure-preserving translation, there would exist a statechart equivalent, for example the one listed in Figure 8. However, in that statechart hyperedge t4 does not have node p5 as source, but tests whether p5 is active using the in-predicate.

Another example to illustrate these implications is shown in Figure 9. The net on the left does not have a direct statechart equivalent, due to the cross-synchronisation between the two parallel branches. The net on the right does

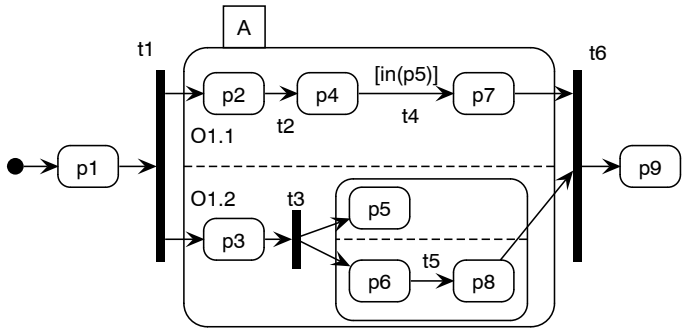

Figure 8: Statechart with in-predicate corresponding to the net in Figure 6

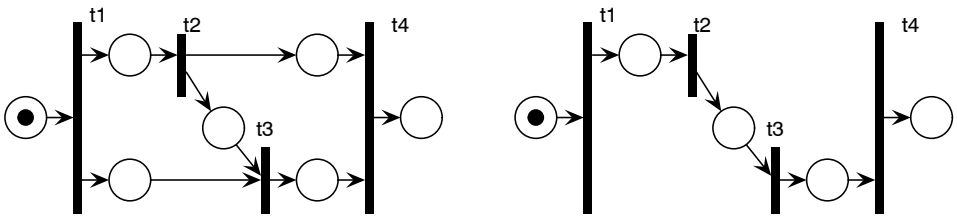

Figure 9: Two equivalent Petri nets. The one on the right has a structure-preserving, equivalent statechart, but the one on the left has not

have a statechart equivalent, namely a simple sequential statechart. However, the net on the left is equivalent to the net on the right (i.e. their transition systems are bisimilar).

3 Preliminaries

We recall some definitions of transition systems, Petri nets and statecharts. Readers familiar with Petri nets and statecharts can skip this section. However, note that the statechart definition differs slightly from the traditional one; details can be found below. Formal definitions of Petri nets can be found in standard text books like Reisig [27] or overview papers like Murata [22]. For statecharts, formal definitions can be found, among others, in papers by Harel et al. [14] and Pnueli and Shalev [25].

3.1 Transition systems

The execution semantics of both Petri nets and statecharts map into transition systems. A transition system is a pair $(S, \longrightarrow, s_i)$ where

- S is a set of states,
- $\longrightarrow \subseteq S \times S$ the transition relation, and
- $s_i \in S$ is the initial state.

We write $x \longrightarrow y$ whenever $(x, y) \in \longrightarrow$.

To compare different transitions systems, we use the notion of a bisimulation [8]. Let $(S_1, \longrightarrow_1, s_{i_1})$ and $(S_2, \longrightarrow_2, s_{i_2})$ be two transition systems. A bisimulation is a relation $R \subseteq S_1 \times S_2$ such that

(i) if $s_1 \ \underline{R} \ s_2$ and $s_1 \longrightarrow_1 s_1'$ then $\exists\, s_2' \in S_2$ such that $s_2 \longrightarrow_2 s_2'$ and $s_1' \ \underline{R} \ s_2'$,

(ii) if $s_1 \ \underline{R} \ s_2$ and $s_2 \longrightarrow_2 s_2'$ then $\exists\, s_1' \in S_1$ such that $s_1 \longrightarrow_1 s_1'$ and $s_1' \ \underline{R} \ s_2'$.

Two transition systems $(S_1, \longrightarrow_1, s_{i_1})$ and $(S_2, \longrightarrow_2, s_{i_2})$ are bisimilar if and only if their initial states s_{i_1} and s_{i_2} are related by a bisimulation relation.

3.2 Petri nets

Syntax. A Petri net (place/transition net) is a tuple (P, T, F, p_i) where

- P is a set of places,
- T is a set of transitions, $P \cap T \neq \varnothing$,
- $F \subseteq (P \times T) \cup (T \times P)$ is a finite set of arcs, the flow relation,
- $p_i \in P$ is the initial place.

We do not consider weights on the arcs, since statecharts do not have such a feature. Normally, Petri nets do not have one specific initial place. However, our definition is not very restrictive: for each Petri net having a set S of initial places, an artificial initial place p_i can be created with an additional transition leaving p_i and filling the original initial places in S.

Given a transition t, $\bullet t = \{\, p \in P \mid (p, t) \in F \,\}$ is the set of input places of t, whereas $t\bullet = \{\, p \in P \mid (t, p) \in F \,\}$ is the set of output places of t. We require that both $\bullet t$ and $t\bullet$ be nonempty. Furthermore, we require the net be *connected*: for any two nodes $n, n' \in P \cup T$, there should be a path $n_0, n_1, .., n_m$, where $n_0 = n$ and $n_m = n'$ such that for every $0 \leq i < m$, either $(n_i, n_{i+1}) \in F$ or $(n_{i+1}, n_i) \in F$.

Petri net are visualised as bipartite graphs, in which circles represent places, bars represent transitions, and arrows represent the flow relation.

Semantics. The global state of a Petri net, called the *marking*, is a function $M : P \longrightarrow \mathbb{N}$ that assigns to each place the number of times it is active. Each single activation of a place is visualised by a black dot in the place, called a *token* in Petri net terminology. In each net shown in this paper, the initial marking $[p_i]$ is depicted, in which initial place p_i contains a single token and every other place has no token.

From a marking M another marking M' can be reached by firing transitions. A transition t can fire in a marking M if and only if M enables t. Marking M enables transition t if and only if all of t's input places are active: for all $p \in \bullet t : M(p) \geq 1$. If t fires in M, marking M' is reached, written $M \xrightarrow{t} M'$, where for every $p \in P$:

$$M'(p) = \begin{cases} M(p) - 1 & \text{, if } p \in \bullet t \setminus t\bullet \\ M(p) + 1 & \text{, if } p \in t\bullet \setminus \bullet t \\ M(p) & \text{, otherwise.} \end{cases}$$

A marking M' is reachable from M if and only if there is a sequence of transitions $t_1, t_2, .., t_n$ such that $M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} M_3 \,..\, M_n \xrightarrow{t_n} M_{n+1}$ where $M_1 = M$ and $M_{n+1} = M'$. A marking M is reachable if and only if it is reachable from $[p_i]$.

This execution semantics maps a Petri net (P, T, F) into a transition system $(S, \longrightarrow, s_i)$ in which the states are markings, the transitions represent firing of some Petri net transitions, and the initial state is the initial marking:

$$\begin{array}{rcl} S & \stackrel{\text{df}}{=} & \{M : P \longrightarrow \mathbb{N}\} \\ \longrightarrow & \stackrel{\text{df}}{=} & \bigcup_{t \in T} \xrightarrow{t} \\ s_i & \stackrel{\text{df}}{=} & [p_i]. \end{array}$$

This transition system is usually called reachability graph in Petri net terminology.

The bound of a place p is the maximum number of tokens assigned to p by a reachable marking. A Petri net is *safe* if and only if every place has bound of 1, i.e., no reachable marking M puts more than one token in some place. A Petri net is *unbounded* if and only if there is a place p that has no bound, i.e., for every bound $k > 0$, there is reachable marking M such that $M(p) > k$.

3.3 Statecharts

The formal definition of the statechart syntax and semantics, presented below, differs slightly from the original definition [14, 25]. In particular, we do not consider default nodes. For each OR node o, the default of o is a child node that is entered if some hyperedge enters o but none of its descendants. Default nodes ensure that taking a hyperedge always results in a valid global next state. We omit them since they do not have any counterpart in Petri nets. Moreover, default nodes can always be eliminated from a statechart by applying some simple preprocessing, as Harel and Naamad indicate [13]. This elimination results in a statechart consisting of hyperedges, called full compound transitions in [13], whose targets are BASIC nodes. Since we omit default nodes, we have to put extra constraints on hyperedges to ensure that when they are taken, a valid global state is reached. These constraints are informally also described in [13].

Syntax. A statechart is a tuple $(N, H, \mathit{source}, \mathit{target}, \mathit{children}, \mathit{type}, n_i)$ where

- N is a set of nodes,
- H is a set of hyperedges, $N \cap H \neq \emptyset$,
- $\mathit{source} : H \longrightarrow N$ is a function defining the set of input nodes for each hyperedge,
- $\mathit{target} : H \longrightarrow N$ is a function defining the set of output nodes for each hyperedge,
- $\mathit{children} : N \longrightarrow \mathbb{P}\, N$ is a function that defines for each node its immediate subnodes. If a node has children, we call it composite.

- *type* : $N \longrightarrow \{\text{BASIC}, \text{AND}, \text{OR}\}$ is the function defining the type for each node. Each noncomposite node is BASIC. Each composite node is either AND or OR. If an AND node is active, all its children are active as well. If an OR node is active, one of its children is active. (So OR is actually a XOR.)
- $n_i \in N$ is the initial node.

Note the similarity with the definition of Petri nets. Nodes resembles places, hyperedges resemble transitions, and functions *source* and *target* resemble the flow relation.

The *children* relation must induce a hierarchy on nodes. To define this formally, we need some auxiliary definitions. Denote by $children^*$ the reflexive-transitive closure of *children*:

$$children^*(n) = \bigcup_{i \geq 0} children_i(n)$$

where

$$\begin{aligned} children^0(n) &= \{n\} \\ children^{i+1}(n) &= \bigcup_{n' \in children(n)} children^i(n) \end{aligned}$$

If $n' \in children^*(n)$, we say that n is *ancestor* of n' and n' is *descendant* of n. Two nodes n, n' are ancestrally related if either n is an ancestor of n' or n' an ancestor of n.

The next three constraints ensure that the *children* relation arranges nodes in a hierarchy [36]:

- There is one single node $root \in N$ that does not have any parents:

  $$\exists_1\, n \in N : n = root \text{ and for all } n' \in N : n \notin children(n').$$

- Node *root* is ancestor of every node in the statechart, including itself:

  $$N = children^*(root).$$

- Every node, except *root*, has one parent:

  $$\text{for all } n \in N, n \neq root \text{ implies } \exists_1\, n' \in N : n \in children(n').$$

We write $parent(n)$ to denote the node n' that has n as child. So $n' = parent(n)$ iff $n \in children(n')$.

For technical reasons, we require that node *root* **has type OR.** Finally, the initial node n_i must be a child of *root*.

Statecharts are visually represented as hierarchical hypergraphs [10]. Figure 10 shows an example statechart, as well as the node hierarchy depicted by the statechart. Rounded rectangles represent nodes. Children of a node are contained in it. The children of an AND node are separated by a dotted line. In the example, AND node `n9` has two OR children: `n7` and `n8`. And OR node `n7` has children `n2` and `n3`. Bars represent hyperedges with multiple sources and/or targets (using a UML-like notation). Sources and targets of such hyperedges are represented by their incoming and outgoing arrows. In the example, the source of `h1` is `n1` and its targets are `n2` and `n4`. The arrow leaving the black dot points at the initial node. The root node is never shown in a statechart diagram.

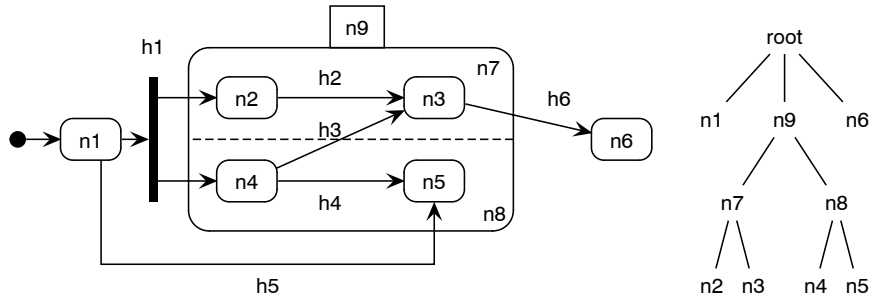

Figure 10: Example statechart and its node hierarchy

Semantics. Not every set of nodes is a valid state of a statechart. Every valid state, called a configuration, must satisfy several constraints, defined below.

First, we introduce some auxiliary definitions. The lowest common ancestor of a set $X \subseteq N$ of nodes, written $lca(X)$, is the most nested node $n \in N$ that is an ancestor of every node in X:

$$X \subseteq children^*(n)$$
$$\text{for all } n' \in N : X \subseteq children^*(n') \Rightarrow n \in children^*(n')$$

For example, in Figure 10, $lca(\{\mathsf{n2}, \mathsf{n3}\})$ is OR node n7, whereas the $lca(\{\mathsf{n4}, \mathsf{n3}\})$ is AND node n9.

Given a set X of nodes, $lca^+(X)$ is the most nested OR node that is ancestor of every node in X. So only if $type(lca(X)) \neq AND$, then $lca^+(X) = lca(X)$. For example, $lca^+(\{\mathsf{n4}, \mathsf{n3}\}) = root$, since *root* is the OR parent of AND node n9.

Two nodes $n, n' \in N$ are orthogonal if and only if they are not ancestrally related and their lca is an AND node. In the example, nodes n2 and n4 are orthogonal, but nodes n2 and n7 are not (since n2 is child of n7) and neither are n2 and n3 (since their lowest common ancestor is an OR node).

A set X of nodes is consistent if and only if for every pair $x, y \in X$, either x and y are ancestrally related or x and y are orthogonal. Thus, set {n2,n7} is consistent.

A *configuration* is a maximal, consistent set of nodes. That is, adding a node to a configuration would make it inconsistent. In the example, set {n1,*root*} is a configuration, as is {n2,n4,n7,n8,n9,*root*}. Configurations are the valid global states of the statechart.

A configuration C satisfies the following properties, for every $x \in C$:

- $type(x) = OR \Rightarrow |children(x) \cap C| = 1$
- $type(x) = AND \Rightarrow children(x) \subset C$
- $x \neq root \Rightarrow parent(x) \in C$.

A hyperedge h is *enabled* in configuration C if all its sources are active: $source(h) \subseteq C$. When a hyperedge is enabled, it can be taken.

To define the effects of taking a hyperedge, we need some additional definitions. The scope of a hyperedge h is the most nested OR node containing the

sources and targets of h:

$$scope(h) \stackrel{\text{df}}{=} lca^{+}(source(h) \cup target(h)).$$

Upon taking h, all descendants of $scope(h)$ will be left. The new configuration entered by taking h consists of $target(h)$, their ancestors, and nodes in the current configuration C that are ancestors of $scope(h)$. More precisely, upon taking h, the configuration C changes into C', written $C \xrightarrow{h} C'$, where

$$C' = dcomp((C \setminus children^{*}(scope(h))) \cup target(h))$$

where given a set S, the default completion $dcomp(S)$ is the smallest set D such that

- $S \subseteq D$
- if $s \in D$ and $s \neq root$ then $parent(s) \in D$.

However, because we do not consider default nodes, a complication that arises is that the new configuration may be incompletely specified. For example, if in Figure 10 the configuration is $\{root, \mathsf{n1}\}$ and $\mathsf{h5}$ is taken, then node $\mathsf{n1}$ is left and node $\mathsf{n5}$ and all its ancestors are entered, so the next configuration would be $\{root, \mathsf{n5}, \mathsf{n8}, \mathsf{n9}\}$. However, that configuration is invalid, since $\mathsf{n9}$ is in the new configuration, but its child $\mathsf{n7}$ is not. As another example, if in the statechart in Figure 7 hyperedge $\mathsf{t1}$ is taken, then $\mathsf{O2}$ is left but not entered, so here also an incomplete configuration results.

To avoid such incomplete configurations, we put two additional constraints on statecharts, not present in the original definition. If statecharts satisfy these two constraints, then after taking an enabled hyperedge with a consistent set of target nodes, a valid next configuration is reached. First, we require that each hyperedge h is basic, i.e. its sources and targets are basic nodes:

$$basic(h) \stackrel{\text{df}}{\Leftrightarrow} (source(h) \cup target(h)) \subseteq BASIC$$

where

$$BASIC \stackrel{\text{df}}{=} \{\, n \in N \mid type(n) = \text{BASIC} \,\}.$$

Otherwise, if a hyperedge would enter an OR node but none of its children, an incomplete configuration would result.

Second, we require that hyperedges are target complete, i.e. they should not enter some AND node only partially. For example, in the statechart in Figure 10, hyperedge $\mathsf{h5}$ only partially enters AND node $\mathsf{n9}$: no descendant of $\mathsf{n7}$ is entered. A hyperedge h is *target complete* if and only if for each AND node a that it enters, i.e. $target(h) \cap children^{*}(a) \neq \emptyset$, it also enters every child of a:

$$\begin{aligned} target\_complete(h) \quad \stackrel{\text{df}}{\Leftrightarrow} \quad & \forall\, n \in children^{*}(scope(h)) : \\ & type(n) = AND \wedge children^{*}(n) \cap target(h) \neq \emptyset \Rightarrow \\ & \quad \forall\, n' \in children(n) : children^{*}(n') \cap target(h) \neq \emptyset. \end{aligned}$$

Note that if we had used default nodes, these two constraints would not have been needed. Then for every hyperedge h that enters an OR node o but

none of its children, in the default completion the default node of o would be included. And for every hyperedge h that partially enters some AND node a, the default nodes of the OR children not entered by h would be added to the partial configuration (cf. [14, 25]).

In addition, we formulate a constraint specifying that a hyperedge h should leave an AND node either completely or not at all. Though the execution semantics of statecharts enforces AND nodes not be left completely (since all nodes below $scope(h)$ are left), we show in the sequel that statecharts satisfying this constraint are bisimilar to Petri nets. A hyperedge $h \in H$ is *source_complete* if and only if for each AND node a that it leaves, it leaves each of a's children as well.

$$\begin{aligned} source\_complete(h) \stackrel{\text{df}}{\Leftrightarrow} \; & \forall\, n \in children^*(scope(h)) : \\ & type(n) = AND \wedge children^*(n) \cap source(h) \neq \emptyset \Rightarrow \\ & \forall\, n' \in children(n) : children^*(n') \cap source(h) \neq \emptyset. \end{aligned}$$

In the statechart of Figure 10, hyperedge h6 is not source complete.

If a hyperedge is both source and target complete, we call it *complete*. If a complete hyperedge is taken in configuration C, the BASIC nodes left in C are the sources of h, and the BASIC nodes entered are the targets of h. Below, we prove this formally (Lemma 3.1 and 3.2). In Figure 10, hyperedges h1, h2, and h4 are complete. Note that h3 is neither source complete nor target complete.

In addition, each hyperedge $h \in H$ should be *consistent*, i.e., it should enter and leave a consistent set of nodes. A hyperedge with an inconsistent source can never become enabled. And taking a hyperedge with an inconsistent target would result in an invalid configuration. All hyperedges of the statechart in Figure 10 are consistent.

$$consistent(h) \stackrel{\text{df}}{\Leftrightarrow} consistent(source(h)) \wedge consistent(target(h)).$$

A statechart SC is *wellformed* iff every hyperedge $h \in H(SC)$ is basic, consistent, and complete. The example statechart in Figure 10 is not wellformed.

$$wellformed(SC) \stackrel{\text{df}}{\Leftrightarrow} \forall\, h \in H : basic(h) \wedge consistent(h) \wedge complete(h).$$

In Section 3.4, we show that wellformed statecharts are equivalent to Petri nets.

The execution semantics maps a statechart into a transition system $(S, \longrightarrow, s_i)$ where

$$\begin{aligned} S &\stackrel{\text{df}}{=} \{C \subseteq N \mid C \text{ is a valid configuration}\} \\ \longrightarrow &\stackrel{\text{df}}{=} \bigcup_{h \in H} \stackrel{h}{\longrightarrow} \\ s_i &\stackrel{\text{df}}{=} \{n_i, root\}. \end{aligned}$$

For wellformed statecharts, the following lemmas can be easily proven.

Lemma 3.1 *Given a basic hyperedge h that is source complete and consistent. If $C \stackrel{h}{\longrightarrow} C'$, then $(C \cap children^*(scope(h))) \cap BASIC = source(h)$.*

Proof. $\subseteq$ direction: Take a node $n \in (C \cap \mathit{children}^*(\mathit{scope}(h))) \cap \mathit{BASIC}$. By definition of C, n is orthogonal to all other BASIC nodes in C. Moreover, n is a descendant of $\mathit{scope}(h)$. By definition of source completeness, then $n \in \mathit{source}(h)$.

$\supseteq$ direction: By definition of $\mathit{scope}(h)$, $x \in \mathit{source}(h)$ implies $x \in \mathit{children}^*(\mathit{scope}(h))$. □

Lemma 3.2 *Given a basic hyperedge h that is target complete and consistent. If $C \xrightarrow{h} C'$, then $\mathit{dcomp}((C \setminus \mathit{children}^*(\mathit{scope}(h))) \cup \mathit{target}(h)) \cap \mathit{BASIC} = \mathit{target}(h) \cup ((C \setminus \mathit{children}^*(\mathit{scope}(h))) \cap \mathit{BASIC})$.*

Proof. By similar reasoning as in the proof of Lemma 3.1. □

Wellformed statecharts have the property that sources and targets of hyperedges specify the next configuration completely: no additional nodes need to be added or removed. This property is default in Petri nets: a change in marking only depends on the transition taken. Thus, wellformed statecharts and Petri net correspond closely to each other. Each wellformed statechart can be translated into an equivalent Petri net by simply removing all composite nodes from it; we prove this in the next subsection. In the next section, we show how some Petri nets can be translated into an equivalent, structure-preserving, wellformed statechart.

3.4 From statecharts to Petri nets

Deriving a Petri net from a statechart structure as specified in the previous paragraph is straightforward: simply drop the composite nodes from the statechart. The function *SCtoPN* defines this formally:

$$\mathit{SCtoPN}((N, H, \mathit{source}, \mathit{target}, \mathit{children}, \mathit{type}, n_i)) = (P, T, F, p_i)$$

where

$$\begin{array}{lcl} P & \stackrel{\text{df}}{=} & \{\, p_n \mid n \in N \wedge \mathit{type}(n) = \text{BASIC} \,\} \\ T & \stackrel{\text{df}}{=} & \{\, t_h \mid h \in H \} \\ F & \stackrel{\text{df}}{=} & \{\, (x_n, y_h) \in P \times T \mid n \in \mathit{source}(h) \,\} \\ & \cup & \{\, (y_h, x_n) \in T \times P \mid n \in \mathit{target}(h) \,\} \\ p_i & \stackrel{\text{df}}{=} & p_{n_i}. \end{array}$$

The correctness of this function is shown by defining a bisimulation on the transition systems induced by the statechart and Petri net. However, the bisimulation only exists if the statechart is wellformed. In general, illformed statecharts like the one in Figure 10 do not have a bisimilar statechart. In exceptional cases, illformed statecharts can also have a bisimilar Petri net (if every hyperedge that is not source or target complete never can become enabled).

Theorem 3.1 (correctness) *Given a wellformed statechart $SC = (N, H, \mathit{source}, \mathit{target}, \mathit{children}, \mathit{type}, n_i)$. The Petri net $\mathit{SCtoPN}(SC)$ bisimulates the original statechart.*

Proof. We define a relation R that relates markings and configurations. Next, we show that R is a bisimulation.

Given a configuration C and a marking M, define R by

$$C \ \underline{R} \ M \stackrel{\text{df}}{\Leftrightarrow} \forall\, n \in \mathit{Nodes} : \mathit{type}(n) = \text{BASIC} \wedge n \in C \ \Leftrightarrow \ M(p_n) = 1$$

Given an enabled hyperedge h in C. Denote by C' the configuration reached by taking h. Denote by M' the marking reached by taking t_h. Then

$$\begin{array}{ll}
& n \in C' \wedge \mathit{type}(n) = \text{BASIC} \\
\Leftrightarrow & \text{[lemma 3.2]} \\
& n \in \mathit{target}(h) \vee ((n \in C \wedge n \notin \mathit{children}^*(\mathit{scope}(h))) \wedge \mathit{type}(n) = \text{BASIC}) \\
\Leftrightarrow & \text{[lemma 3.1, lemma 3.2, def. scope]} \\
& n \in \mathit{target}(h) \vee ((n \in C \wedge n \notin \mathit{source}(h) \cup \mathit{target}(h)) \wedge \mathit{type}(n) = \text{BASIC}) \\
\Leftrightarrow & \text{[logic]} \\
& (n \in \mathit{target}(h) \wedge (n \in C \vee n \notin C)) \\
& \vee \\
& ((n \in C \wedge n \notin \mathit{source}(h) \cup \mathit{target}(h)) \wedge \mathit{type}(n) = \text{BASIC}) \\
\Leftrightarrow & \text{[lemma 3.1, def. } \stackrel{h}{\longrightarrow}\text{]} \\
& (n \in \mathit{target}(h) \wedge ((n \in C \wedge n \in \mathit{source}(h)) \vee (n \notin C \wedge n \notin \mathit{source}(h)))) \\
& \vee \\
& ((n \in C \wedge n \notin \mathit{source}(h) \cup \mathit{target}(h)) \wedge \mathit{type}(n) = \text{BASIC}) \\
\Leftrightarrow & \text{[logic]} \\
& (n \in \mathit{source}(h) \cap \mathit{target}(h) \wedge n \in C) \\
& \vee \\
& (n \in \mathit{target}(h) \setminus \mathit{source}(h) \wedge n \notin C) \\
& \vee \\
& ((n \in C \wedge n \notin \mathit{source}(h) \cup \mathit{target}(h)) \wedge \mathit{type}(n) = \text{BASIC}) \\
\Leftrightarrow & \text{[definition } t_h\text{, } R\text{]} \\
& (p_n \in \bullet t_h \cap t_h\bullet \wedge M(p_n) = 1) \\
& \vee \\
& (p_n \in t_h\bullet \setminus \bullet t_h \wedge M(p_n) = 0) \\
& \vee \\
& (M(p_n) = 1 \wedge p_n \notin \bullet t_h \cup t_h\bullet) \\
\Leftrightarrow & \text{[def. } \stackrel{t_h}{\longrightarrow}\text{, } M \text{ is safe]} \\
& M'(p_n) = 1.
\end{array}$$

□

In the next section, we focus on the reverse direction.

4 Translating Petri nets to statecharts

Motivated by the example translations in Section 2, we present a structural algorithm that translates a Petri net into an equivalent, structure-preserving statechart, by mapping each place to a BASIC node, each transition to a hyperedge and imposing an AND/OR hierarchy of nodes on the BASIC nodes. If the algorithm does not find an equivalent statechart, it fails. The algorithm is formally proven correct, but we show it is not complete. That is, every statechart constructed by the algorithm is a valid statechart whose behaviour is equivalent to the original Petri net, but the algorithm fails on some Petri nets that do have a statechart equivalent. However, as we argue, statecharts not constructible by the algorithm are not likely to be drawn in practice.

4.1 Algorithm

Before the algorithm is presented, some of its features are listed, motivated by the example translations provided in Section 2. Key feature of the algorithm is that composite nodes are modelled as well-founded sets, whose elements are the children of the composite nodes. Atoms of the compound sets are the BASIC nodes, which correspond to Petri net places.

Second, since the algorithm is structure preserving, places map to BASIC nodes and transitions to hyperedges. Sources of a hyperedge h corresponding to transition t are the BASIC nodes corresponding to the places in the preset of t. Similarly, targets from h_t are the BASIC nodes corresponding to places in the postset of t.

- $\surd$ The algorithm initially creates a BASIC node for each place and a hyperedge for each transition.

Third, each AND node in the statechart corresponds to a non-singleton preset or postset of some transition in the Petri net. For example, in Figure 3, AND nodes correspond to sets $\{\mathtt{p2}, \mathtt{p3}, \mathtt{p4}\}$ and $\{\mathtt{p5}, \mathtt{p6}\}$. For ease of reference, in all nets shown in this paper, each transition has either a singleton preset or singleton postset or both (but this is not a restriction the algorithm imposes). For such nets, instead of referring to preset and postsets, we can refer to the corresponding (fork and join) transitions. Thus, for the example net in Figure 3, we can also say that AND nodes correspond to transitions `t1` and `t2`. OR nodes are created for each place and each AND node.

- $\surd$ The algorithm initially creates AND nodes for each non-singleton preset and non-singleton postset of the net. Next, it creates an OR node for each BASIC node and each AND node.

Fourth, we need only consider statecharts in which for each hyperedge h, the least common OR ancestor ls of its sources equals the least common OR ancestor lt of its targets. From the definition of scope, it follows that then this node ls (lt) equals the scope of h. If a statechart does not satisfy this property, either (i) the ls is an ancestor of lt or vice vera, or (ii) both ls and lt are descendants of $scope(h)$. Case (i) occurs when OR nodes are nested in each other, which is not needed for our purposes. Case (ii) occurs if the statechart is not wellformed; see for example Figure 6. However, we are only interested in

wellformed statecharts, since such statecharts are bisimilar to Petri nets having an isomorphic structure, according to Theorem 3.1.

- $\surd$ The algorithm ensures that the set of sources and set of targets of each hyperedge share the same least common OR ancestor.

Fifth, the correspondence between non-singleton sets and AND nodes is not one to one. Sometimes an AND state corresponds to multiple of such presets or postsets. For example, in Figure 1 transitions t1 and t4 share the same AND node, as well as t2 and t3. Thus, some of the initially created AND nodes are superfluous.

Moreover, from the previous item it follows that some of the initially created OR nodes are superfluous too. For example, in the statechart in Figure 2, nodes p2 and p3 share the same OR parent, because of the simple edge connecting them. But initially, two OR nodes are created for these two nodes. Thus one of these OR nodes is superfluous.

- $\surd$ The algorithm ensures that initially created composite nodes that turn out to be superfluous are removed.

Sixth, some of the AND nodes initially created by the algorithm need to be nested inside each other. For example, in both Figure 1 and Figure 3, the AND node created for t1 contains the AND node created for t2.

- $\surd$ The algorithm nests AND nodes inside each other.

Nesting an AND node inside another AND node can make another AND node superfluous. For example, in Figure 2, the AND node a initially created for t1 has three OR children. But the algorithm also creates AND nodes for t2 and t3, and nesting these makes a superfluous.

Seventh, not every Petri net has a structure-preserving statechart equivalent (for example the nets in Figures 5 and 4). So sometimes the translation algorithm cannot give a result.

- $\surd$ The algorithm fails if it cannot find an equivalent, structure-preserving statechart.

Eighth, Theorem 3.1 suggests that a statechart needs to be wellformed in order to be bisimilar to the original Petri net. To see this, suppose the algorithm returns a statechart $SC(PN)$ for a given net PN. Then by removing the statechart hierarchy, a Petri net PN' isomorphic to PN can be obtained. Theorem 3.1 shows that if $SC(PN)$ is wellformed, it is bisimilar to PN'. Otherwise, for some hyperedge h taken in some configuration C, BASIC nodes in C are left that are not sources of h.

- $\surd$ The algorithm returns a wellformed statechart.

The algorithm is defined in Figure 11. It has all these eight features: if it does not fail, it translates a Petri net into a wellformed statechart, with nested AND nodes, that is bisimilar to the original Petri net. The algorithm consists of five phases, listed in Table 2.

We now discuss these phases in more detail, also defining and motivating some auxiliary operations used in the algorithm.

```
1: procedure PetriNetToStatechart((P, T, F, p_i))
2:     BASICNodes := { n_p | p ∈ P }
3:     H := { h_t | t ∈ T }
4:     for h_t ∈ H do
5:         source(h_t) := { n_p | p ∈ •t }
6:         target(h_t) := { n_p | p ∈ t• }
7:     end for
8:     ANDNodes := {set(X) | |X| > 1 ∧ ∃ h ∈ H : source(h) = X ∨ target(h) = X}
9:     ORNodes := { {x} | x ∈ BASICNodes ∪ ANDNodes }
10:    Nodes := BASICNodes ∪ ANDNodes ∪ ORNodes
11:    for h ∈ H do
12:        s_lca+ := pick(find_lca+(source(h)))
13:        t_lca+ := pick(find_lca+(target(h)))
14:        if (contains(s_lca+, t_lca+) ∨ contains(t_lca+, s_lca+)) then
15:            cycle in children relation, so fail
16:        else
17:            replace(Nodes, s_lca+, s_lca+ ∪ t_lca+)
18:            replace(Nodes, t_lca+, s_lca+ ∪ t_lca+)
19:        end if
20:    end for
21:    tovisit := {n ∈ Nodes | isAND(n)}
22:    while tovisit ≠ ∅ do
23:        X := a lower bound of tovisit
24:        AND := AND_cover(X, tovisit)
25:        uncovered := {x ∈ X | ¬ contains(x, AND)}
26:        if (uncovered ≠ ∅ ∨ |AND| > 1) then
27:            OR := { Y ∈ Nodes | ∃ A ∈ AND : A ∈ Y }
28:            if contains(X, OR) then
29:                cycle in children relation, so fail
30:            end if
31:            X' := OR ∪ uncovered
32:        else
33:            X' := pick(AND)
34:        end if
35:        if X' ∈ Nodes then
36:            replace(Nodes, parent(X), parent(X) ∪ parent(X'))
37:            replace(Nodes, parent(X'), parent(X) ∪ parent(X'))
38:        end if
39:        replace(Nodes, X, X')
40:        tovisit := tovisit \ { X }
41:    end while
42:    if some OR node has two parents, fail
43:    if some hyperedge has inconsistent sources or targets, fail
44:    create children and type relation
45:    return (Nodes, H, source, target, children, type, n_{p_i})
46: end procedure
```

Figure 11: Algorithm that translates a Petri net into a statechart

Phase	Lines
initialisation of nodes and hyperedges	2-10
merging of OR nodes	11-20
nesting AND nodes	21-41
checking the statechart	42-43
creating the children and type relations	44

Table 2: High-level structure of the algorithm in Figure 11

Initialisation. The BASIC nodes and hyperedges of the statechart are initialised by mapping each place into a BASIC node and each transition into a hyperedge (lines 2 and 3). The sources (targets) of each hyperedge are the basic nodes corresponding to the places in the preset (postset) of the corresponding transitions (for loop starting at line 4).

Composite nodes are modelled as compound sets. A compound set is a set of which the elements are either other sets, i.e., other composite nodes, or atoms, i.e., basic nodes. The membership relation models the children relation, that is, if $x \in y$ then node x is a child of node y. We require that all sets are well-founded, i.e., we do not allow a hyperset [1] which contains itself.

Next, the composite nodes are initialised. For each non-singleton set X that is the source or target set of some hyperedge h, an AND node a is created, whose children are the OR parents of the BASIC nodes in X, that is, if $x \in X$, then $\{x\}$ is child of A. (l. 8). To define a, we use auxiliary function *set*, which maps a set X into a set of singletons, one for each element $x \in X$:

$$set(X) \stackrel{\text{df}}{=} \{\, \{x\} \mid x \in X \,\}.$$

Finally, for each node n that is BASIC or AND, an OR node $\{n\}$ is created (l. 9).

To illustrate these lines, Figure 12 shows the structure of set *Nodes* for the example net in Figure 2 after the initalisation phase has finished (immediately before line 11). The structure is visualised as a graph [1]: nodes represent compound sets or atoms, and a directed edge from m to n means that n is member of m.

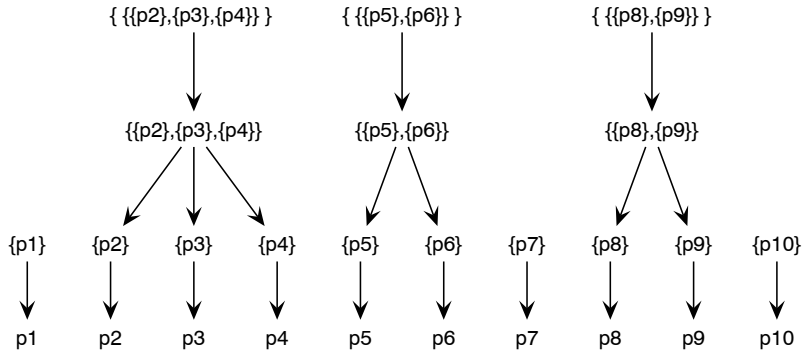

Figure 12: Structure of set *Nodes* for the net in Figure 2 immediately after the initialisation phase has finished

Merging OR nodes. In the previous phase, for each hyperedge h having a set S of sources and a set T of targets two unique OR nodes x_S and x_T have been created (l. 9), whose descendants comprise S and T respectively. In this phase, these two OR nodes x_S and x_T are merged, that is, both x_S and x_T are replaced by a single OR node x_h whose children are the children of both x_S and x_T.

We discuss this part line by line. Given a hyperedge h, line 12 retrieves the unique most nested OR node containing all sources of h, i.e., the least common OR ancestor of set $source(h)$. Similarly, line 13 retrieves the unique most nested OR node containing all targets of h, i.e., the lca$^+$ of set $target(h)$.

In these lines, two auxiliary functions are used, *pick* and *find_lca*$^+$. Function *pick* expects a singleton and returns the element contained in the singleton: $pick(\{x\}) \stackrel{\text{df}}{=} x$. The function is not defined for non-singletons. We will show below in Lemma 4.2 *pick* only gets a singleton argument, i.e., that *find_lca*$^+$ returns singletons.

Given a set X of BASIC nodes (source set or target set of some hyperedge), function *find_lca*$^+(X)$ gives the set of OR parents for those nodes that are the least common ancestor (lca) of X:

$$\mathit{find\_lca}^+(X) \stackrel{\text{df}}{=} \{\, n \in \mathit{Nodes} \mid \exists\, n' \in \mathit{Nodes} : \mathit{is\_lca}(n', X) \wedge n' \in n \,\}.$$

Before we define $is\_lca(n, X)$, observe that initially for X a unique AND or BASIC node n has been created that is the lca of X. If set X is a singleton, n is BASIC: $n = pick(X)$; otherwise, n is AND: $n = set(X)$. However, the definition $is\_lca(n, X)$ needs to take into account that if n is an AND node, the OR children of n might have gained additional members. For example, for the net in Figure 2, the initally created AND node {{p5},{p6}} might have become {{p5},{p3,p6}}, namely if the transition connecting p3 and p6 has been processed.

Thus, *is_lca* is formally defined as:

$$\mathit{is\_lca}(n, X) \stackrel{\text{df}}{\Leftrightarrow} \begin{cases} X \subseteq \bigcup_{n' \in n} n' & , \text{ if } |X| > 1 \\ pick(X) = n & , \text{ otherwise} \end{cases}$$

After the OR nodes *s_lca*$^+$ and *t_lca*$^+$ have been retrieved, they are merged. For merging, we make use of the fact that a composite node n is a set, the elements of which are children of n. So the new OR node can be created by simply taking the union of *s_lca*$^+$ and *t_lca*$^+$. Next, each occurrence of *s_lca*$^+$ and *t_lca*$^+$ has to be replaced by the new OR node (l. 17-18).

However, merging is not allowed if *s_lca*$^+$ is a descendant of *t_lca*$^+$ or vice versa (l. 14). Then merging would cause a cycle in the children relation; and the merged set x_h would not be wellfounded. The procedure therefore fails in that case (l. 15). For example, suppose the algorithm is applied to the net in Figure 4, processing first t3, then t1, and finally t2. The node structure immediately before t2 is processed is shown in Figure 13, while the structure that would result if the *s_lca*$^+$ and *t_lca*$^+$ of t2 would be merged is shown in Figure 14.

To define the merging procedure, we use two auxiliary operations on wellfounded sets, *contains* and *replace*. Given a wellfounded set S, function $contains(x, S)$ is true if element x appears in S. If some element $s \in S$ is a set, *contains* is recursively applied to every element in s. For example, $contains(\{1\}, \{1, \{1, 2\}\})$

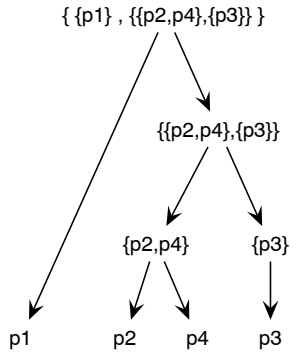

Figure 13: Structure of *Nodes* for the net in Figure 4 after the for loop started at line 11 has processed t3 and t1, but not yet t2

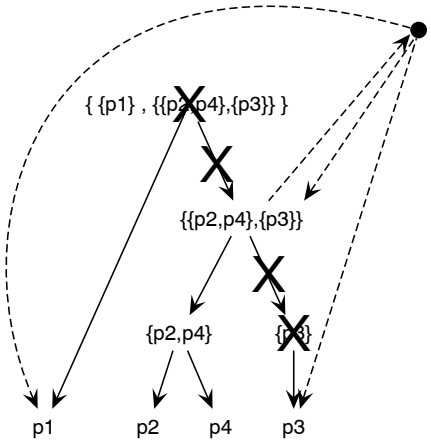

Figure 14: Structure of *Nodes* for the net in Figure 4 while processing t2 as h in the for loop started at line 11. Nodes s_lca^+ and t_lca^+ and their incoming arrows are crossed. The black dot denotes the new, merged OR node. Dotted lines indicate the membership relation if the merged OR node would replace the two existing s_lca^+ and t_lca^+

is false, but $contains(\{1\}, \{\{\{1\}\}\})$ is true.

$$
\begin{array}{rcl}
contains(x, \emptyset) & = & false \\
contains(x, y) & = & false \quad \text{if } y \text{ is not a set} \\
contains(x, \{\, y \,\} \cup S) & = & \begin{cases} true, \text{ if } x = y \\ contains(x, y) \vee contains(x, S), \text{ otherwise} \end{cases}
\end{array}
$$

Given a wellfounded set S and two elements x and y, function $replace(x, y, S)$ replaces each appearance of x in S by y. Function *replace* is recursive: if an element $s \in S$ is a set, then *replace* is also applied to every element of s. For exam-

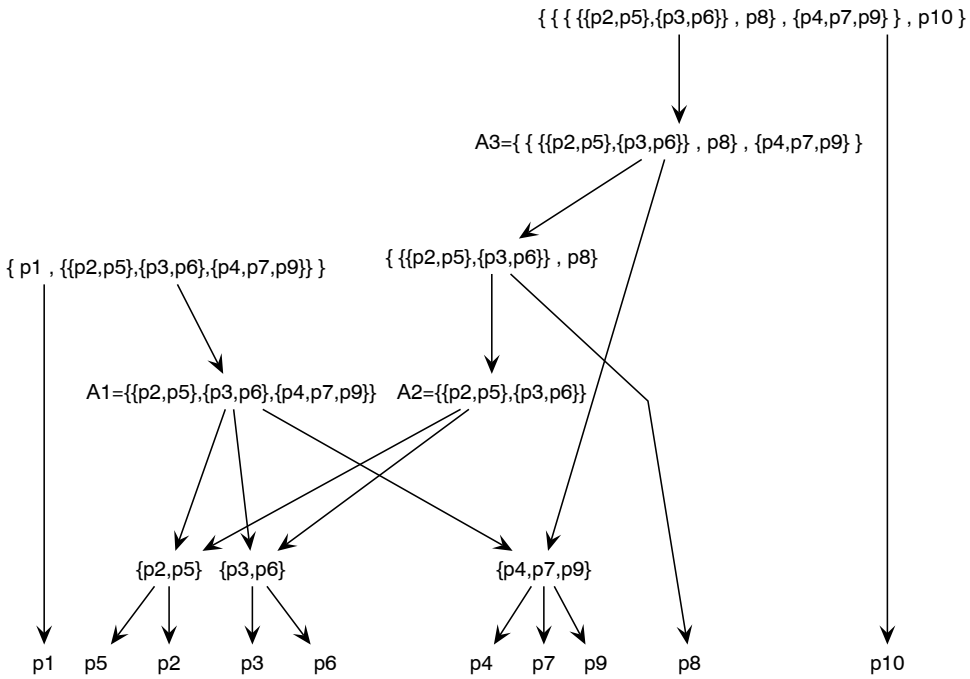

Figure 15: Structure of set *Nodes* for the net in Figure 2 immediately after OR nodes have been merged (start of line 21)

ple, $replace(1, 2, \{\{1, 3\}, \{3\}\})$ gives set $\{\{2, 3\}, \{3\}\}$. And $replace(\{1\},\{2\},\{\{1\},\{1,3\}\})$ gives set $\{\{2\}, \{1, 3\}\}$.

$$
\begin{aligned}
replace(x, y, \emptyset) &= \emptyset \\
replace(x, y, z) &= z \quad \text{if } z \text{ is not a set} \\
replace(x, y, \{z\} \cup S) &= \begin{cases} \{ y \} \cup replace(x, y, S), & \text{if } z = x \\ \{ z \} \cup replace(x, y, S), & \text{if } z \neq x \text{ and } z \text{ is not a set} \\ \{ replace(x, y, z') \mid z' \in z \} \cup replace(x, y, S), & \\ & \text{if } z \neq x \text{ and } z \text{ is a set} \end{cases}
\end{aligned}
$$

Figure 15 shows the structure of set *Nodes* for the example net in Figure 2 after OR nodes have been merged.

Nesting AND nodes. After the previous phases, some AND node a can be a subset of another AND node b, i.e., every OR child of a is also an OR child of b. For example, for the net in Figure 2, after merging OR nodes, the AND node A2 created for t2 is a subset of the AND node A1 created for t1, since both share OR nodes {p2,p5} and {p3,p6} (see Figure 15). In that case, the common OR children of a and b have two AND parents, which violates the statechart hierarchy constraints.

In this phase, an AND node a which is subset of another AND node b is nested inside b; this way a statechart hierarchy is obtained. For example, AND

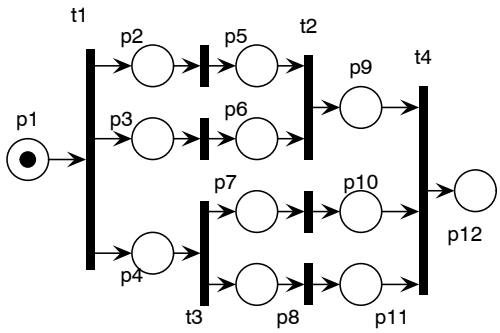

Figure 16: Petri net for which the AND nodes created for t2 and t3 can be nested in the AND nodes created for t1 and t4, respectively

node A2 in Figure 15 is nested inside A1. Nesting is done by replacing b by an AND node b' that contains the OR parent node of a plus all (OR) children of b that are not contained in a. The general case is a bit more complex, since several AND nodes might be nested inside the same AND node. For example, in Figure 16 both the AND nodes created for t2 and t4 can be nested inside the AND nodes for t1 and t4. Nesting fails if the new node structure would contain a cycle. Note that nesting does not guarantee a valid statechart hierarchy; for example, applying the algorithm to the net in Figure 6, after the nesting phase OR node {p5} still has two AND parents.

We discuss this part line by line. First, AND nodes are collected in set *tovisit* (l. 21). Next, the actual nesting is done by processing AND nodes in *tovisit* one by one in the while loop started at l. 22. During processing, each AND node X is replaced by another AND node X'. Node X' is equivalent to X, i.e., it has the same BASIC descendants as X, but X' is constructed using already processed AND nodes, which can be nested inside X'. To ensure a proper nesting, it is required that lower bound AND nodes are processed before other AND nodes (l. 23). An AND node a is defined to be a lower bound of *tovisit* if and only if there is no unprocessed AND node a' which is contained by a and moreover there is no unprocessed AND node b that is a subset of a:

$$a \text{ is a lower bound of } tovisit \stackrel{\text{df}}{\Leftrightarrow} \nexists a' \in tovisit : contains(a', a) \wedge \nexists b \in tovisit : b \subset a.$$

The lower bound AND node is put in variable X (l. 23).

For the construction of X', already processed AND nodes are used that *cover* X, i.e. whose BASIC descendants are also BASIC descendants of X (l. 24). These nodes are nested inside X'. Using function *AND_cover* the processed AND nodes that cover X are collected in set *AND*. To ensure a proper nesting, every AND node a must be maximal, i.e., there is no other processed AND node $a' \in AND$ such that a' contains a or $a \subset a'$. This leads to the following formal definitions for function *AND_cover*:

$$AND\_cover(X, tovisit) := maximal(\{A \in Nodes \setminus tovisit \mid isAND\_cover(A, X)\})$$

where

$$
\begin{array}{rcl}
\mathit{isAND\_cover}(A, X) & \Leftrightarrow & \mathit{isAND}(A) \wedge \mathit{flatten}(A) \subseteq \mathit{flatten}(X) \\
\mathit{maximal}(S) & := & \{s \in S \mid \nexists s' \in S : \mathit{contains}(s, s') \vee s \subset s'\} \\
\mathit{flatten}(\varnothing) & = & \varnothing \\
\mathit{flatten}(\{\, x \,\} \cup S) & = & \begin{cases} \{\, x \,\} \cup \mathit{flatten}(S), \text{ if } x \text{ is not a set} \\ \mathit{flatten}(x) \cup \mathit{flatten}(S), \text{ otherwise} \end{cases}
\end{array}
$$

Note that function *flatten* is applied recursively to all elements of in its argument. For example, $\mathit{flatten}(\{1, \{\{2, \{3\}\}\}\})$ gives $\{1, 2, 3\}$.

To illustrate *AND_cover*, suppose the nodes in Figure 15 are processed. Then $\mathit{AND\_cover}(A1, \{A1, A3\}) = \{A2\}$. However, $\mathit{AND\_cover}(A1, \{A1\}) = \{A2\}$. In the latter case, $A3$ is not an AND cover of $A1$, since $A3$ has p8 as BASIC descendant, which is not a BASIC descendant of $A1$.

Next (l. 25), set *uncovered* is computed, which contains the OR children of X that do not have any ancestors in set *AND*. These OR children are used in the construction of X', to ensure that X' covers X completely, i.e., that X and X' have the same BASIC descendants.

Then (l. 26), if either some nodes in X are not covered (so $\mathit{uncovered} \neq \varnothing$) or more than one AND node covers X (so $|\mathit{AND}| > 1$), then a new node X' is constructed. Otherwise (l. 32), there is only one AND node covering X completely (if $\mathit{uncovered} = \varnothing$, then $|\mathit{AND}| = 1$ by definition of *uncovered*), and X' is simply $\mathit{pick}(\mathit{AND})$ (l. 33).

If X' is to be constructed (l. 27-31) then the following steps have to be taken. First, the OR parents of the AND nodes in *AND* are retrieved and put in set *OR* (l. 27). Nodes in *OR* will become OR children of the new node X'. However, if a node $n \in \mathit{OR}$ already contains X, replacing X by X' would cause n to be both ancestor and descendant of X', causing a cycle in the node structure.

To illustrate this, consider the net in Figure 17. Figure 18 shows the structure of set *Nodes* immediately after the OR nodes have been merged. There are two AND nodes, A1={{p2},{p3},{p4}}, and A2={{p2},{p3}}. These two AND nodes are contained in OR node X1={{p1},A1,A2}. Note that there is no other OR node containing A1 or A2. Next, the while loop started at line 22 is executed. Suppose node A2 is processed before A1. Processing node A2 does not change the structure of Figure 18, but the subsequent processing of node A1 does (see Figure 19). Then immediately after line 24, $\mathit{AND} = \{\text{A2}\}$. So *OR* becomes X1. Then, X' ought to become {X1,{p4}} (visualised as the black dot in Figure 19). However, replacing A1 by X' would cause a cycle, as indicated in Figure 19.

Next (l. 31), AND node X' is constructed, the children of which are the OR nodes in *OR* together with the OR nodes in *uncovered*. Note that if $\mathit{uncovered} = X$, then $X' = X$.

After line 34, a new node X' has been constructed, and X can be replaced by X' (l. 39). The only complication arises if X' is an already existing node, so $X' \in \mathit{Nodes}$. Then X' has already an OR parent $o_{X'}$. So, replacing X by X' implies that X' gets two parents (the parent of X and $o_{X'}$). To avoid this, before replacing X by X', the OR parent $o_{X'}$ needs to be merged with the OR parent of X. For example, if AND node $A1$ in Figure 15 is processed, while nodes $A2$ and $A3$ have already been processed, then $X' = A3$ (even though $A3 \notin \mathit{AND}$, as explained above when defining *AND_cover*). Then, $A1$ can only

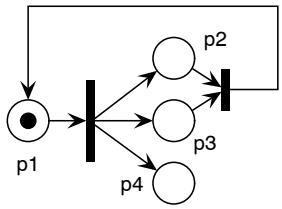

Figure 17: Petri net for which the procedure fails at line 29

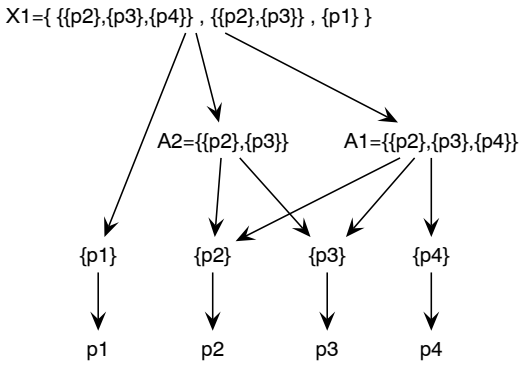

Figure 18: Structure of *Nodes* for the net in Figure 17 immediately before line 21

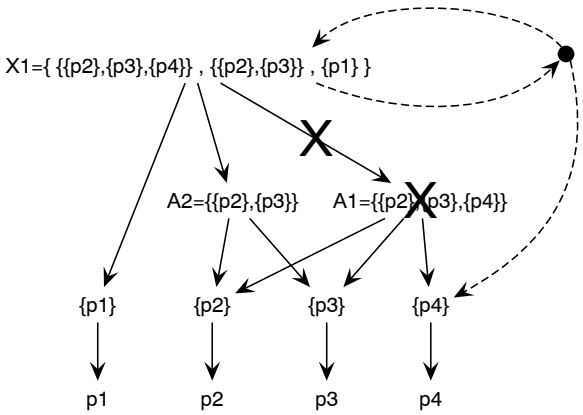

Figure 19: Structure of *Nodes* for the net in Figure 17 while processing A2 as X in the iteration started at line 22. The blackdot denotes X'. Dotted lines indicate the membership relation if X' would replace A2

be replaced by $A3$ if their OR parents have been merged, which ensures that nodes p1 and p10 share the same OR parent after replacement of A1.

Checking the statechart. Even though the constructed statechart structure has no cycles in its node hierarchy, still it can be illegal. Therefore, some checks are done next.

The first check tests for absence of multiple parents for the same node (l. 42). For some nets, e.g. in Figure 6, after the while loop started at line 22 has ended, still some AND nodes may share some OR children with other AND nodes. In that case the algorithm fails at line 42.

It is easy to check that if AND nodes cannot be nested, the subset relation on AND nodes does not induce a tree, that is, some OR node is child of two different AND nodes. However, there is a statechart variant that allows such overlapping nodes [11]. In that statechart variant, for the net in Figure 6 a bisimilar statechart does exist; see [5].

The second check tests whether hyperedges have consistent sources and targets (l. 43). For example, for the net of Figure 5 the algorithm will construct a node structure in which the OR parent of p4 is ancestor of the OR parents of p2 and p3 because of t3 and t4. Thus, nodes p2, p3, and p4 are inconsistent. Hence, hyperedge t1 is inconsistent.

Creating the children relation. After the previous phase, if the algorithm did not fail, a valid statechart structure has been constructed. In the final phase, the *type* and *children* relations have to be derived from the structure of the nodes. Figure 20 gives the code for deriving the *type* and *children* relation.

The children of a node are the elements it has in its set. Thus, $x \in children(y)$ if and only if $x \in y$.

For deriving the *type* relation, observe that BASIC nodes are atoms, whereas composite nodes are sets. Next, OR nodes only contain AND and BASIC nodes whereas AND nodes only contain OR nodes. This implies that the nesting depth of elements in an OR node differs from that in an AND node. More precisely, it is easy to check (for example using Figure 12) that OR nodes have an uneven depth whereas AND nodes have an even depth.

```
for n ∈ Nodes do
    if isBASIC(n) then
        type(n) := BASIC
        children(n) := ∅
    else if isAND(n) then
        type(n) := AND
        children(n) := { x | x ∈ n }
    else if isOR(n) then
        type(n) := OR
        children(n) := { x | x ∈ n }
    end if
end for
```

Figure 20: Code for deriving the *type* and *children* relations

The predicates used in Figure 20 are defined as follows:

$$\begin{aligned} \mathit{isBASIC}(x) &\Leftrightarrow x \text{ is an atom} \\ \mathit{isAND}(x) &\Leftrightarrow x \text{ is a set and } \mathit{depth}(n) \text{ is even} \\ \mathit{isOR}(x) &\Leftrightarrow x \text{ is a set and } \mathit{depth}(n) \text{ is uneven.} \end{aligned}$$

Function *depth* returns the maximum depth of a set:

$$\begin{aligned} \mathit{depth}(\emptyset) &= 0 \\ \mathit{depth}(\{\, x \,\} \cup S) &= \mathit{max}(1, \mathit{depth}(S)), \quad \text{if } x \text{ is an atom} \\ \mathit{depth}(\{\, x \,\} \cup S) &= \mathit{max}(1 + \mathit{depth}(x), \mathit{depth}(S)), \quad \text{if } x \text{ is a set.} \end{aligned}$$

For example, $\mathit{depth}(\{1, \{\{2, \{3\}\}, 4\}\} = 4$.

4.2 Correctness

Before proving the correctness of the algorithm, we show the correctness of some of the assumptions made in the algorithm.

Lemma 4.1 *Given a non OR node n, so $n \in \mathit{BASICNodes} \cup \mathit{ANDNodes}$. Initially (l. 9), n has only one parent. At the end of each iteration of the for loop started at line 11, n still has only one parent.*

Proof. By induction on the depth of nodes. The base case follows immediately from the definition in line 9. For the induction case, say in some iteration of the for loop started at line 11 two OR nodes x and y are merged. Take an arbitrary node n that is member of x or y. By the induction hypothesis, n has only one parent, so if n is in x, it cannot be in y and vice versa. So x and y are disjoint: $x \cap y \neq \emptyset$. Next, x and y are merged by replacing each occurrence of x or y by $x \cup y$. After merging, n still has only parent, namely $x \cup y$. (So n might gain additional brothers.) □

Lemma 4.2 *At lines 12 and 13, function find_lca$^+$ yields a singleton.*

Proof. We only show the proof for line 12; the proof for line 13 goes by similar reasoning. Function *find_lca*$^+$ returns all OR nodes that subsume the unique OR node constructed for $\mathit{source}(h)$ at line 9. By Lemma 4.1, only one such OR node exists. □

The next lemma motivates why hyperedges in the for loop of line 11 can be processed in arbitrary order.

Lemma 4.3 *Let h_1, h_2 and h_3 be three hyperedges. Then h_1, h_2 and h_3 can be processed in arbitrary order in the for loop started at line 11: the outcome (a merged OR node or failure is the same).*

Proof. Follows immediately from the associativity of $\cup$ (lines 12 and 13). □

The following lemma shows that the for loop started at line 11 preserves the subset relation between AND nodes. This motivates why OR nodes are merged before AND nodes are nested (so why the while loop started at line 22 follows the for loop started at line 11 and not the other way around).

Lemma 4.4 *Let a, a' be two AND nodes. If $a \subseteq a'$ at the beginning of an iteration of the for loop started at line 11, then after the iteration $a \subseteq a'$.*

Proof. By construction, a and a' only have OR members. If some OR node $x \in a$ is merged with another OR node y, then at lines 17 and 18 a global replacement of x and y by $x \cup y$ is done. So after the iteration, $x \cup y \in a$ and $x \cup y \in a'$. □

For the following lemma, we first introduce some terminology. Two hyperedges h_1 and h_2 *overlap* if the source or target set of one is a subset of the source or target set of the other:

$$\begin{aligned} \mathit{overlap}(h_1, h_2) \quad \Leftrightarrow \quad & \exists\, S_1, S_2 \subseteq N : \\ & (S_1 = \mathit{source}(h_1) \vee S_1 = \mathit{target}(h_1)) \\ & \wedge\, (S_2 = \mathit{source}(h_2) \vee S_2 = \mathit{target}(h_2)) \\ & \wedge\, (S_1 \subseteq S_2 \vee S_2 \subseteq S_1) \end{aligned}$$

We now show that if the algorithm terminates successfully by returning a statechart, that the sources and targets of overlapping hyperedges share the same single root node.

Lemma 4.5 *Given a connected Petri net (P, T, F, p_i) on which the algorithm terminates successfully. Let h_1 and h_2 be two overlapping hyperedges in the statechart returned at line 45. Let n_1 be a node in $\mathit{source}(h_1) \cup \mathit{target}(h_1)$ and n_2 a node in $\mathit{source}(h_2) \cup \mathit{target}(h_2)$.*

Then n_1 and n_2 share the same root r, i.e., r has no parent and $\{n, n'\} \subseteq \mathit{children}^*(r)$.

Proof. For the definition of *overlap*, we have either (i) $S_1 \subseteq S_2$ or (ii) $S_2 \subseteq S_1$.

For case (i) we argue as follows (case (ii) is by similar reasoning):

- If $S_1 = S_2$ then for both sets the same AND node a is created at line 9.
- If $S_1 \subset S_2$, then S_2 cannot be a singleton, so S_2 is covered by some AND node a.
  - If S_1 is a singleton $\{s\}$, then s has a unique OR parent o (Lemma 4.1). Node o is a child of a (l. 8).
  - If S_1 is not a singleton, then S_1 is also covered by some AND node a' that is nested in a (in the while loop that begins at line 22).

In both subcases, AND node a has a unique OR parent o (by Lemma 4.1) whose descendants include n_1 and n_2. From Lemma 4.1 and the fact that the algorithm did not fail at line 42, it follows that each descendant of o has one parent. Thus, the root of o is the unique root of n_1 and n_2.

□

Next, we generalise the previous lemma by showing that in case of successful termination, any pair of nodes in the returned statechart share the same single root.

Lemma 4.6 *Given a connected Petri net (P, T, F, p_i) on which the algorithm terminates successfully. Let n, n' be two BASIC nodes in the statechart returned at line 45. Then n and n' share the same root r, i.e., r has no parent and $\{n, n'\} \subseteq children^*(r)$.*

Proof. First, we observe that each node has only one root (using Lemma 4.1 and line 42). Since the input Petri net (P, T, F, p_i) is connected, BASIC nodes n and n' are connected by a path h_1, h_2, .., h_k of hyperedges, where for every h_i, h_{i+1}, $1 \leq i \leq k$, $overlap(h_i, h_{i+1}$, since otherwise the algorithm would fail at line 42. Consider an arbitrary pair of hyperedges h_i, h_{i+1} on this path, where $1 \leq i < k$. By Lemma 4.5, any pair of nodes $n_i \in source(h_i) \cup target(h_i)$ and $n_{i+1} \in source(h_{i+1}) \cup target(h_{i+1})$ share the same, single root. The result that n and n' share the same root then follows directly. □

Theorem 4.1 (validity) *Given a connected Petri net (P, T, F, p_i). If the algorithm terminates successfully (so line 45 is reached), a valid statechart is returned.*

Proof. Simple checking of statechart definition. For the *children* relation, we have to show that:

1. each node has at most one parent,
2. there is a single root, i.e., there is exactly one node that has no parent,
3. the root node is ancestor of all other nodes.

Point 1 follows from Lemma 4.1 and the fact that algorithm did not fail at line 42. Points 2 and 3 follow from Lemma 4.6. □

Lemma 4.7 *The statechart returned at line 45 is wellformed.*

Proof. Simple checking of the definition of wellformedness. Consistency follows from line 43. Completeness follows from the construction of AND nodes (l. 8) and the merging of OR nodes. □

Next, we show the correctness of the translation.

Theorem 4.2 (correctness) *Given a connected Petri net PN such that the algorithm returns a statechart $SC(PN)$. Then $SC(PN)$ bisimulates PN.*

Proof. Observe that by Lemma 4.7 $SC(PN)$ is wellformed. Next, from lines 2–7, it follows that $PNToSC(SC(PN))$ is a Petri net isomorphic to PN. Since isomorphism implies bisimulation [8], the result then follows immediately from Theorem 3.1. □

The algorithm is not complete: it can fail even though there does exist a bisimilar wellformed statechart. For example, the algorithm fails on the net in Figure 21, because two AND nodes are created with overlapping OR nodes that cannot be nested according to the algorithm. Of course, an equivalent statechart can be constructed by simply merging the OR nodes introduced for

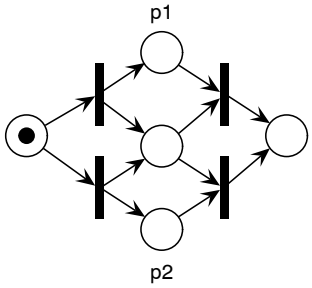

Figure 21: Petri net on which the algorithm fails but which does have an equivalent statechart

p1 and p2. The algorithm can be refined and made complete, by (arbitrary) merging OR nodes in case of a failure. But the complexity will then be worse (below we estimate the complexity).

However, the non-completeness of the algorithm does not appear to be a real limitation in practice. Statecharts which the algorithm fails to construct have some OR child o with unconnected children, i.e., there is some pair of children that are connected by an undirected sequence of hyperedges whose scope is o. For example, the equivalent statechart for the net in Figure 21 has an OR node o in which children n_{p_1} and n_{p_2} are not related by any hyperedge whose scope is o. It is easy to check that the algorithm only returns statecharts in which every OR node has connected children. (For such statecharts, the algorithm is complete.)

However, such statecharts are not very obvious to define, though they are definitely not excluded by the statechart syntax. Children of an OR node that are unconnected are unrelated. From a modelling point of view, it therefore does not seem to make sense to group those unrelated children under one OR node, which would suggest that they are related. Examining example statecharts from the literature, we indeed did not encounter any statechart in which an OR node had unconnected children. So the fact that the algorithm is non-complete does not seem a very severe limitation in practice.

Complexity. The worst-case space complexity of the algorithm is $O(|T|)$. In the worst case, there are $|P|$ BASIC nodes, at most $2 \cdot |T|$ AND nodes, and at most $|P| + 2 \cdot |T|$ OR nodes. Since we only consider connected Petri nets, there are no isolated places. Thus, the number of places $|P|$ is bounded by the number of transitions. Consequently, the number of nodes is linear in the number of transitions $|T|$.

The worst-case time complexity of the algorithm in Figure 11 is $O(|T|^2)$, because of the while loop started at line 22 and finding a lower bound AND node in this loop (l. 23). In the worst case, there are at most $2 \cdot |T|$ AND nodes. Thus, the while loop started at line 22 is done at most $2 \cdot |T|$ times. Next, finding a lowerbound AND node is polynomial in the number of nodes, which is bounded by $|T|$. Thus, the worst-case time complexity of the algorithm is $O(|T|^2)$.

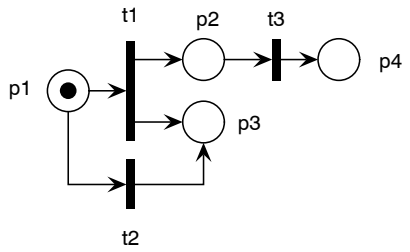

Figure 22: Safe Petri net for which no structure-preserving equivalent statechart exists

4.3 Relation with Petri nets

As we pointed out in Section 2, if a Petri net is unsafe or unbounded, there is no equivalent statechart. We now show how the algorithm fails for such nets.

Lemma 4.8 *If the Petri net is unbounded, then there is a potential cycle, i.e, the algorithm fails at line 15 or 29.*

Proof.(Sketch) If the net is unbounded, then there is a reachable marking M and a sequence of transitions $\sigma = t_1, t_2, .., t_n$ such that $M \xrightarrow{t_1} M_1 \xrightarrow{t_2} .. \xrightarrow{t_n} M'$, such that $M \neq M'$ and for every $p \in P$, $M(p) \leq M'(p)$ [18]. Sequence σ is called a token generator. In M' some place p contains more tokens than in M: $M(p) < M'(p)$. Some transition t in the sequence must have filled p. Denote by P_o the maximal set of output places of t, $P_o \subseteq t\bullet$, such that for every $p_o \in P_p$, $M(p_o) = M'(p_o)$.

Since σ is cyclic, after processing all hyperedges corresponding to transitions in σ minus t, the nodes created for the input places of t and the places in P_o all share the same root OR node o (*). Next, processing h_t will cause a cycle in the child relation, since for the output places of t an AND node a has been created, one child of which will be o (possibly after nesting). But by (*), o is also the source lca^+ of the nodes created for the input place of t. Merging the target lca^+ with the source lca^+ will cause o to be parent of a in addition to child. Hence a cycle in the child relation results.

□

The reverse implication does not hold. Figure 22 shows a Petri net that is a slight variation on the unbounded net of Figure 4. Though the net in Figure 22 is safe, the algorithm will fail because it finds a cycle in the OR nodes. This example illustrates an important property of the algorithm: presets and postsets of transitions are treated in exactly the same way. Hence, the algorithm treats the net in Figure 22 exactly the same as the net in Figure 4, even though transition `t2` has its pre and postset swapped.

If the Petri net is unsafe but bounded, and the algorithm does not fail at lines 15 and 29, then it fails at lines 42 or 43. For the unsafe net in Figure 23, the algorithm fails at line 42 because the OR node containing the node created for `p5`, has two AND parents. Figure 5 shows an unsafe Petri net for which the algorithm fails at line 43, because the algorithm will create an OR node which has

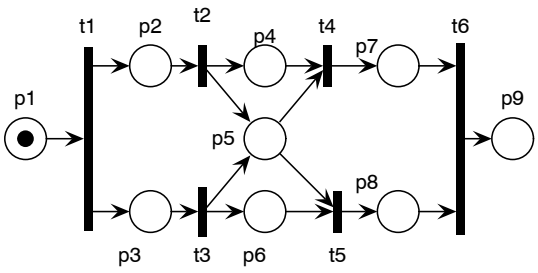

Figure 23: Unsafe Petri net on which the algorithm fails at line 42

as children, among others, BASIC node p4 and AND node {{p2,p5},{p3,p6}}. Clearly, the hyperedge created for t1 then has an inconsistent target, since neither p2 and p4 nor p3 and p4 are orthogonal.

Elsewhere [5] we have compared the class of nets that can be translated into a structure-preserving, equivalent statechart with existing classes of nets and statecharts.

5 Conclusion

An algorithm has been presented that maps a Petri net to a statechart. The translation is structure preserving: places map to BASIC nodes and transitions to hyperedges. Loosely speaking, the algorithm tries to impose an AND/OR hierarchy of nodes on a Petri net structure. The algorithm is polynomial in the size of the Petri net, and thus also efficient for large Petri nets.

The algorithm has been proven correct. However, it is not complete. But the algorithm only fails to construct wellformed statecharts in which some children of some OR node are not connected by any undirected sequence of hyperedges. We have not found any example of such a statechart in the literature, though the standard definition definitely does not exclude it. Apparently, such a statechart is not obvious to draw. So the limitation of non-completeness does not seem very severe in practice.

There are several directions for further work. First, the algorithm can be extended to deal with non-structure-preserving translations as well, to allow for example the translation of the net in Figure 6 into the statechart of Figure 8. In particular, it is interesting to extend the algorithm with repair actions, that modify the Petri net in case of a failure. For example, the net in Figure 5 could be modified into an equivalent net on which the algorithm does not fail by duplicating place p10 and letting t4 (or t3) enter this duplicate place, rather than p10. Next, data can be considered by looking at coloured Petri nets and statecharts with local variables and guard and action labels on the hyperedges.

On the more applied side, the algorithm can be used as a foundation for implementing model transformations between Petri net-like models and statechart models. For example, UML activity diagrams [35] are a visual language whose syntax resembles Petri net syntax quite closely. The algorithm can be used to transform a UML activity diagram into a UML statechart.

References

[1] P. Aczel. *Non-Well-Founded Sets*. Stanford Center for the Study of Language and Information, 1988. CSLI Lecture Notes number 14.

[2] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. J. Wiley, 1995.

[3] D. Drusinsky and D. Harel. On the power of bounded concurrency i: Finite automata. *Journal of the ACM*, 41(3):517–539, 1994.

[4] R. Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. PhD thesis, Centre for Telematics and Information Technology, University of Twente, 2002. Available at `http://www.ctit.utwente.nl/library/phd/eshuis.pdf`.

[5] R. Eshuis. On nets with structured concurrency. Beta Working Paper Series, Eindhoven University of Technology, 2005.

[6] R. Eshuis and J. Dehnert. Reactive Petri nets for workflow modeling. In W.M.P. van der Aalst and E. Best, editors, *Proc. 24th International Conference on the Applications and Theory of Petri Nets (ICATPN) 2003*, Lecture Notes in Computer Science 2679, pages 296–315. Springer, 2003.

[7] R. Eshuis and R. Wieringa. Comparing Petri net and activity diagram variants for workflow modelling – a quest for reactive Petri nets. In H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors, *Petri Net Technology for Communication Based Systems*, Lecture Notes in Computer Science 2472, pages 321–351. Springer, 2003.

[8] R.J. van Glabbeek. The linear time — branching time spectrum I. The semantics of concrete, sequential processes. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 3–99. North-Holland, 2001.

[9] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[10] D. Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[11] D. Harel and C.-A. Kahana. On statecharts with overlapping. *ACM Transactions on Software Engineering and Methodology*, 1(4):399–421, 1992.

[12] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M.B. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, 1990.

[13] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Trans. on Software Engineering and Methodology*, 5(4):293–333, 1996.

[14] D. Harel, A. Pnueli, J. P. Schmidt, and S. Sherman. On the formal semantics of statecharts. In *Proceedings of the Second IEEE Symposium on Logic in Computation*, pages 54–64. IEEE, 1987.

[15] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: the STATEMATE approach*. McGraw-Hill, 1998.

[16] G. Huszerl, I. Majzik, A. Pataricza, K. Kosmidis, and M. Dal Cin. Quantitative analysis of uml statechart models of dependable systems. *Computer Journal*, 45(3):260–277, 2002.

[17] K. Jensen. *Coloured Petri Nets. Basic concepts, analysis methods and practical use*. EATCS monographs on Theoretical Computer Science. Springer, 1992.

[18] R.M. Karp and R.E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969.

[19] M. Kishinevsky, J. Cortadella, A. Kondratyev, L. Lavagno, A. Taubin, and A. Yakovlev. Coupling asynchrony and interrupts: Place chart nets. In P. Azéma and G. Balbo, editors, *Proc. 18th International Conference on the Applications and Theory of Petri Nets (ICATPN) 1997*, Lecture Notes in Computer Science 1248, pages 328–347. Springer, 1997.

[20] The Mathworks. Stateflow and stateflow coder users guide, 2005. Available at `http://www.mathworks.com`.

[21] J. Merseguer, S. Bernardi, J. Campos, and S. Donatelli. A compositional semantics for UML state machines aimed at performance evaluation. In M. Silva, A. Giua, and J.M. Colom, editors, *Proceedings of the 6th International Workshop on Discrete Event Systems (WODES)*, pages 295–302. IEEE Computer Society Press, 2002.

[22] T. Murata. Petri nets: Properties, analysis, and applications. *Proc. of the IEEE*, 77(4):541–580, 1989.

[23] E.-R. Olderog. *Nets, Terms and Formulas*. Cambridge Tracts in Theoretical Computer Science 23. Cambridge University Press, 1991.

[24] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962.

[25] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science 526, pages 244–265. Springer, 1991.

[26] A.V. Ratzer, L. Wells, H.M. Lassen, M. Laursen, J.F. Qvortrup, M.S. Stissing, M. Westergaard, S. Christensen, and K. Jensen. CPN Tools for editing, simulating, and analysing coloured Petri nets. In W.M.P. van der Aalst and E. Best, editors, *Proc. 24th International Conference on the Applications and Theory of Petri Nets (ICATPN) 2003*, Lecture Notes in Computer Science 2679, pages 450–462. Springer, 2003.

[27] W. Reisig. *Petri Nets: An Introduction*. Number 4 in EATCS Monographs on Theoretical Computer Science. Springer, 1985.

[28] W. Reisig and G. Rozenberg, editors. *Lectures on Petri nets I: Advances in Petri nets*, Lecture Notes in Computer Science 1492. Springer, 1998.

[29] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design*. Prentice-Hall, 1991.

[30] J.A. Saldhana, S.M. Shatz, and Z. Hu. Formalization of object behavior and interactions from uml models. *International Journal of Software Engineering and Knowledge Engineering*, 11(6):643–673, 2001.

[31] M. Schnabel, G. Nenninger, and V. Krebs. Konvertierung sicherer petrinetze in statecharts (in German). *Automatisierungstechnik*, 47(12):571–580, 1999.

[32] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object Oriented Modeling*. John Wiley & Sons, 1994.

[33] H. Stöerrle. *Models of Software Architecture*. PhD thesis, Ludwig-Maximilians-Universität München, 2000.

[34] UML Revision Taskforce. *OMG UML Specification v. 1.5*. Object Management Group, 2003. OMG Document Number formal/2003-03-01. Available at `http://www.uml.org`.

[35] UML Revision Taskforce. *UML 2.0 Superstructure Specification*. Object Management Group, 2003. OMG Document Number ptc/03-07-06. Available at `http://www.uml.org`.

[36] A. Wąsowski and P. Sestoft. On the formal semantics of visualSTATE statecharts. Technical Report TR-2002-19, IT University of Copenhagen, 2002.