

Chapter 2

Translating MA Processes into Safe Petri Nets — The Idea

The notion already developed in the original "Mobile Ambients"-paper [CG98, p. 144] that an MA process is a **tree** gives rise to the idea to use this tree as the data structure encoded into the Petri nets. We depict first MA trees in Section 2.1. Restrictions and ambients are inner nodes and all terms starting with a capability, replication, 0, or a call are leaves.

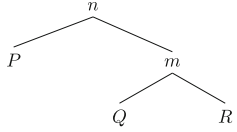
When we encode the capability actions into transitions in Section 2.2 the **locality of changes** in the MA calculus greatly helps. This property is only too natural when we consider MA's origin in fire wall modeling where each level must be step-wisely taken. It naturally fits into the Petri net world since only few and easily identifiable places are affected by each transition.

Based on the requirements of such a translation we propose an encoding into Petri net places which are named after all possible parent-child relations. We discuss problematic ambiguities which arise due to the management by name. They are resolved in Section 2.3 where we introduce two distinct name sets $\mathcal{A}$, which keeps the tree unambiguous, and $\mathcal{L}$, which allows us to drop the restrictions. We derive their management via additional Petri net transitions.

2.1 Translating MA Terms into Safe Petri Net Markings

Every MA process term can be represented as a tree where the inner node structure depicts the ambient hierarchy with the restrictions while the leaves are prefix guarded, that is each leaf starts with a capability π , is a call or 0, or a replication. The parallel compositions separate siblings.

Example 2.1 (Simple ambient hierarchies). The process $n[P|m[Q|R]]$ corresponds to the tree:

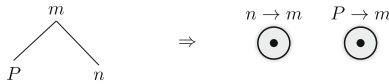


We do not wish to order a node's children in the tree but define it to represent all process terms congruent by the rule $P|Q \equiv Q|P$. Thus, the example tree above also stands for the process terms $n[P|m[R|Q]]$, $n[m[Q|R]|P]$ and $n[m[R|Q]|P]$. We will later abstract even more congruence rules away so that one marking represents an even larger subset of a process term's congruence class.

To maintain a tree rather than a forest if parallel compositions occur at the process's top level it is necessary to add an artificial root *root* adopting all so far orphaned process terms. We will always have this root as to prevent any process term from becoming an orphan but only depict it in figures if necessary to keep them representing a tree.

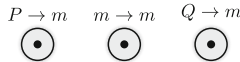
We use the Safe Petri net's places to represent all possible parent-child relations with the tokens representing the actual ones. E.g., a token on the place labeled $n \rightarrow m$ shows that an ambient n is a direct child of an ambient m whereas another token on $P \rightarrow m$ represents a similar relation between a (leaf) process term P and the ambient.

Example 2.2 (Translating a tree). The process term $m[P|n[\dots]]$ represents the following tree which is then translated into a net marking:

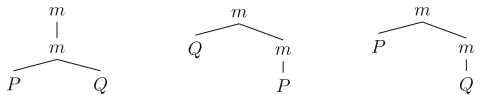


Just like the tree it depicts the marking doesn't define an ordering on a node's children. Thus, each marking represents all processes modulo reordering $P|Q \equiv Q|P$, too. So far, the parent-child relation is maintained only by name so that problems arise due to ambiguities if there are two ambients with the same name within a process.

Example 2.3 (Ambiguities in the net). The process term $m[P|m[Q]]$ corresponds to



This allows for several interpretations:



The last tree is the correct one but the representation allows for all of them.

The ambiguity becomes fatal as soon as capabilities are executed on unintended subtrees leading then to actually unreachable configurations. We will hence later use unique names $\mathcal{A} = \{a_i \mid i = 1, \dots, n\}$ for all ambients. These newly assigned names are not in any relation to the original ones. Thus, we need to make sure that each capability knows "its" ambients.

Even with unique ambient names, yet another problem remains: Identical leaf process terms, so called twins, under one ambient hurt the one-safeness.

Example 2.4 (Multiple tokens in the presence of twins). We depict the situation for a double occurrence of the leaf process term $P = \pi n.Q$ under a :

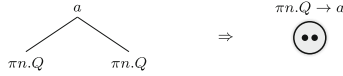
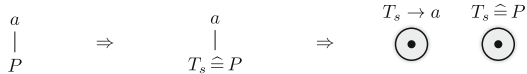


Fig. 2.1 The translation so far allows for multiple tokens.

We introduce unique identifiers $\mathbb{T} = \{T_s \mid i = 1, \dots, n\}$, so called twigs, which we put instead of the leaf below its parent ambient. The relation to the actual leaf term P is maintained via places $T_s \hat{=} P$:



Example 2.5 (Dealing with twins without and with twigs). With twigs, the situation from Figure 2.4 translates to

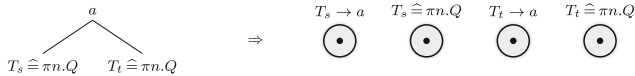


Fig. 2.2 Representing twigs in the net.

Although the introduction of twigs is a rather simple step, we will not incorporate them in our introductory constructions that simply transport the idea. They create a blow-up on every transition which may hide the central idea behind formalisms. The final construction presented in Section 3.2.2 will of course provide all features.

To separate the different groups of places with different functions we will use a colour encoding for the rest of this work. The colours are added step-by-step and later summarised in Figure 3.1.

2.2 Translating Capability Actions into Petri Net Transitions

The MA calculus is a tree rewriting system since a change in the process state, i.e. the performance of a step according to the transition relation $\rightsquigarrow$ or the congruence $\equiv$, is a transition from one tree into another one.

Example 2.6 (Tree transition). Let $k \notin fn(P_2)$. We can then apply the scope extrusion:



The MA calculus solely provides actions with a local impact. All three capabilities only affect adjacent levels in the tree but behave quite differently on any other account. Our first rough translation of these actions will visualise which further information we should encode in our net in order to exclude behaviour impossible in MA. We will refine the construction by including twigs and the sophisticated name management. Restrictions are removed from the tree by the introduction of unique restricted link names $\mathcal{R} = \{r_i \mid i = 1, \dots, n\}$. Even though they are not yet introduced the reader may already assume a restriction-free tree. The call and some other operations are straight forward so that we will only depict them in Section 3.2.2 when we present the final translation for each MA construct.

We will now shortly summarise the capabilities' tree transitions. For the rest of this section we fix some naming conventions: Each process incarnation is contained in an ambient¹ which we call o . It is important to keep in mind that the process variables P , Q , and R used in MA's transition relation represent arbitrary process terms including those with an ambient or a parallel composition at the top level. However, in the case of *open* m it will be important to highlight that such a process can be a parallel composition. We will do so by using Q_1 to Q_n instead of Q .

2.2.1 The Capability Action in m

The action $n[in\ m.P|Q]|m[R] \rightsquigarrow m[n[P|Q]|R]$ corresponds to:

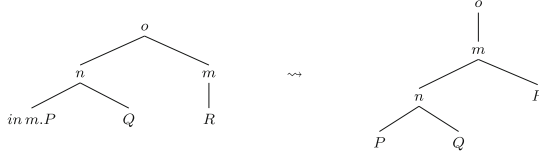
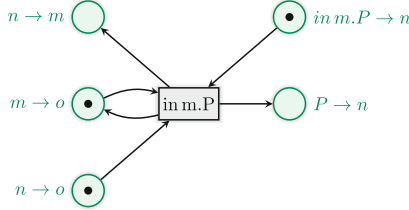


Fig. 2.3 $o[n[in\ m.P|Q]|m[R]] \rightsquigarrow o[m[n[P|Q]|R]]$

We make the parent n of $in\ m.P$ a child of its former sibling m . We neither depict Q nor R nor any further children of o in our net fragment since they do not interfere with the transition. In order to avoid n jumping through the tree we need to verify that they are siblings: We assumed that o is n 's parent and thus need to check that it is also m 's parent via a test on $m \rightarrow o$.



We depend on $in\ m.P$, o , and n for the transition. Thus, there is one such transition per $(o, n, in\ m.P)$ -combination.

2.2.2 The Capability Action out m

The action $m[n[out\ m.P|Q]|R] \rightsquigarrow n[P|Q]|m[R]$ corresponds to:

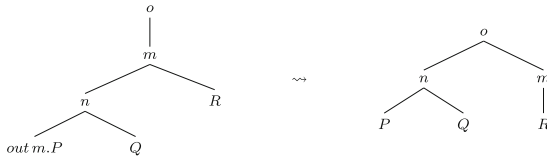
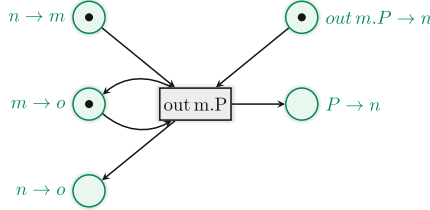


Fig. 2.4 $o[m[n[out\ m.P|Q]|R]] \rightsquigarrow o[n[P|Q]|m[R]]$

¹ Remember that at least *root* is such an ambient.

The operation $out\ m$ is inverse to $in\ m$. The ambient n , formerly the parent of $out\ m.P$ and the child of ambient m , becomes m 's sibling during the transition. Therefore, the transition has to know m 's parent (which we assume to be o) to introduce n as its new child. This requires the test on $m \rightarrow o$ here.



In comparison to an in transition, the $n \rightarrow m$ place is a token source while $n \rightarrow o$ now receives a token. Everything else is equal for both transitions (of course, now we consume the $out\ m$ prefix) and thus we again need one transition for each $(o, n, out\ m.P)$ -combination.

2.2.3 The Capability Action $open\ m$

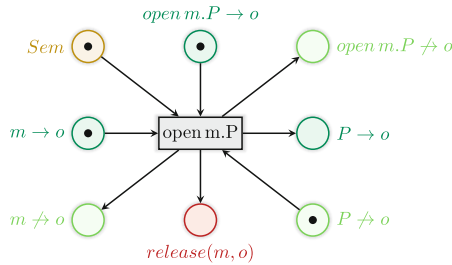
The action $open\ m.P \mid m[Q] \rightsquigarrow P \mid Q$ corresponds to:



Fig. 2.5 $o[open\ m.P \mid m[Q]] \rightsquigarrow o[P \mid Q]$

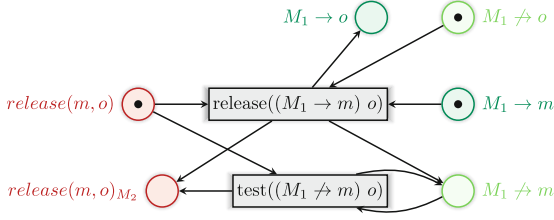
The ambient m is removed by its sibling $open\ m.P$. This requires *all* processes which were so far m 's children to introduce themselves to m 's and $open\ m.P$'s common parent o . Obviously, a new test on o is required but it can be realised with the change from $open\ m.P \rightarrow o$ to $P \rightarrow o$ which is anyway necessary.

Unfortunately, the Safe Petri net's structure so far is not able to decide *when* all subprocesses of m subscribed to o . We install complement places to overcome this short-coming. Now we may either inspect all place and complement place pairs in a predefined order or we can visit them concurrently. Since the latter approach requires an additional huge transition to unite the intermediate visits into an "all-ambients-considered" place – and since additional arguments will be necessary when arguing for the transitions' correctness – we will implement the first approach. Depending on the Petri net model checker at hand one may of course choose the concurrent approach instead.



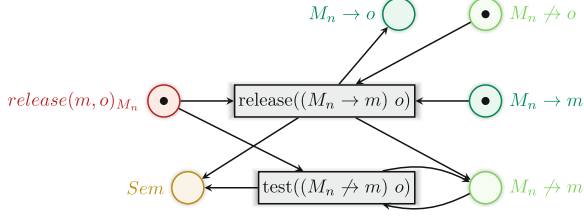
We use the semaphore to block other transitions which may influence the actual tree. This will prevent undesired races before we are able to restore a correct tree. After the actual $open\ m.P$ transition the tree is already degenerated into a forest since the connection $m \rightarrow o$ is erased. At first the second tree with root m possesses the children Q_1 to Q_n but during the chain's execution these children vanish. It

doesn't make a difference whether an ambient or a leaf process is m 's child so that we unite them in a set $M = \{P \mid P \text{ leaf term}\} \cup \{n \mid n \text{ name}\} \setminus \{m, o\}$ which we stepwisely process.



The chain is hard-coded via progress logging places $release(m, o)_{M_s}$ where $release(m, o)_{M_s}$ is consumed for the $(s + 1)$ -th transition and $release(m, o)_{M_{s+1}}$ is fed by it. We can unify the depicted first steps with all intermediate steps by a slight modification: Rather than $release(m, o)$ the *open* transition should fill $release(m, o)_{M_1}$.

The process chain provides the same two transition for each ambient or leaf process term M_s under m . In the case of a token on the complement place, nothing has to be done and the token is simply passed to the logger for the next step. However, if the token is on any place $M \rightarrow m$ we have to remove this connection and replace it by $M \rightarrow o$ before allowing the next transition to take over. The last transition re-installs the semaphore:

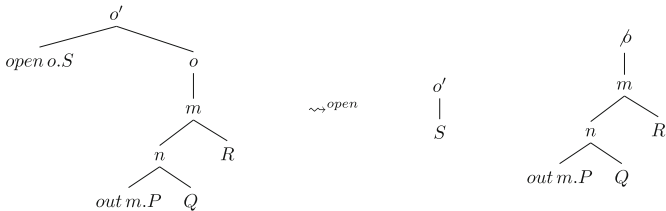


Finally, we are done and can be happy that only two ambients were involved. In fact, the release-chain only depends on the (m, o) -combination while the original *open*-transition depends on the $(o, open.m.P)$ -combination.

2.2.4 The Semaphore's Necessity

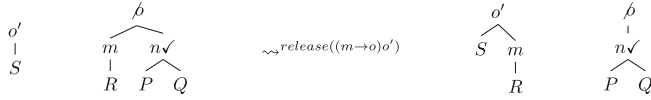
The usage of a semaphore on all transitions that modify the tree avoids races. We illustrate a race condition induced by the parallel execution of *out* and *open* on a child and its parent ambient in the example below.

Example 2.7 (Possible race without semaphore). We consider the process $o'[open\ o.S[o[m[n[out\ m.P[Q]]R]]]$. The initial *open* transition fires and its subsequent $release(o, o')$ sequence waits to be processed in parallel to the *out* m operation:



The ambient o should now vanish and all its children (here only m) should move under o' . If we now assume that the release chain starts its execution and the ambient n is tested ($\checkmark$) before *out* is fired, but

m is not, the token on $n \rightarrow o$ is seen and allows the test chain to proceed. Now out fires, thus n is hung under o . The release step on m correctly puts m below o' but the already tested n remains:



Since the n to o relation was already considered it won't be tested again so that n remains under the actually removed o and we run into an inconsistent system state. The new second root o should actually have vanished and n should be another child of o' . Instead, n 's connection to the main tree is lost.

The strict use of the semaphore prevents such unpleasant behaviour. We now turn our attention to guaranteeing the uniqueness of names.

2.3 Managing Names in the Petri Net

As we pointed out in Example 2.3 the relation by name causes ambiguous trees since several ambients may carry the same name. We now use the ambient name set $\mathcal{A}$ which our Petri net will use instead of the original name. In order to react to the right capabilities the connection to that name is still maintained. We remove the restrictions from the tree by assigning a unique name from the new set of restricted link names $\mathcal{R}$. Their management requires new commands which we incorporate in the MA process equations via a preprocessing. We derive transitions to manage both link and ambient names.

Names and their respective positions store the information in the MA processes. The proof of Turing-completeness for the MA calculus [CG98, p. 149f] encodes the tape into a chain of nested ambients. The ambient's name corresponds to the current symbol on the dedicated tape position.

Example 2.8 (Ambient hierarchy encoding a word).

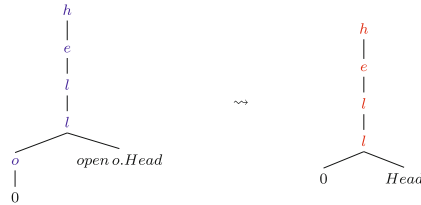


Fig. 2.6 Exemplary word encoding: The word "hello" becomes "hell".

Hence, the saving of name and position information in each ambient is the heart of the information storage in the MA calculus. Capability operations and the addition of new ambients via ambient spawning ($n[P]$) manipulate such information.

2.3.1 Distinguishing Link from Ambient Names

Actually, a name m accomplishes two completely different tasks: It stores information in the ambient tree ($m[\dots]$) via its name and position but it also maintains the connection between the ambients called m and the capability operations ($in\ m$, $out\ m$ and $open\ m$).

We want to separate these concerns in our construction. Hence, we distinguish **link names** ($\mathcal{L}$), which maintain the connection between capabilities and the actual instance in the tree, and the globally unique instance names which we call **ambient names** ($\mathcal{A}$) and which form the tree. We demand $\mathcal{A} \cap \mathcal{L} = \emptyset$.

Example 2.9 (Distinguishing Link from Ambient Names).

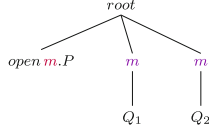


Fig. 2.7 Distinguishing link from ambient names: Link name m and ambient name m .

Link names $\mathcal{L}$ unify all ambients in a scope and represent them in capabilities and process calls. We will enforce link names to be unique, too, so that we can manage their relation to the ambients ($a \mapsto l$) by name and remove the restrictions from the tree. The incorporated ambients of one link name should not be affected by any other link name, nor should any ambient without a corresponding link name exist. Hence, the link names will induce a partition of the ambients into scopes.

Example 2.10 (A capability's scope). In the process $\nu m.(open\ m.P \mid m[Q_1] \mid m[Q_2]) \mid \nu m.(open\ m.P \mid m[R_1] \mid m[R_2])$ each $open\ m$ -capability may only affect the two m -ambients in its restriction's scope. We assign the link names l_1 and l_2 :

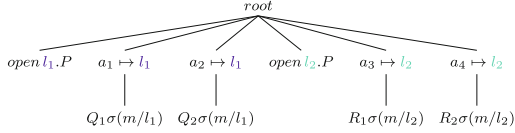


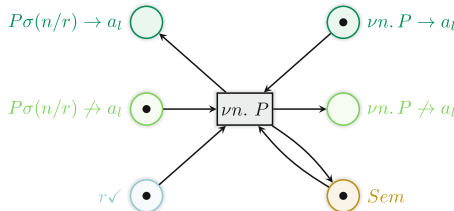
Fig. 2.8 Different scopes replaced by different names.

The omission of restrictions keeps the tree flat. Because of NC only locally restricted names may carry the same name. This makes a distinction between public link names $\mathcal{P}$ and restricted link names $\mathcal{R}$ appropriate. We have $\mathcal{L} = \mathcal{P} \cup \mathcal{R}$.

In summary, link and ambient names are globally unique but link names may occur several times within a process incarnation. They are only present at the leaf level, while the ambient names only occur above this level in the inner tree and at most once there.

2.3.2 Assigning a Restricted Link Name

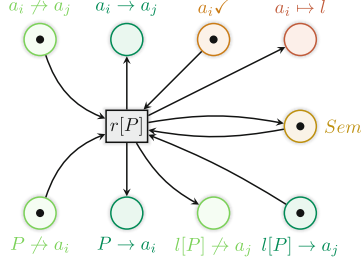
We assign a new restricted link name $r \in \mathcal{R}$ whenever we process a leaf $\nu n.P$: The net non-deterministically chooses a currently unused name $r \in \mathcal{R}$ and substitutes n by r in the whole term P . Since n 's scope can only be intermitted by a new νn which is not allowed within the same process equation due to NC we can substitute n by r on the whole leaf ($\nu n.P \rightsquigarrow P\sigma(n/r)$) without introducing a new restriction. It is necessary to rename the n -ambients as well: this propagates the information on which name r to link. Our substitution also replaces occurrences in process calls and thus makes sure that the name is passed on to subsequent equations.



The token on $r\checkmark$ guarantees that r is currently unused. The transition occupies the name and performs $\nu n.P \rightsquigarrow P\sigma(n/r)$. The inner ambient hierarchy is not touched.

2.3.3 Assigning an Ambient Name

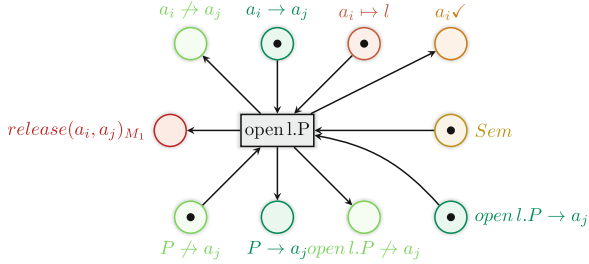
When processing an ambient spawn $l[P]$ we choose an arbitrary unused name $a_i \in \mathcal{A}$ ($a_i \checkmark$) and replace the link name $l \in \mathcal{L}$ by a_i , while not changing P at all. Note that this is no substitution on P but just a very local renaming in one position. We store the knowledge that a_i reacts to l -capabilities on the place $a_i \mapsto l$.



This transition is more complex than the link version since it must also integrate a_i into the ambient tree.

2.3.4 Releasing an Ambient Name

An ambient hierarchy remains even if there is no process running in it any more. This makes the release of an ambient name fairly easy, since the only situation to do so is when processing an *open* capability. There are only minor changes necessary to adapt the *open* procedure presented in Section 2.2.3. We now use the name l and write *open l* to indicate that we work on a link name.



Due to the separation of link and ambient names we now need to check via a place $a_i \mapsto l$ that a_i reacts on l capabilities, namely on *open l*. We mark a_i as unused via $a_i \checkmark$ and erase a_i from the ambient hierarchy by the $\text{release}(a_i, a_j)$ procedure. This procedure remains unchanged in comparison to v2.2.3 and is hence not repeated here.

2.3.5 Releasing a Restricted Link Name

The releasing of a restricted link name is only possible if it is completely unused in the process incarnation P . In the marking representing this incarnation it should neither appear in a leaf process term nor have any ambient linked to it. A marking with a name r which is unused but not marked as unused ($r \checkmark$) corresponds to an MA process term $\nu r.P$ where r does not occur in P . In this analogy our release performs the congruence transformation to P that removes the unnecessary restriction.

We must start by verifying that the name is forgotten among all leaves since a leaf carrying a spawn could easily introduce a new link occurrence in the ambient tree while an *open* command could remove one: As long as there are leaves using a name r the information on its use in the tree is volatile.

2.3.5.1 Incorporating the Necessary Information — the Preprocessing

Whether or not a name is forgotten at the leaf level can best be monitored via the name's spread among the Petri net leaves. A name r 's spread $s(r)$ is the number of Petri net leaves which currently use it. Although the spread's behaviour is totally dynamic the positions at which this value changes are already visible in the process equations. Thus, we can do a preprocessing on the original process equations with the original names. We mark the relevant positions on a name n with the commands *used*(n) and *unused*(n), respectively. These new commands allow the net to log the appropriate changes to a name's spread.

We determine the relevant positions by examining the right-hand side of each defining equation as well as the initial term. At first, only the leaf which carried the restriction νn can know the name n and thus later the chosen instance. Since there is no name passing mechanism like there is in the π calculus the leaf cannot spread its knowledge to other leaves arbitrarily. The name's spread can only increase if the leaf splits itself into two leaves. This can either happen due to a parallel composition or due to a replication. If after a parallel composition $P|Q$ both leaves use the name n we should increase $s(n)$ by 1.

Example 2.11 (Spread increase with parallel compositions). Both process terms following the parallel composition use n so that its spread increases as soon as this parallel composition is no longer prefixed.

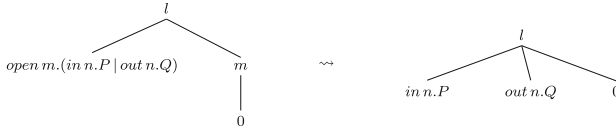


Fig. 2.9 Counting leaves using n : n 's spread changes from 1 to 2.

We can check for each parallel composition $P|Q$ in each defining equation (and in I) whether P and Q both continue to use a name n which was used before the parallel composition. If they do, we put the command *used*(n) before the parallel composition. One could argue that this increases the spread too early, before the parallel composition is actually processed, but since we are only interested in the question whether the spread is above 0 or not this can cause no harm.

However, if we put the *used*(n) command say in front of Q we could reach wrong conclusions: Imagine $s(n) = 1$, that is the leaf carrying $P|Q$ is the only leaf knowing n . If we now freed P for execution it could easily reach and execute its *unused*(n) out of which the net would conclude $s(n) = 0$ although Q still knows n but did not tell the net yet.

The dealing with the replication $!P$ is similar to that for the parallel composition: we must put a used command right behind the replication sign $!$ for each name occurring in P . But since a name occurring in a replication term can never become unused our spread count is useless there. We only keep it to exclude the wrong conclusion of a spread of 0 which may otherwise arise just like it may in the parallel composition's case.

The *unused* commands are put behind each last occurrence of a name on the right-hand side of a process equation. Due to *minPara* in Definition 1.5 this last occurrence is always either a capability or a spawn. Thus, we check behind each capability πn and spawn $n[Q]$ whether n is used in the following process term Q . If it is not used we put the *unused*(n) command before Q .

2.3.5.2 Forgotten Among All Leaves? — Maintaining a Name's Spread

The new commands are executed during runtime and thus performed on the instance name r . As soon as a spread declines to 0 we immediately know that the respective name is forgotten among all leaves ($r \checkmark_{\text{leaves}}$). The one-safeness requires us to use one place $s(r) = i$ per possible spread value i . Whenever we process a restriction $\nu n.P$ we initialise the spread of the chosen name r with 1.

A Polynomial Translation of Mobile Ambients into Safe
Petri Nets

Understanding a Calculus of Hierarchical Protection
Domains

Göbel, S.

2016, IX, 66 p. 15 illus. in color., Softcover

ISBN: 978-3-658-11764-1