On nets with structured concurrency

On Nets with Structured Concurrency

Rik Eshuis

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

Abstract. An extension of Petri nets with a statechart-like AND/OR state hierarchy is defined and studied. The resulting net variant, statechart nets, is shown to coincide, under certain conditions, with a strict subclass of safe nets in which concurrency is structured. Next, syntactic constraints on statechart nets are defined that guarantee absence of deadlocks and livelocks. Such syntactic constraints are hard to give for ordinary Petri nets. Together, these results give more insight into the expressive power and usefulness of AND/OR state hierarchies, and into the differences between statecharts and Petri nets.

1 Introduction

Petri nets [19, 21] are a popular formalism for modelling concurrent systems. One of the most interesting subclasses of Petri nets are safe nets, in which each place holds at most one token during execution. Properties like reachability and deadlock can be more efficiently decided for safe nets than for ordinary nets [4]. Moreover, safe nets are more easy to understand and implement than ordinary nets.

Safeness is a property that is decided by computing the semantics (reachability graph) of a Petri net. The complexity of deciding safeness or k-boundedness in general is PSPACE complete [9, 16] and takes in practice exponential time. Fortunately, there are ways to ensure safeness, thus rendering a safeness check superfluous. For example, in condition/event nets [21] and elementary nets [22] a transition cannot fire if one of its output places is filled. The standard Petri net firing rule does not have this restriction. Condition/event and elementary nets ensure safeness in a semantic way.

Another, less obvious, way to ensure safeness is to use an AND/OR hierarchy of states, which is one of the core features of statecharts [11]. AND states denote concurrency whereas OR states denote sequential behaviour. States can only be concurrent at the semantic level (so marked with a token in the same state) if they are specified as potentially concurrent in the syntax using an enclosing AND state. In particular, a state cannot be concurrent with it self. Thus, an AND/OR state hierarchy ensures safeness in a syntactic way.

Apart from the fact that every statechart is safe by definition, the expressive power of concurrency models with an AND/OR state hierarchy is still unclear. For example, while condition/event and elementary nets are known to correspond

to safe nets in which every enabled transition has none of its output places filled, it is still unclear what class of safe nets is induced by an AND/OR hierarchy of states. Also, it is unclear what exactly the benefits are of using an AND/OR state hierarchy.

This paper defines and studies an extension of Petri nets with a statechart-like AND/OR state hierarchy imposed on places. The resulting net variant, statechart nets, uses a firing rule that closely resembles the basic statechart execution semantics. In contrast to an enabled Petri net transition, an enabled statechart net transition might remove tokens from non-input places, thus violating the locality principle which states that only tokens in input places can be removed [6]. Despite this major difference, we show that under certain conditions statechart nets coincide with a strict subclass of safe nets. Nets in this subclass have a restricted form of concurrency, since cross-synchronisation between parallel branches does not occur. Thus, concurrency in these nets is structured, in a similar vein as goto-less programs are structured. This result gives more insight into the expressive power of AND/OR state hierarchies.

Next, the paper defines syntactic constraints on statechart nets that guarantee absence of deadlocks and livelocks. Such syntactic constraints are hard to define for ordinary Petri nets. Thus, one benefit of using an AND/OR state hierarchy is that it enables the formulation of such constraints. Under mild conditions, the constraints are sufficient to guarantee that for every pair of potentially concurrent places there is a reachable state in which these two places are indeed concurrent. This allows for an efficient syntactic decision procedure for reachability properties.

The remainder of this paper is organised as follows. Section 2 recalls some standard Petri net definitions. Next, the syntax and semantics of statechart nets are defined in Section 3. Also, structured concurrency nets are defined as the subclass of safe nets for which an equivalent statechart net exists. In Section 4 we compare structured concurrency nets with existing net subclasses. We show that structured concurrency nets do not coincide with any of these. This justifies their introduction as separate net subclass in this paper, and yields more insight into the expressive power of an AND/OR state hierarchy. In Section 4 we also study some proposed statechart extensions [12, 23] that relax the AND/OR hierarchy, and show that the resulting statechart net variants do not coincide with a subclass of Petri nets. In Section 5 we define structural constraints that guarantee absence of deadlock and livelock in statechart nets, making an expensive semantic analysis of these properties superfluous. Section 6 ends the paper with a discussion of related work and conclusions.

2 Preliminaries

We recall some basic definition of Petri nets [21]. A Petri net (place/transition net) is a tuple $PN = (P, T, F, M_0)$ where

- P is a finite set of places,
- T is a finite set of transitions, $P \cap T \neq \emptyset$,

– $F \subseteq (P \times T) \cup (T \times P)$ is a finite set of arcs, the flow relation, and
– $M_0 : P \longrightarrow \mathbb{N}$ is the initial marking.

Given a transition t, preset $\bullet t = \{\, p \in P \mid (p,t) \in F \,\}$ is the set of input places of t, whereas postset $t\bullet = \{\, p \in P \mid (t,p) \in F \,\}$ 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 $(n_{i+1}, n_i) \in F$.

Marking $M : P \longrightarrow \mathbb{N}$ enables transition t if and only if all of t's input places have a token: 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 $p \in P$:

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

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 a reachable marking of a Petri net (P, T, F, M_0) if and only if M is reachable from M_0.

A Petri net is *safe* if and only if for every reachable marking M and every place p, $M(p) \leq 1$. A net is *bounded* if the set of reachable markings is finite.

We only consider nets that have an initial marking that is safe, i.e. $M_0 : P \longrightarrow \{0, 1\}$, so each place is initially filled with at most one token.

3 Statechart nets

Statechart nets extend Petri nets with a statechart-like set of composite AND/OR nodes which are arranged in a hierarchy. The leaves of this hierarchy are places. This section formally defines the syntax and semantics of statechart nets. In the formal definitions, we borrow several concepts from statechart theory [14, 20, 13]. We show that if a statechart net satisfies certain syntactic constraints, it is bisimilar to the underlying Petri net.

Statechart nets can be considered as a simplified, less expressive version of statecharts [11]. The main difference with statecharts is that statechart transitions can connect both BASIC nodes (i.e., places) and composite (AND/OR) nodes, while in statechart nets a transition only connects places. Another difference is that statecharts typically have a reactive step semantics, whereas statechart nets have a token-game semantics, which is not reactive [8].

3.1 Syntax

A statechart net is a tuple $SN = (P, T, F, AND, OR, children, C_0)$ where

– $(P, T, F, C_0 \cap P)$ is a Petri net,

- AND is a set of AND nodes,
- OR is a set of OR nodes,
- $children : (AND \longrightarrow OR) \cup (OR \longrightarrow (P \cup AND))$ is a function that defines for each AND/OR node its immediate subnodes, and
- $C_0 \subseteq P \cup AND \cup OR$ is the initial configuration.

Configurations are explained in Section 3.2.

In the sequel, we use the term "nodes" to refer to both places and composite nodes, and let N denote the set of all places and composite nodes:

$$N \stackrel{\text{df}}{=} P \cup AND \cup OR.$$

We require that sets P, AND and OR are pairwise disjoint, so for example a place cannot be an AND node. Furthermore, we do not allow that an AND (OR) node has an AND (OR) node as subnode. This is not a severe restriction: an AND (OR) child c of an AND (OR) node n can be eliminated by letting the children of c become children of n.

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 places and composite nodes in a hierarchy [24]:

- There is one single composite node *root* that does not have any parents. For technical reasons, we require that *root* has type OR.

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

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

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

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

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

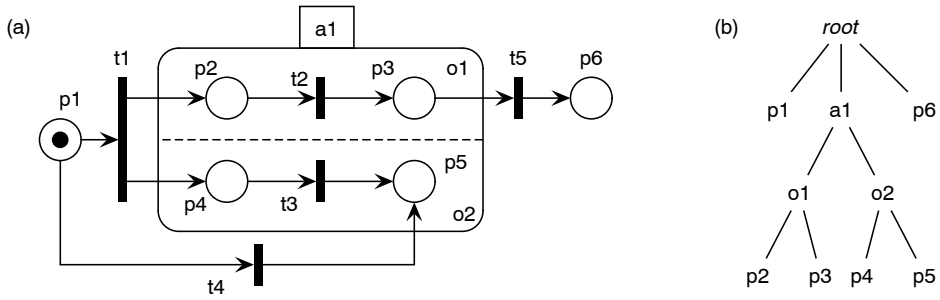

Fig. 1. Example statechart net (a) and its node hierarchy (b)

By the last constraint, each composite node or place n has a unique parent. We write $parent(n)$ to denote the composite node n' that has n as child. So $n' = parent(n)$ iff $n \in children(n')$.

We use the standard statechart notation for visualising AND/OR hierarchies, so OR children of AND nodes are separated by a dotted line, and the name of an AND node is specified in a box attached to the node [11]. For example, Fig. 1 shows a statechart net and its AND/OR hierarchy. The AND node has name a1. Note that *root* is by default not shown in a statechart net.

3.2 Semantics

For statechart nets, a marking is a set, consisting of places, AND nodes and OR nodes. However, some sets can be invalid. Every valid marking, called a configuration in statechart terminology, must satisfy several constraints.

Before we define these constraints, we need some additional statechart concepts. 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 Fig. 1, $lca(\{\mathsf{p2}, \mathsf{p3}\})$ is OR node o1, whereas $lca(\{\mathsf{p3}, \mathsf{p4}\})$ is AND node a1.

Given a set X of nodes, $lca^+(X)$ is the lowest OR node that is ancestor of every node in X. For example, in Fig. 1 $lca^+(\{\mathsf{p3}, \mathsf{p4}\}) = root$, since *root* is the OR parent of AND node a1.

Two nodes $x, y \in N$ are orthogonal if and only if x is neither ancestor nor descendant of y, and their lca is an AND node. In the example, nodes p2 and p4 are orthogonal, but nodes p2 and o1 are not (since p2 is child of o1) and neither are p2 and p3 (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 is descendant of ancestor of y, or x and y are orthogonal. Thus, for the statechart net in Fig. 1 set {p2,p4,o1} is consistent.

A *configuration* is a maximal, consistent set of nodes. Adding a node to a configuration would make it inconsistent. In the example, set {p1,*root*} is a configuration, as well as {p2,p4,o1,o2,a1,*root*}. Configurations are the valid global states of the statechart.

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

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

Like Petri nets, statechart nets change state (configuration) by taking enabled transitions. A transition t is *enabled* in configuration C if all its input places are active: $\bullet t \subseteq C$.

Defining the effect of taking a transition is a bit more complex than for ordinary Petri nets, since the next configuration has to be valid, i.e., satisfy the configuration constraints. To ensure this, we need the concept of scope. The *scope* of a transition t is the least common ancestor of its preset and postset:

$$scope(t) \stackrel{\text{df}}{=} lca^{+}(\bullet t \cup t\bullet).$$

Upon taking t, all descendants of the scope of t will be left. The configuration C' entered by taking t contains places in $t\bullet$, their ancestors, and nodes in the current configuration C that are ancestors of $scope(t)$:

$$C' \stackrel{\text{df}}{=} dcomp((C \setminus children^{*}(scope(t))) \cup t\bullet),$$

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 $parent(s) \in D$.

While the definition of C' ensures that nodes below $scope(t)$ are left, it does not guarantee that C' is a valid configuration. Transition t might enter an AND node a only partially by not entering any descendant of some child of a. For example, in the statechart in Fig. 1, transition t4 only partially enters AND node a1: no descendant of o1 is entered. To rule out such a transition, we require that each transition t be *target complete*: if t enters some AND node a, $t\bullet \cap children^{*}(a) \neq \emptyset$, then t also enters every child of a:

$$\begin{aligned} target\_complete(t) \stackrel{\text{df}}{\Leftrightarrow} \forall\, a \in children^{*}(scope(t)) : \\ a \in AND \wedge children^{*}(a) \cap t\bullet \neq \emptyset \Rightarrow \\ \forall\, n \in children(a) : children^{*}(n) \cap t\bullet \neq \emptyset. \end{aligned}$$

Symmetrically, we formulate a constraint specifying that a transition t should leave an AND node either completely or not at all. Though the execution semantics of statechart nets already enforces AND nodes to be left completely already, since all nodes below $scope(t)$ are left, this constraint is useful for two

reasons. First, it ensures that the statechart firing rule satisfies the Petri net locality principle [6]: only places in the preset of t and their ancestors are left. For instance, transition t5 in Fig. 1 violates this principle: if for example p5 is in the current configuration, taking t5 will cause the net to leave p5, even though it is not in the preset of t5. Second, without this constraint, a statechart net cannot be proven equivalent to its underlying Petri net (see Theorem 1 below).

A transition $t \in T$ is *source_complete* if and only if for each AND node a that it leaves, i.e., $children^*(a) \cap {\bullet}t \neq \emptyset$, it leaves each of a's children as well.

$$\begin{aligned} source\_complete(t) \stackrel{\text{df}}{\Leftrightarrow} \forall\, a \in children^*(scope(t)) : \\ a \in AND \wedge children^*(a) \cap {\bullet}t \neq \emptyset \Rightarrow \\ \forall\, n \in children(a) : children^*(n) \cap {\bullet}t \neq \emptyset. \end{aligned}$$

In the statechart net of Fig. 1, transition t5 is not source complete.

If a transition is both source and target complete, we call it *complete*. If a complete transition is taken in configuration C, the places left in C are those contained in ${\bullet}t$, and the places entered are in the postset $t{\bullet}$. Elsewhere we have proven this formally [7]. In Fig. 1, only transitions t1, t2, and t3 are complete.

In addition, each transition $t \in T$ should be *consistent*, i.e., it should have a consistent preset and a consistent postset:

$$consistent(t) \stackrel{\text{df}}{\Leftrightarrow} consistent({\bullet}t) \wedge consistent(t{\bullet}).$$

If a transition t has an inconsistent preset ${\bullet}t$, then there is no valid configuration enabling t, and if $t{\bullet}$ is inconsistent, the next configuration will be invalid. All transitions of the statechart net in Fig. 1 are consistent.

A statechart net SN is *wellformed* iff every transition $t \in T$ is consistent and complete:

$$wellformed(SN) \stackrel{\text{df}}{\Leftrightarrow} \forall\, t \in T : consistent(t) \wedge complete(t).$$

The statechart net in Fig. 1 is not wellformed. If transitions t4 and t5 are removed, the resulting statechart net will be consistent.

3.3 From statechart net to Petri net

By definition, each statechart net has an underlying Petri net. Deriving a Petri net from a statechart net is straightforward: simply drop the composite nodes from the statechart net and remove the composite nodes from the configuration. Function *SNtoPN* defines this formally:

$$SNtoPN((P, T, F, AND, OR, children, C_0)) \stackrel{\text{df}}{=} (P, T, F, C_0 \cap P)$$

The following theorem asserts the correctness of this function for wellformed statechart nets.

Theorem 1 (correctness). *Given a wellformed statechart net SN. The Petri net $SNtoPN(SN)$ bisimulates SN.*

Proof. Modify the statechart net SN into SN', by adding an initial place p_i and a transition t_i such that $\bullet t_i = \{p_i\}$ and $t_i\bullet = C_0 \cap P$. Next, apply Theorem 3.1 in [7].

In the remainder of this paper, we only consider wellformed statechart nets.

3.4 From Petri net to statechart net

Some Petri nets have a corresponding equivalent (bisimilar) statechart net. Naturally, such nets are safe.

A safe net PN is called a *structured concurrency net* if there exists a wellformed statechart net SN such that $SNtoPN(SN) = PN$. By Theorem 1, SN bisimulates PN. A structured concurrency net has a restricted form of concurrency, since the statechart AND/OR hierarchy cannot express cross-synchronisation between parallel branches, as explained in Section 4.1. Thus, concurrency in these nets is structured, just as goto-less programs are structured. In the next section, we study the expressiveness of structured concurrency nets.

Elsewhere we have formally defined a polynomial algorithm that maps a Petri net to a wellformed statechart net by imposing an AND/OR hierarchy on the places [7]. The variant of statecharts used there is isomorphic to statechart nets, so the algorithm and its proof of correctness carry immediately over to statechart nets. The algorithm fails if it cannot construct a statechart net. The algorithm is not complete, so it fails on some structured concurrency nets.

4 Expressiveness

In the previous section, we defined the class of structured concurrency nets. In the literature, several other Petri net subclasses have been defined, that are at first sight remarkably similar to structured concurrency nets. This section shows that nevertheless all these subclasses are different. This result gives more insight into the expressive power of an AND/OR state hierarchy.

4.1 Safe Petri nets

By definition, every structured concurrency net is safe. However, not every safe Petri net is a structured concurrency net. The net in Fig. 2 has a cross-organisation between parallel branches, which prevents that an equivalent statechart net exists. To see why, consider the statechart net in Fig. 3 that seems to have a cross synchronisation. However, this statechart net is not wellformed, because t is not complete, since its postset does not include any descendant of O2.

Another example of a safe Petri net which does not have an equivalent statechart net is shown in Fig. 4. Depending on whether t1 or t2 has fired, p3 is concurrent with p4 or not. This "conditional" concurrency between p3 and p4 cannot be captured in an AND/OR hierarchy.

So while an AND/OR hierarchy ensures safeness, not every safe net can be captured in such a hierarchy.

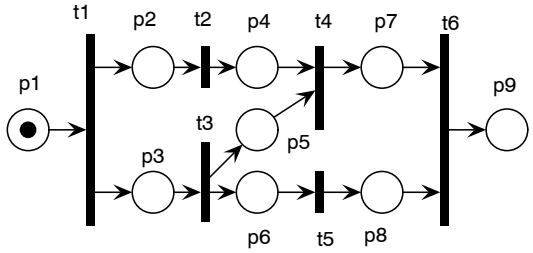

Fig. 2. Safe Petri net with cross-synchronisation for which no equivalent statechart net exists

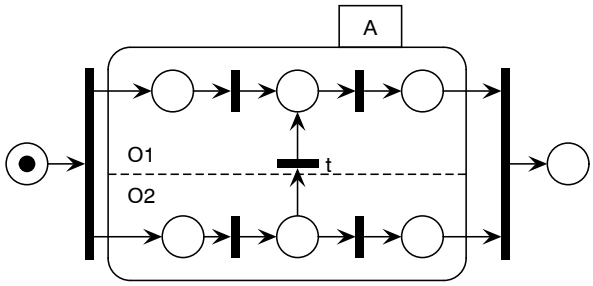

Fig. 3. Statechart net with a transition connecting parallel nodes

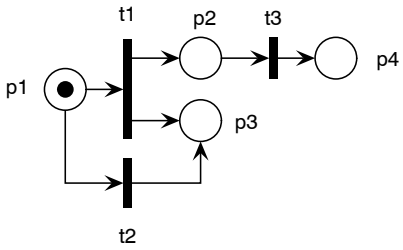

Fig. 4. Safe Petri net that is not a structured concurrency net

4.2 State machines

A state machine is a Petri net in which every transition t has one input place and one output place: $|\bullet t| = |t\bullet| = 1$. Thus, state machines represent sequential processes.

Some safe nets are covered by sequential (S-) components. An S-component of a net (P, T, F, M_0) is any net (P', T', F', M_0') such that

- (P', T', F', M_0') is a state machine,
- $P' \subseteq P$,
- $T' \subseteq T$,
- $F' = F \cap ((P' \times T') \cup (T' \times P'))$.
- For every $p' \in P'$ and $t \in T$, if $(p', t) \in F$ then $(p', t) \in F'$, and if $(t, p') \in F$ then $(t, p') \in F'$.
- For every $p \in P'$, $M_0'(p) = 1 \Leftrightarrow M_0(p) = 1$.

A net is S-coverable if each place is covered by some S-component [5]. It is well-known that every S-coverable net is safe. But not every safe net is S-coverable, as is illustrated by the safe but not S-coverable net in Fig. 4.

There is a close connection between structured concurrency nets and S-coverable nets. Two places p_1, p_2 in a structured concurrency net are part of the same S-component if and only if in the corresponding statechart net p_1 and p_2 are not orthogonal, i.e., their *lca* is an OR node.

Theorem 2. *Given a wellformed statechart net* $SN = (P, T, F, AND, OR, children, C_0)$. *A tuple* (P', T', F', M_0) *is an S-component of SN if*

- $P' \subseteq P$ *is a maximal set of places such that for every* $p_1, p_2 \in P'$, $p_1 \neq p_2$ *implies* $lca(\{p_1, p_2\}) \in OR$.
- $T' = \{t \in T \mid \exists\, p_1 \in P' : (p_1, t) \in F \wedge \exists\, p_2 \in P' : (t, p_2) \in F\}$
- $F' = F \cap ((P' \times T') \cup (T' \times P'))$.
- *For every* $p \in P'$, $M_0(p) = 1 \Leftrightarrow p \in C_0$

Proof. We show (i) (P', T', F', M_0) is a state machine and (ii) for every $p' \in P'$ and $t \in T$, if $(p', t) \in F$ then $(p', t) \in F'$, and if $(t, p') \in F$ then $(t, p') \in F'$.

(i) Take a transition $t \in T'$. We show that $\bullet t = 1$; the case $t\bullet = 1$ can be proven with similar reasoning. Clearly, $\bullet t > 0$, otherwise t would not be in T'. For a contradiction, suppose there is a transition $t' \in T'$ such that $|\bullet t \cap P'| > 1$, and let p_1, p_2 be two different places in $\bullet t \cap P'$. Since t is consistent, $lca(\{p_1, p_2\})$ has type AND. But then, by definition of P', places p_1 and p_2 cannot be both in P'.

(ii) We show that for every $p' \in P'$ and $t \in T$, $(p', t) \in F \Rightarrow (p', t) \in F'$. The other case is by similar reasoning.
Assume $(p', t) \in F$. The postset of t is not empty. There is a place p'' in the postset of t such that $lca(\{p', p''\}) \in OR$. (If $scope(t) \in OR$, this is trivially true. If $scope(t) \in AND$, by target completeness such a p'' always exists.) By definition of P'' and T', then $p'' \in P'$ and $t \in T'$ and thus $(p', t) \in F'$ by definition of F'.

For structured concurrency nets, each place p is covered by an S-component. Thus, each structured concurrency net is S-coverable. The reverse implication, however, does not hold. For example, the net in Fig. 2 is S-coverable but is not a structured concurrency net.

4.3 TP and PT handles

A Petri net has a TP handle if there is a transition t and place p and there are two directed paths from t to p that are disjoint. A Petri net has a PT handle if there is a place p and transition t and there are two directed paths from p to t that are disjoint. The notions of TP and PT handle were introduced by Esparza and Silva [10].

Intuitively, a TP handle between t and p causes p to be unsafe whereas a PT handle between p and t causes a deadlock at t. This intuition suggests that (i) a Petri net that does not have PT and TP handles is a structured concurrency net, and that (ii) a structured concurrency net does not have PT or TP handles. However, both suggestions are false. A counterexample for claim (i) is the Petri net in Fig. 4. Though that net does not have a TP and PT handle, no equivalent statechart net exists. A counter example for claim (ii) is shown in Fig. 5 (adapted from Fig. 6 in Esparza and Silva [10]). The net contains a TP handle between `t1` and `p1`, and a PT handle between `p2` and `t2`. Yet an equivalent statechart net exists, shown in the same figure.

Nevertheless, there are indeed Petri nets with PT and TP handles that are not structured concurrency nets. For example, the net in Fig. 6 is safe and has a PT (`p5-t6`) and a TP handle (`t1-p5`). An equivalent statechart net does not exist.

4.4 Free choice nets

In a free choice net [5], every pair of t, t' of transitions has either a disjoint or equal preset: $\bullet t \cap \bullet t' = \emptyset \vee \bullet t = \bullet t'$. For example, the net in Fig. 7 is not free choice since for example `t4` and `t7` share input place `p3`, but do not have an equal preset.

There is no relation between free choice nets and structured concurrency nets. The net in Fig. 7 is not free choice, yet an equivalent statechart net exists. The net in Fig. 2 is free choice, yet no equivalent statechart net exists.

4.5 Dead Petri nets

In a Petri net, a transition t is defined to be *dead* if there is no reachable marking that enables t. A Petri net that has dead transitions is called dead.

In wellformed statechart nets, each transition has by definition a consistent set of sources. Each consistent set of nodes can be turned into a configuration. This might suggest that in a structured concurrency net, each transition cannot be dead.

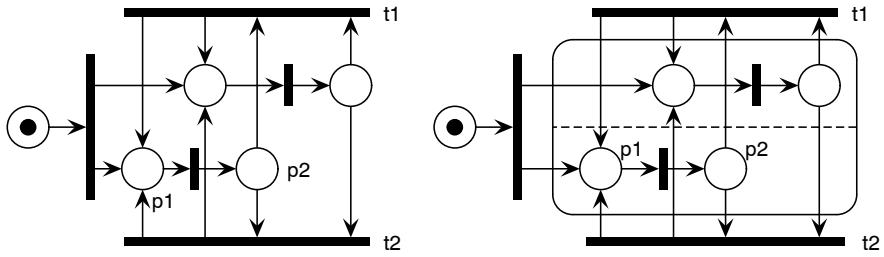

Fig. 5. Petri net with PT and TP handle and equivalent statechart net

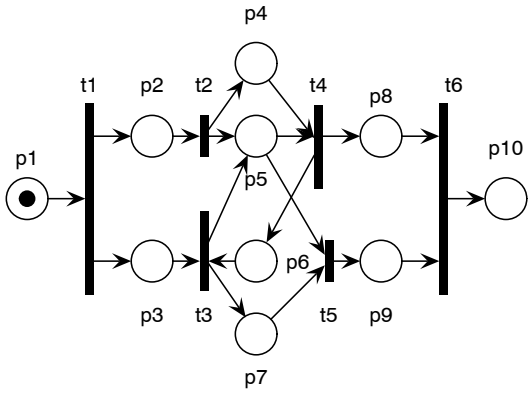

Fig. 6. Safe Petri net with PT and TP handles that is not a structured concurrency net

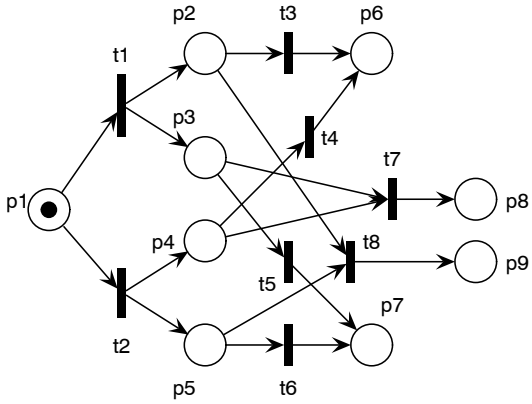

Fig. 7. Free choice structured concurrency net that has dead transitions

However, a structured concurrency net may have dead transitions. Consider for example the net in Fig. 7. Clearly, transitions t7 and t8 are dead, because t1 and t2 are exclusive. Yet an equivalent statechart net exists, by letting both p2,p4,p6 and p3,p5,p7 have the same OR parents.

4.6 Statechart variants

As explained in the introduction of Section 3, statechart nets can be seen as a simplified, less expressive version of statecharts. We now study some proposed extensions of statecharts that extend or relax the hierarchical AND/OR structure, thus allowing more forms of concurrency. We show that the resulting statechart net variants do not coincide with a subclass of Petri nets, i.e., there are statechart nets for which no equivalent Petri net exists.

Statecharts with overlapping. Harel and Kahana [12] propose an extension to statecharts in which a node can overlap another node. Technically, the statechart constraint that each node should have one parent is relaxed. The node structure is no longer required to be a tree. However, the structure should (still) be acyclic.

Obviously, statechart nets with overlapping are more expressive than ordinary statechart nets. Cross-synchronisations between parallel branches can be modelling using AND nodes that overlap each other. Figure 8 shows a statechart net with overlapping equivalent to the net in Figure 2, while Fig. 9 shows the node hierarchy of this statechart net. And even the net in Fig. 6 can be modelled in a statechart net with overlapping.

Nevertheless, safe nets and statechart nets with overlapping do have different expressive power. There exist safe nets which have no equivalent statechart net with overlapping, for example the net in Fig. 4. Conversely, not every statechart net with overlapping has an equivalent Petri net. For example, the statechart in Fig. 10 has no equivalent net. Note that the hierarchy in this case is hard to visualise; the node structure is shown on the right. In the Petri net obtained by dropping all composite nodes, after firing t1, t2 and t3, a marking is reached in which place p5 has two tokens. However, in statechart nets with overlapping, each node can be active at most once, so there cannot be a configuration in which p5 is active twice.

Statecharts with synchronisation nodes. In UML 1.x [23], statecharts have been extended with synchronisation nodes (called "synch states" in UML). A synchronisation node n synchronises concurrent OR children of some AND node a. Node n is entered by some transition inside an OR child of a, and left by a transition inside another OR child of a.

With synchronisation nodes, the net in Fig. 2 can be modelled straightforwardly; see Fig. 11. The synchronisation node is represented by circle on the dotted line separating the OR nodes that it synchronises. The '1' indicates that the synch node is activated at most once.

Though this example shows that cross-synchronisations can be modelled in statecharts with synch nodes, there do exist safe nets that have no equivalent

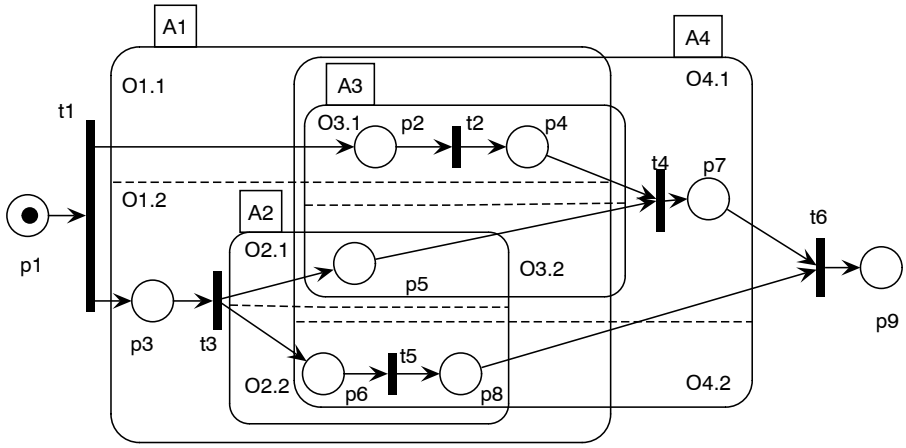

Fig. 8. Statechart net with overlapping nodes equivalent to net in Fig. 2

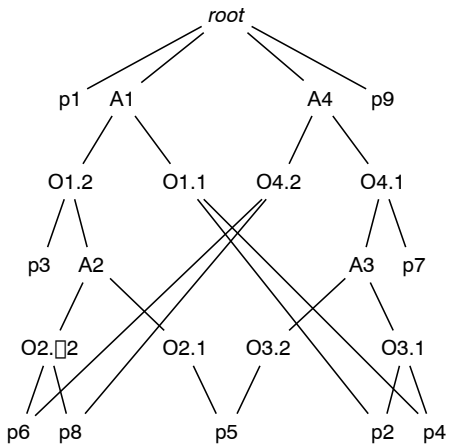

Fig. 9. Node hierarchy of the statechart of Fig. 8

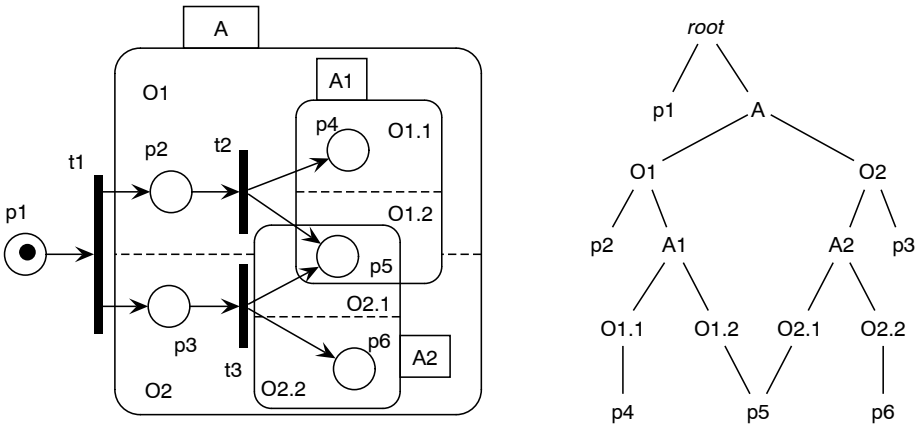

Fig. 10. Statechart net with overlapping and correponding node hierarchy for which no equivalent Petri net exists

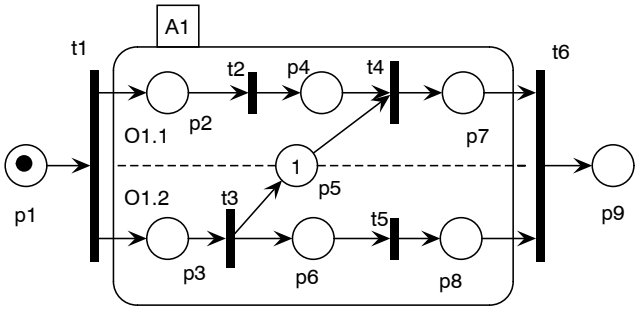

Fig. 11. Statechart with synchronisation nodes equivalent to net in Fig. 2

statechart with synch nodes, for example the nets in Fig. 6 and 4. Moreover, not every statechart net with synch nodes has an equivalent safe net. For instance, if the statechart net of Fig. 11 were extended with a transition t leaving p6 and entering p3, the synch node would get activated multiple times. Such multiple activations are forgotten according to the UML 1.x semantics [23]. The underlying Petri net, however, would grow unbounded in the synch node.

5 Deadlock and livelock

In the previous section, we showed that structured concurrency nets can contain dead transitions. Moreover, they also can contain deadlocks and livelocks. In this section, we define constraints on the syntax of statechart nets that guarantee the absence of deadlocks and livelocks in statechart nets. Such constraints are hard, if not impossible, to specify on the syntax of safe Petri nets.

The definitions we use for deadlock and livelock are a bit nonstandard, since we consider statecharts nets that can terminate properly, i.e. nets that can reach a final state, whereas usually Petri nets are considered as cyclic systems that have no final state. (If they do, it is considered a deadlock.) The definitions we take are useful to model the behaviour of a life cycle, for example of an object, component, business process, or workflow [1, 3].

5.1 Definitions

Deadlock, livelock, and proper termination. Call a place *final* if it is not input to a transition:

$$\mathit{final}(p) \stackrel{\text{df}}{\Leftrightarrow} \forall\, t \in T : p \notin {\bullet t}.$$

A configuration is *final* if all places it contains are final. Final configuration are the desirable end states of a net. Note that there can be arbitrarily many final configurations.

A configuration C *deadlocks* iff no transition is enabled in C and C is not final. Then there is some transition t some of whose input places are filled. A statechart net *deadlocks* iff one of its reachable configurations deadlocks. For example, the statechart net in Fig. 12 deadlocks in configuration $\{\mathsf{p1}, \mathsf{p3}\}$.

A configuration C *livelocks* iff no final configuration can be reached from C. A statechart net *livelocks* iff one of its reachable configurations livelocks.

A statechart net *terminates properly* iff from all configurations, a final configuration can be reached. Clearly, a statechart net terminates properly iff it neither deadlocks nor livelocks.

OR graphs. To define syntactic constraints that guarantee absence of deadlock and livelock, we need some additional concepts that can be derived from the statechart net syntax. For each OR node o, we define a directed graph $G_o = (\mathit{Nodes}, \mathit{Edges})$ where

- $\mathit{Nodes} \stackrel{\text{df}}{=} \mathit{children}(o)$
- $\mathit{Edges} \stackrel{\text{df}}{=} \{(o_1, o_2) \mid o_1, o_2 \in \mathit{Nodes} \wedge \exists\, t \in T : \mathit{lca}({\bullet t}) = o_1 \wedge \mathit{lca}(t{\bullet}) = o_2\}$

We call G_o the *OR graph* of o.

Next, we introduce the concept of drains for OR node. For an OR node o, a drain is a set of child nodes of o that are not eventually left by edges in $\mathit{Edges}(G_o)$. Formally, a set of state nodes X is a *drain* of o iff

- $X \subseteq \mathit{children}(o)$.
- X is a strongly connected component (SCC) of G_o, i.e, any node in X is reachable from any other node in X and X is maximal.
- For every $x \in X$, if there is an edge $(x, x') \in \mathit{Edges}(G_o)$, then $x' \in X$.

The last constraint ensures that X is not eventually left while the net stays in o. In the statechart net shown in Fig. 12, OR node $\mathsf{o1}$ has two drains: $\{\mathsf{p1}\}$ and $\{\mathsf{p2}\}$.

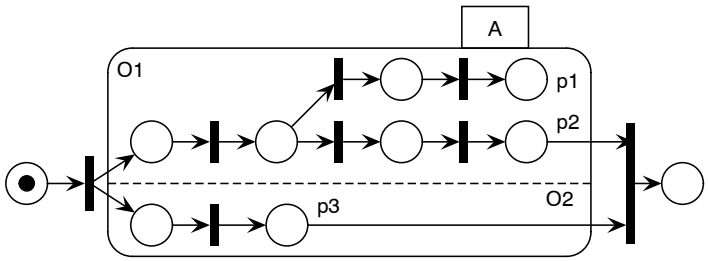

Fig. 12. A deadlocking statechart net

Consider an AND node a. A set $Ds = \{D_1, D_2, .., D_n\}$ is defined to be an *arbitrary drain set* of a iff

- For each OR child o of a, there is an element $D \in Ds$ such that D is a drain of o.
- For each $D \in Ds$, there is an OR child o of a such that D is a drain of o.
- $n = |children(a)|$.

These three constraints ensure that there is a bijection between the drains in Ds and the OR children of a. For the statechart net in Fig. 12, AND node A has two arbitrary drain sets: $Ds_1 = \{\{\mathsf{p1}\}, \{\mathsf{p3}\}\}$ and $Ds_2 = \{\{\mathsf{p2}\}, \{\mathsf{p3}\}\}$.

Each arbitary drain set of an AND node should be left by some transition. AND node a is defined to be *exit complete* if and only if for each arbitrary drain set Ds of a, there is a transition $t \in T$ such that for every $D \in DSs$ there is an input place of t that is descendant of a node in D, i.e. t leaves D:

$$exit\_complete(a) \Leftrightarrow \text{for an arbitrary drain set } Ds \text{ there is a } t \in T \text{ such that for every } D \in Ds \; (\bigcup_{d \in D} children^*(d)) \cap {\bullet t} \neq \emptyset.$$

AND node A in Fig. 12 is not exit complete, since there is no transition leaving Ds_1. We call a statechart net exit complete if each of its AND nodes is exit complete.

In principle, each AND node should be exit complete. However, the net can end in certain AND nodes, for example in AND node A in the statechart net in Fig. 13. Such AND nodes are not required to be exit complete. To deal with them, we slightly modify a statechart net by finalising it. We introduce a special final place, that signifies that the statechart net has completed. This special place is filled if in the original statechart net a final configuration is reached. In principle, configurations are computed by building a reachability graph. However, the syntactic constraints on statechart nets allow for a much more efficient solution. We can compute maximal consistent sets of final places that are candidate configurations (candidate only because they might not be reachable). Next,

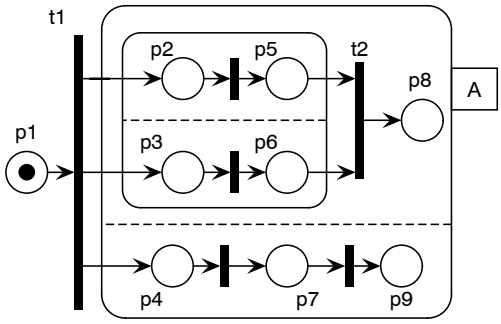

Fig. 13. Statechart net that ends in AND node

we define transitions whose preset is a candidate final configuration and whose postset only contains the final place.

Formally, we define this as follows. Given a statechart net $SN = (P, T, F, AND, OR, children, p_i)$, its finalisation $final(SN)$ is a statechart net SN' where

- $P' = P \cup \{end\}$
- $T' = T \cup \{t_X \mid X \text{ is a maximal, consistent set of final places in } SN\}$
- $F' = F \cup \{(t_X, final) \mid t_X \in T' \setminus T\}$
- $AND' = AND$
- $OR' = OR$
- $children' = children \cup \{(root, end)\}$.

Next, we show that finalising a statechart net does not affect its deadlock and livelock properties.

Theorem 3. *(i) A wellformed statechart net SN deadlocks iff its finalisation final(SN) deadlocks.*

(ii) A wellformed statechart net SN livelocks iff its finalisation final(SN) livelocks.

Proof. We prove (i); the other case is by similar reasoning.

$\Rightarrow$. The finalisation $final(SN)$ extends SN by adding a new place and new transitions. Hence, each deadlock in SN is also a deadlock in $final(SN)$.

$\Leftarrow$. By definition of *final*, net $final(SN)$ can only have deadlocks not present in SN in final configurations of SN or in configuration $\{end, root\}$. But by construction, the novel place *end* is not in the preset of any transition, so it cannot be part of a deadlock. Moreover, each final configuration C of SN is a preset of a transition t, so each place $p \in C$ can always be left. Hence, each deadlock of $final(SN)$ is also a deadlock of SN.

5.2 Absence of deadlocks

Using the notions introduced in Section 5.1, we can finally define a sufficient constraint that guarantees absence of deadlock.

Theorem 4. *Given a wellformed statechart net SN. If $final(SN)$ is exit complete, then SN is deadlock free.*

Proof. We show that $final(SN)$ is deadlock free. From Theorem 3 then follows that SN is deadlock free.

Take a non-final configuration C that deadlocks, so there is no enabled transition in C. Then there is a transition t such that $\bullet t \cap C \neq \emptyset$. Let a be $lca(\bullet t)$. Then $Ds = \{\{y\} \mid y \in children(x) \wedge x \in children(a) \wedge \{x, y\} \subset C\}$ is a drain set of a. But there is no transition leaving Ds, so a is not exit complete.

The reverse implication is not true, because an AND node violating exit completeness might not be reachable because of dead transitions.

5.3 Proper Termination

While exit completeness rules out deadlock, it does not ensure absence of livelocks. For example, the net in Fig. 14 is exit complete. Yet it can livelock, namely if t1 has been fired. Then p4 is never filled, and t3 can never fire, and so AND node A is never left.

To rule out such statechart nets, we require that each OR node has a single entry point. A child c of an OR node o is an *entry point* of o if there is a transition t such that

- a place in $t\bullet$ is descendant of c, and
- $scope(t)$ is ancestor of o, but not equal to o.

In Fig. 14 OR nodes O1 and O2 have each two entry points: p3, p5, and p6, p7 respectively. Note that p4 is not an entry point.

Under the assumption that each OR node has a single entry point, we can indeed prove that a wellformed exit-complete statechart net terminates properly. First we prove a helpful theorem that states that for such a statechart net each AND node can be left. In the proof, we use the concept of depth of a node. Places have depth zero, while the depth of a composite node is the maximum depth of its children plus one. Formally,

$$depth(p) = 0, \quad \text{where } p \in P$$
$$depth(n) = 1 + max(\{depth(c) \mid c \in children(n)\}), \quad \text{if } n \in AND \cup OR$$

The nesting depth of a drain is the maximum depth of drain members. The nesting depth of a drain set is the maximum depth of the individual drains.

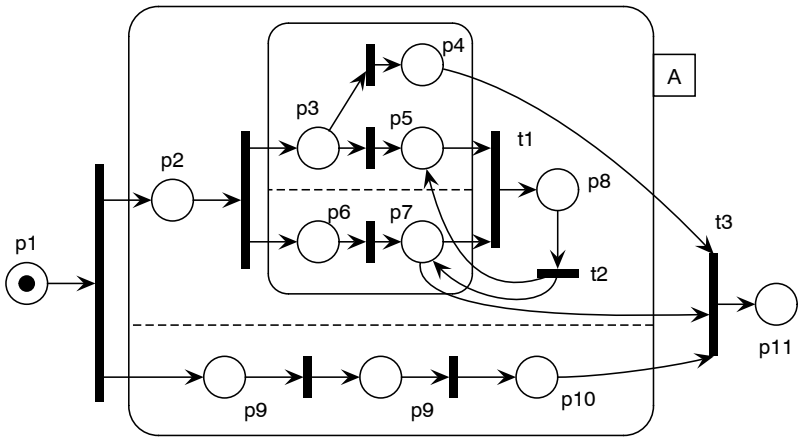

Fig. 14. Statechart net that can livelock

Theorem 5. *Given a wellformed statechart net SN such that final(SN) is exit complete and each OR node has a single entry point. Let a be an arbitrary AND node of SN.*

For every non-final configuration C containing a, there is a transition t leaving a and a configuration C′ reachable from C, such that $\bullet t \cap children^*(a) \subset C'$.

Proof. We prove the claim by induction on the depth of the drains of a. Denote by P_a the set of second level children of a: $\{n \in C \mid parent(parent(n)) = a\}$.

Basic Case.

For the basic case, P_a only contains places: $P_a \subseteq P$. Take some place $p \in P_a$. If p is not part of a drain of $parent(p)$, then by definition of drain, from p a sequence of transitions (all with singleton presets and singleton postsets) can be taken such that a place p' is reached that is part of a drain.

So assume every place $p \in P_a$ is part of a drain of $parent(p)$. By definition of exit completeness, there is a t such that for every $p \in P_a$ there is a $p' \in \bullet t$ such that p and p' are in the same drain. By definition of drain, p and p' are connected by a sequence of transitions all having singleton presets and postsets. Hence, p can reach p' and t can become partly enabled.

Induction case.

By the definition of OR nodes, for each OR child c of a, exactly one node in P_a is a node of c's OR graph. There are two cases:

- Some node $p \in P_a$ is not part of a drain of $parent(p)$. By definition of drain, there exists a place $p' \in P$ that is part of a drain, and p' is reachable from p in the OR graph. Thus, by definition of OR graph, there is a sequence of transitions $t_1, t_2, .., t_n$ where $lca(\bullet t_1) = p$ and $lca(t_n \bullet) = p'$. For each transition t_i, where $0 < i \leq n$, we have $lca^+(\bullet t_i) = lca^+(t_i \bullet) = parent(p)$.

Denote by C_i the configuration entered by taking t_{i-}. We now show that t_i can be taken. Clearly, C_i contains $lca(\bullet t_i)$. There are two cases:

(i) $lca(\bullet t_i) \in P$. Then $\bullet t_i$ is a singleton, say $\{p_i\}$. Since p_i is in C_i, transition t_i can be taken.

(ii) $lca(\bullet t_i) \in AND$. Let $a = lca(\bullet t_i)$. By the induction hypothesis, there is a configuration C_a, reachable from C_i and a transition t_a, such that $\bullet t_a \cap children^*(a) \subseteq C_a$. Since each OR child of a has a unique entry point, we have that *each* transition leaving a, including t_i, can become partly enabled by some configuration C_a reachable from C_i.
It remains to be shown that t_i is completely enabled in C_a. Because $lca^+(\bullet t_i) = parent(p)$, we have $\bullet t_i \subseteq children^+(a)$, and thus $\bullet t_i \cap children^+(a) = \bullet t_i$. Thus $\bullet t_i \subset C_a$ and hence t_i is enabled in C_a.

– If every node $p \in P_a$ is part of a drain D_p of $parent(p)$, then by definition of exit completeness, there is a transition t such that for some $p' \in D_p$, $\bullet t \cap D_p = \{p'\}$. By definition of drain, in the OR graph of $parent(p)$ there is a path $p, p_1, p_2, .., p_n, p'$ leading from p to p'. By similar reasoning as in (ii), we can prove that p' is reachable.

Using Theorem 5 we now show that exit completeness and single entry points for OR nodes are indeed sufficient constraints for proper termination.

Theorem 6. *A wellformed statechart net SN terminates properly if final(SN) is exit complete and each OR node has a single entry point.*

Proof. Denote by G_{root} the OR graph of the root node *root* of *final*(SN) and let i be a child node of *root* such that $i \in C_0(SN)$. By definition of *final*(SN), there is a directed path p from i to *end* in G_{root}. We show that every node on this path, once entered, can be left.

Take a pair of neighbouring nodes n, n' on this path, where $n \neq end$, and let t_n be the transition for which $lca(\bullet t_n) = n$ and $lca(t_n \bullet) = n'$. Denote by C_n the configuration entered by taking a transition t_m where $lca(t_m \bullet) = n$.

– If n in P, then by consistency of $\bullet t_n$, we have $\bullet t_n = \{n\}$ and so t_n is enabled in C_n.
– If n is an AND node, then by Theorem 5 and the assumption that each OR child of n has a single entry point, there is a configuration C' that is reachable from C_n such that $\bullet t_n \cap children^*(n) \subseteq C'$.

Next, since $lca^+(\bullet t_n) = root$, we have $\bullet t_n \subseteq children^+(root)$, so $\bullet t_n \subseteq C'$ and thus C' enables t_n.

The reverse implication is not true in general, but holds if for every place $p \in P$ there is a path from some initial place $p_0 \in C_0$ to p. Under that condition, each potential configuration is a reachable configuration. This gives an efficient syntactic decision procedure for reachability properties.

6 Discussion and Conclusion

In this paper, we have studied nets with structured concurrency, i.e. nets whose concurrency can be cast into an AND/OR state hierarchy. We introduced a novel Petri net variant, statechart nets, which extend Petri nets with a statechart-like AND/OR hierarchy on places. We have shown that structured concurrency nets are a strict subclass of safe and S-coverable nets, but do not coincide with existing Petri net subclasses. The main distinguishing feature of structured concurrency nets is that cross-synchronisation between parallel branches is not possible. This result gives more insight into the expressive power of AND/OR state hierarchies.

Next, we defined syntactic constraints on statechart nets that guarantee absence of deadlock and livelock. For standard Petri nets, such syntactic constraints are hard if not impossible to specify. This shows one advantage of using statechart nets over Petri nets. Another advantage of statechart nets is that potential configurations can be syntactically characterised. For example, this feature allowed the simple transformation of a statechart net with multiple end places into one with only one end place in Section 5, without introducing deadlocks or livelocks. Van der Aalst and Hofstede [2] define a similar transformation in the context of Petri nets, but their transformation is a lot more complex than ours, requiring the use of weighted transitions and a shadow place that keeps track of the number of parallel streams.

Another contribution of this paper is that it sheds some light on the differences between Petri nets and statecharts by studying one specific statechart feature, the AND/OR hierarchy, in a Petri net setting. Other works incorporating statechart features in Petri nets are by Holvoet and Baeten [15] and Kishinevsky et al. [18]. But in the net variants proposed there, places can be decomposed into a subnet containing other places, which amounts to an OR decomposition. In particular, concurrency is still expressed in the Petri net way, so it might be not structured and places might be unsafe or even unbounded. Another difference is that these variants allow a transition to have a composite place as input, which we do not consider here.

There are several directions for further work. One is to extend statechart nets with transitions that leave composite (AND/OR) nodes, thus introducing the concept of priority, which is typical of statecharts [11, 13, 23]. Such an extension might imply that the syntactic constraints guaranteeing absence of deadlock and livelock have to be adjusted. Another direction is to apply the results of this paper in a practical setting. We are currently using the syntactic constraints listed in Section 5 to implement an efficient check for absence of deadlocks and livelocks in workflow models expressed in Petri nets [17].

References

1. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems.* MIT press, 2002.
2. W.M.P. van der Aalst and A.H.M. ter Hofstede. Verification of Workflow Task Structures: A Petri-net-based Approach. *Information Systems*, 25(1):43–69, 2000.

3. G. Agha, F. Decindio, and G. Rozenberg, editors. *Concurrent Object-Oriented Programming and Petri Nets*. Lecture Notes in Computer Science 2001. Springer, 2001.
4. A. Cheng, J. Esparza, and J. Palsberg. Complexity results for 1-safe nets. *Theoretical Computer Science*, 147(1–2):117–136, 1995.
5. J. Desel and J. Esparza. *Free choice Petri nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1995.
6. J. Desel and G. Juhás. What is a Petri net? Informal answers for the informed reader. In H. Ehrig, G. Juhás, J. Padberg, and G. Rozenberg, editors, *Unifying Petri Nets*, Lecture Notes in Computer Science 2128, pages 1–27. Springer, 2001.
7. R. Eshuis. Statecharting Petri nets. Beta Working Paper Series, WP 153, Eindhoven University of Technology, 2005.
8. 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.
9. J. Esparza and M. Nielsen. Decidability issues for petri nets: A survey. *Journal of Information Processing and Cybernetics*, 30:143–160, 1994.
10. J. Esparza and M. Silva. Circuits, handles, bridges and nets. In G. Rozenberg, editor, *Advances in Petri Nets*, volume 483 of *Lecture Notes in Computer Science*, pages 210–242. Springer, Berlin, 1991.
11. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
12. D. Harel and C.-A. Kahana. On statecharts with overlapping. *ACM Transactions on Software Engineering and Methodology*, 1(4):399–421, 1992.
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. T. Holvoet and P. Verbaeten. Petri charts: an alternative technique for hierarchical net construction. In *Proceedings of the 1995 IEEE Conf. on Systems, Man and Cybernetics: Intelligent Systems*. IEEE Press, 1995.
16. N.D. Jones, L.H. Landweber, and Y.E. Lien. Complexity of some problems in Petri nets. *Theoretical Computer Science*, 4(3):277–299, 1977.
17. B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows*. PhD thesis, Queensland University of Technology, 2002.
18. 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.
19. T. Murata. Petri nets: Properties, analysis, and applications. *Proc. of the IEEE*, 77(4):541–580, 1989.
20. 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.
21. W. Reisig. *Petri Nets: An Introduction*. Number 4 in EATCS Monographs on Theoretical Computer Science. Springer, 1985.

22. G. Rozenberg and J. Engelfriet. Elementary net systems. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri nets I: Basic Models*, Lecture Notes in Computer Science 1491, pages 12–121. Springer, 1998.
23. 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`.
24. A. Wąsowski and P. Sestoft. On the formal semantics of visualSTATE statecharts. Technical Report TR-2002-19, IT University of Copenhagen, 2002.